MX



macromedia®
COLDFUSION® MX 7

ColdFusion MX Developer's Guide

Trademarks

1 Step RoboPDF, ActiveEdit, ActiveTest, Authorware, Blue Sky Software, Blue Sky, Breeze, Breezo, Captivate, Central, ColdFusion, Contribute, Database Explorer, Director, Dreamweaver, Fireworks, Flash, FlashCast, FlashHelp, Flash Lite, FlashPaper, Flex, Flex Builder, Fontographer, FreeHand, Generator, HomeSite, JRun, MacRecorder, Macromedia, MXML, RoboEngine, RoboHelp, RoboInfo, RoboPDF, Roundtrip, Roundtrip HTML, Shockwave, SoundEdit, Studio MX, UltraDev, and WebHelp are either registered trademarks or trademarks of Macromedia, Inc. and may be registered in the United States or in other jurisdictions including internationally. Other product names, logos, designs, titles, words, or phrases mentioned within this publication may be trademarks, service marks, or trade names of Macromedia, Inc. or other entities and may be registered in certain jurisdictions including internationally.

This product includes code licensed from RSA Data Security.

Third-Party Information

This guide contains links to third-party websites that are not under the control of Macromedia, and Macromedia is not responsible for the content on any linked site. If you access a third-party website mentioned in this guide, then you do so at your own risk. Macromedia provides these links only as a convenience, and the inclusion of the link does not imply that Macromedia endorses or accepts any responsibility for the content on those third-party sites.

Copyright © 1999–2005 Macromedia, Inc. All rights reserved. U.S. Patents Pending. This manual may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without written approval from Macromedia, Inc. Notwithstanding the foregoing, the owner or authorized user of a valid copy of the software with which this manual was provided may print out one copy of this manual from an electronic version of this manual for the sole purpose of such owner or authorized user learning to use such software, provided that no part of this manual may be printed out, reproduced, distributed, resold, or transmitted for any other purposes, including, without limitation, commercial purposes, such as selling copies of this documentation or providing paid-for support services.
Part Number ZCF70M500

Acknowledgments

Project Management: Randy Nielsen

Writing: Hal Lichtin, Randy Nielsen, Robert Berry, Chris Bedford, Anne Sandstrom

Editing: Linda Adler, Noreen Maher

Production Management: Patrice O'Neill

Media Design and Production: John Francis, Adam Barnett

Special thanks to Sawako Gensure, Seungmin Lee, Takashi Koto, Nozomi Kugita, Masayo Noda, Hiroshi Okugawa, Bowne Global Solutions

First Edition: January 2005

Macromedia, Inc.
600 Townsend St.
San Francisco, CA 94103

CONTENTS

INTRODUCTION: About ColdFusion MX Documentation 13
Using this manual 13

CHAPTER 1: Introducing ColdFusion MX 19
About Internet applications and web application servers 19
About ColdFusion MX 22
About J2EE and the ColdFusion architecture 24

PART I: The CFML Programming Language

CHAPTER 2: Elements of CFML 27
CFML Basics 28
Comments 28
Tags 29
Functions 31
ColdFusion components 32
Constants 32
Variables 33
Expressions 34
Data types 35
Flow control 35
Character case 39
Special characters 39
Reserved words 39
CFScript 40

CHAPTER 3: Using ColdFusion Variables 43
Creating variables 44
Variable characteristics 45
Data types 45
Strings 47
Using periods in variable references 56
Data type conversion 59

About scopes 64
Ensuring variable existence 70
Validating data 73
Passing variables to custom tags and UDFs 73

CHAPTER 4: Using Expressions and Number Signs 75

Expressions 75
Using number signs 81
Dynamic expressions and dynamic variables 84

CHAPTER 5: Using Arrays and Structures 97

About arrays 97
Basic array techniques 100
Populating arrays with data 105
Array functions 108
About structures 108
Creating and using structures 111
Structure examples 117
Structure functions 121

CHAPTER 6: Extending ColdFusion Pages with CFML Scripting 123

About CFScript 123
The CFScript language 125
Using CFScript statements 128
Handling exceptions 136
CFScript example 137

CHAPTER 7: Using Regular Expressions in Functions 141

About regular expressions 141
Regular expression syntax 143
Using backreferences 150
Returning matched subexpressions 153
Regular expression examples 157
Types of regular expression technologies 158

PART II: Building Blocks of ColdFusion Applications

CHAPTER 8: Creating ColdFusion Elements 161

About CFML elements you create 161
Including pages with the cfinclude tag 162
About user-defined functions 164
Using ColdFusion components 165
Using custom CFML tags 166
Using CFX tags 167
Selecting among ColdFusion code reuse methods 169

CHAPTER 9: Writing and Calling User-Defined Functions. 171

About user-defined functions 171
Creating user-defined functions. 172
Calling user-defined functions. 178
Working with arguments and variables in functions 178
Handling errors in UDFs 186
A user-defined function example 192
Using UDFs effectively 194

CHAPTER 10: Building and Using ColdFusion Components 201

About ColdFusion components. 201
Creating ColdFusion components 204
Using ColdFusion components 216
Passing parameters to methods 225
CFC variables and scope 227
Using CFCs effectively 230
ColdFusion component example 239

CHAPTER 11: Creating and Using Custom CFML Tags 241

Creating custom tags 241
Passing data to custom tags 245
Managing custom tags. 249
Executing custom tags 250
Nesting custom tags 253

CHAPTER 12: Building Custom CFXAPI Tags 259

What are CFX tags?. 259
Before you begin developing CFX tags in Java. 260
Writing a Java CFX tag 261
ZipBrowser example 264
Approaches to debugging Java CFX tags 266
Developing CFX tags in C++ 269

PART III: Developing CFML Applications

CHAPTER 13: Designing and Optimizing a ColdFusion Application 275

About applications 275
Elements of a ColdFusion application 276
Structuring an application 279
Defining the application and its event handlers in Application.cfc. 282
Migrating from Application.cfm to Application.cfc 294
Using an Application.cfm page 295
Optimizing ColdFusion applications. 298

CHAPTER 14: Handling Errors 307
About error handling in ColdFusion 307
Understanding errors 308
Error messages and the standard error format 314
Determining error-handling strategies 314
Specifying custom error messages with the cferror tag 317
Logging errors with the cflog tag 321
Handling runtime exceptions with ColdFusion tags 322

CHAPTER 15: Using Persistent Data and Locking 339
About persistent scope variables 340
Managing the client state 342
Configuring and using client variables 346
Configuring and using session variables 352
Configuring and using application variables 357
Using server variables 359
Locking code with cflock 360
Examples of cflock 368

CHAPTER 16: Securing Applications 373
ColdFusion security features 374
About resource and sandbox security 375
About user security 376
Using ColdFusion security tags and functions 381
Security scenarios 386
Implementing user security 388

CHAPTER 17: Developing Globalized Applications 403
Introduction to globalization 403
About character encodings 406
Locales 408
Processing a request in ColdFusion 410
Tags and functions for globalizing 413
Handling data in ColdFusion MX 416

CHAPTER 18: Debugging and Troubleshooting Applications 423
Configuring debugging in the ColdFusion MX Administrator 423
Using debugging information from browser pages 426
Controlling debugging information in CFML 434
Using the cftrace tag to trace execution 436
Using the cftimer tag to time blocks of code 440
Using the Code Compatibility Analyzer 442
Troubleshooting common problems 443

PART IV: Accessing and Using Data

CHAPTER 19: Introduction to Databases and SQL 447

What is a database? 447
Using SQL 452
Writing queries using an editor 460

CHAPTER 20: Accessing and Retrieving Data 465

Working with dynamic data 465
Retrieving data 466
Outputting query data 469
Getting information about query results 471
Enhancing security with cfqueryparam 472

CHAPTER 21: Updating Your Database 475

About updating your database 475
Inserting data 475
Updating data 480
Deleting data 487

CHAPTER 22: Using Query of Queries 489

About record sets 489
About Query of Queries 491
Query of Queries user guide 497

CHAPTER 23: Managing LDAP Directories 515

About LDAP 516
The LDAP information structure 517
Using LDAP with ColdFusion 520
Querying an LDAP directory 521
Updating an LDAP directory 527
Advanced topics 538

CHAPTER 24: Building a Search Interface 547

About Verity 547
Creating a search tool for ColdFusion applications 553
Creating a search page 561
Enhancing search results 564
Working with data returned from a query 572

CHAPTER 25: Using Verity Search Expressions 581
About Verity query types 581
Using simple queries 582
Using explicit queries 583
Using natural queries 587
Using Internet queries 587
Composing search expressions 591
Refining your searches with zones and fields 601

PART V: Requesting and Presenting Information

CHAPTER 26: Introduction to Retrieving and Formatting Data 609
Using forms in ColdFusion MX 609
Working with action pages 613
Working with queries and data 617
Returning results to the user 621
Dynamically populating list boxes 625
Creating dynamic check boxes and multiple-selection list boxes 627

CHAPTER 27: Building Dynamic Forms with cfform Tags 631
Creating custom forms with the cfform tag 631
Building tree controls with the cftree tag 634
Building drop-down list boxes 642
Building slider bar controls 643
Creating data grids with the cfgrid tag 644
Embedding Java applets 655

CHAPTER 28: Validating Data 659
About ColdFusion MX validation 659
Validating form fields 667
Handling invalid data 669
Masking form input values 670
Validating form data with regular expressions 671
Validating form data using hidden fields 675
Validating form input and handling errors with JavaScript 680
Validating data with the IsValid function and the cfparam tag 683

CHAPTER 29: Creating Forms in Macromedia Flash 687
About Flash forms 687
Building Flash forms 689
Binding data in Flash forms 699
Setting styles and skins in Flash forms 700
Using ActionScript in Flash forms 704
Best practices for Flash forms 707

CHAPTER 30: Creating Skinnable XML Forms 709
About XML skinnable forms 709
Building XML skinnable forms 712
ColdFusion XML format. 715
Creating XSLT skins 728

CHAPTER 31: Creating Charts and Graphs. 733
About charts 733
Creating a basic chart 734
Charting data 735
Controlling chart appearance 743
Creating charts: examples 751
Administering charts 756
Writing a chart to a variable 756
Linking charts to URLs 758

CHAPTER 32: Creating Reports for Printing 763
About printable output 763
Creating PDF and FlashPaper output with the cfdocument tag. 764
Creating reports with the ColdFusion MX 7 reporting 770
Font management with printable reports 804
Creating reports with Crystal Reports (Windows only) 805

CHAPTER 33: Using the Flash Remoting Service 807
About using the Flash Remoting service with ColdFusion 807
Configuring the Flash Remoting Gateway 809
Using the Flash Remoting service with ColdFusion pages 812
Using Flash with CFCs 817
Using the Flash Remoting service with ColdFusion Java objects 819
Handling errors with ColdFusion and Flash 820

CHAPTER 34: Using Server-Side ActionScript 823
About server-side ActionScript 823
Connecting to the Flash Remoting service. 826
Using server-side ActionScript functions 827
Global and request scope objects 829
About the CF.query function and data sources 829
Using the CF.query function 831
Building a simple application 833
About the CF.http function 836
Using the CF.http function 837

PART VI: Using Web Elements and External Objects

CHAPTER 35: Using XML and WDDX 843

About XML and ColdFusion 843
The XML document object 844
ColdFusion XML tag and functions 850
Using an XML object 852
Creating and saving an XML document object 856
Modifying a ColdFusion XML object 858
Validating XML documents 868
Transforming documents with XSLT 869
Extracting data with XPath 869
Example: using XML in a ColdFusion application 870
Moving complex data across the web with WDDX 875
Using WDDX 879

CHAPTER 36: Using Web Services 885

Web services 885
Working with WSDL files 888
Consuming web services 891
Publishing web services 898
Using request and response headers 907
Handling complex data types 909
Troubleshooting SOAP requests and responses 914

CHAPTER 37: Integrating J2EE and Java Elements in CFML Applications 917

About ColdFusion, Java, and J2EE 917
Using JSP tags and tag libraries 921
Interoperating with JSP pages and servlets 922
Using Java objects 928

CHAPTER 38: Integrating COM and CORBA Objects in CFML Applications 945

About COM and CORBA 945
Creating and using objects 947
Getting started with COM and DCOM 948
Creating and using COM objects 951
Getting started with CORBA 961
Creating and using CORBA objects 961
CORBA example 969

PART VII: Using External Resources

CHAPTER 39: Sending and Receiving E-Mail 973

Using ColdFusion with mail servers 973
Sending e-mail messages 974
Sample uses of the cfmail tag 977
Using the cfmailparam tag 981
Receiving e-mail messages 983
Handling POP mail 985

CHAPTER 40: Interacting with Remote Servers 991

About interacting with remote servers 991
Using cfhttp to interact with the web 992
Creating a query object from a text file 994
Using the cfhttp Post method 996
Performing file operations with cfftp 999

CHAPTER 41: Managing Files on the Server 1003

About file management 1003
Using cffile 1003
Using cfdirectory 1012
Using cfcontent 1014

CHAPTER 42: Using Event Gateways 1019

About event gateways 1020
Event gateway facilities and tools 1024
Structure of an event gateway application 1027
Configuring an event gateway instance 1028
Developing an event gateway application 1029
Deploying event gateways and applications 1037
Using the CFML event gateway for asynchronous CFCs 1038
Using the example event gateways and gateway applications 1040

CHAPTER 43: Using the Instant Messaging Event Gateways 1047

About ColdFusion and instant messages 1047
Configuring an IM event gateway 1050
Handling incoming messages 1052
Sending outgoing messages 1053
Sample IM message handling application 1054
Using the GatewayHelper object 1058

CHAPTER 44: Using the SMS Event Gateway 1065
About SMS and ColdFusion 1065
Configuring an SMS event gateway 1071
Handling incoming messages 1073
Sending outgoing messages 1075
ColdFusion MX SMS development tools 1080
Sample SMS application 1082

CHAPTER 45: Creating Custom Event Gateways 1085
Event gateway architecture 1085
Event gateway elements 1086
Building an event gateway 1092
Deploying an event gateway 1100

INDEX 1103

INTRODUCTION
About ColdFusion MX Documentation

ColdFusion MX Developer's Guide provides the tools needed to develop Internet applications using Macromedia ColdFusion MX. This manual is intended for web application programmers who are learning ColdFusion MX or wish to extended their ColdFusion MX programming knowledge. It provides a solid grounding in the tools that ColdFusion MX provides to develop web applications.

Because of the power and flexibility of ColdFusion MX, you can create many different types of web applications of varying complexity. As you become more familiar with the material presented in this manual, and begin to develop your own applications, you will want to refer to *CFML Reference* for details about various tags and functions.

Contents

Using this manual 13

Accessing the ColdFusion MX documentation. 18

Using this manual

This manual can to help anyone with a basic understanding of HTML learn to develop ColdFusion MX applications. However, this manual is most useful if you have basic ColdFusion experience, or have read *Getting Started Building ColdFusion MX Applications*. The *Getting Started Building ColdFusion MX Applications* manual provides an introduction to ColdFusion and helps you develop the basic knowledge that will make using this manual easier.

Approaches to using this manual

This section describes approaches to using this manual for beginning ColdFusion developers, developers with some experience who want to develop expertise, and advanced developers who want to learn about the new and enhanced features of ColdFusion MX.

Beginning with ColdFusion

If you are learning ColdFusion, it might be most effective to read this manual in the following order:

1. Chapter 1, "Introducing ColdFusion MX" through Chapter 4, "Using Expressions and Number Signs" to learn the basics of CFML.
2. Chapter 19, "Introduction to Databases and SQL" through Chapter 21, "Updating Your Database" to learn about using databases.
3. Chapter 26, "Introduction to Retrieving and Formatting Data" and Chapter 27, "Building Dynamic Forms with cfform Tags" to learn about requesting data from users.

   After you read these chapters, you should have a basic understanding of the basic elements of ColdFusion, and be able to create simple ColdFusion applications. To learn to produce more complete and robust applications, you could proceed with the following chapters:
4. Chapter 13, "Designing and Optimizing a ColdFusion Application" through Chapter 18, "Debugging and Troubleshooting Applications" to learn how to build a complete ColdFusion application.
5. Chapter 22, "Using Query of Queries" to learn how to use queries effectively.
6. Chapter 5, "Using Arrays and Structures" through Chapter 10, "Building and Using ColdFusion Components" to learn to use more advanced features of CFML, including ways to reuse code.

You can then read the remaining chapters as you add new features to your ColdFusion application.

Developing an in-depth knowledge of ColdFusion

If you have a basic understanding of ColdFusion as presented in *Getting Started Building ColdFusion MX Applications* or the Fast Track to ColdFusion course, you might want to start at Chapter 1 and work through to the end of the book, skipping any specialized chapters that you are unlikely to need.

Learning about new and modified ColdFusion features

If you are an advanced ColdFusion developer, you might want to learn about new or changed ColdFusion features. The following chapters document features that are new or substantially enhanced in ColdFusion MX 7:

- Chapter 24, "Building a Search Interface"
- Chapter 25, "Using Verity Search Expressions"
- Chapter 27, "Building Dynamic Forms with cfform Tags"
- Chapter 28, "Validating Data"
- Chapter 29, "Creating Forms in Macromedia Flash"
- Chapter 30, "Creating Skinnable XML Forms"
- Chapter 32, "Creating Reports for Printing"

- Chapter 42, "Using Event Gateways"
- Chapter 43, "Using the Instant Messaging Event Gateways"
- Chapter 44, "Using the SMS Event Gateway"
- Chapter 45, "Creating Custom Event Gateways"

Nearly all chapters contain information that is new in ColdFusion MX 7, so you should also review all other chapters for useful information. The index and the table of contents are useful tools for finding new features or changed documentation.

Note: If you are moving from ColdFusion 4.5 or 5, start by reading *Migrating ColdFusion 5 Applications*, available on the Macromedia website.

ColdFusion features described in this manual

ColdFusion provides a comprehensive set of features for developing and managing Internet applications. These features enhance speed and ease-of-development, and let you dynamically deploy your applications, integrate new and legacy technologies, and build secure applications.

The following table describes the primary ColdFusion features that are discussed in this manual, and lists the chapters that describe them. This table is only a summary of major CFML features; this manual also includes information about other features. Also, this table does not include features that are described in other manuals.

Feature	Description	Chapters
CFML	CFML is a fully featured tag-oriented Internet application language. It includes a wide range of tags, functions, variables, and expressions.	2-5
CFScript	CFScript is a server-side scripting language that provides a subset of ColdFusion functionality in script syntax.	6
Regular expressions	ColdFusion provides several functions that use regular expressions for string manipulation. It also lets you use regular expressions in text input tags.	7, 28
Reusable elements	ColdFusion lets you create several types of elements, such as user-defined functions and ColdFusion components, that you write once and can use many times.	8-12
User-defined functions (UDFs)	You can use CFScript or the `cffunction` tag to create your own functions. These functions can incorporate all of the built-in ColdFusion tags and functions, plus other extensions.	9
ColdFusion components	ColdFusion components encapsulate multiple functions and related data in a single logical unit. ColdFusion components can have many uses, and are particularly useful in creating web services and Flash interfaces for your application.	10
Custom CFML tags	You can create custom ColdFusion tags using CFML. These tags can have bodies and can call other custom tags.	11
ColdFusion extension (CFX) tags	You can create custom tags in Java or C++. These tags can use features that are only available when using programming languages. However, CFX tags cannot have tag bodies.	12

Feature	Description	Chapters
ColdFusion application structure	ColdFusion supports many ways of building an application, and includes specific features, such as the Application.cfc file or Application.cfm page, built-in security features, and shared scopes, that help you optimize your application structure.	13-17
Error handling mechanisms	ColdFusion provides several mechanisms for handling data, including custom error pages and exception-handling tags and functions, such as `cftry` and `cfcatch`.	14
Shared and persistent variable scopes	Using shared and persistent scopes, you can make data available to a single user over one or many browser sessions, or to multiple users of an application or server.	15
Code locking	You lock sections of code that access in-memory shared scopes or use external resources that are not safe for multiple simultaneous access.	15
Application security	ColdFusion provides mechanisms, including the `cflogin` tag, for authenticating users and authorizing them to access specific sections of your application. You can also use resource security, which secures access to ColdFusion resources based on the ColdFusion page location.	16
Application globalization	ColdFusion supports global applications that use different character sets and locales, and provides tags and functions designed to support globalizing your applications.	17
Debugging tools	Using debugging output, the `cftrace` tag, logging features, and the Code Analyzer, you can locate and fix coding errors.	18
Database access and management	ColdFusion can access SQL databases to retrieve, add, and modify data. This feature is one of the core functions of many dynamic applications.	19-21
Queries of Queries	You can use a subset of standard SQL within ColdFusion to manipulate any data that is represented as a record set, including database query results, LDAP (Lightweight Directory Access Protocol) directory information, and other data.	22
LDAP directory access and management	ColdFusion applications can access and manipulate data in LDAP directory services. These directories are often used for security validation data and other directory-like information.	23
Indexing and searching data	ColdFusion applications can provide full-text search capabilities for documents and data sources using the Verity search engine.	24-25
Dynamic forms	With ColdFusion, you can use HTML and forms to control the data displayed by a dynamic web page. You can also use the `cfform` tag to enrich your forms with sophisticated graphical controls, and perform input data validation.	26-30
Validating data	ColdFusion provides several ways to validate data in forms and in CFML variables.	28
Flash forms	ColdFusion can display forms using Macromedia Flash, which presents a pleasing appearance and includes features such as accordion and tab navigators.	29

Feature	Description	Chapters
XML skinnable forms	ColdFusion can convert your CFML forms into XML and format the XML using XSLT skins and style sheets.	30
Data graphing	You can use the `cfchart` tag to display your data graphically.	31
Reports and printable output	You can create output that is formatted for print as PDF or FlashPaper documents. You can also use ColdFusion reporting to create banded reports for display or printing.	32
Macromedia Flash integration	You can use native Flash connectivity built into ColdFusion to help build dynamic Flash user interfaces for ColdFusion applications.	33
Server-side ActionScript	Macromedia Flash Remoting lets Macromedia Flash MX developers create server-side ActionScript. ActionScript files can directly access ColdFusion query and HTTP features through two functions: `CF.query` and `CF.http`.	34
XML document processing and creation	ColdFusion applications can create, use, and manipulate XML (Extensible Markup Language) documents. ColdFusion also provides tools to use WDDX (Web Distributed Data Exchange), an XML dialect for transmitting structured data.	35
Web services	ColdFusion applications can use available SOAP (Simple Object Access Protocol)-based web services, including Microsoft .NET services. ColdFusion applications can also use ColdFusion components to provide web services to other applications over the Internet.	36
Java and J2EE integration	You can integrate J2EE elements, including JSP (JavaServer Pages) pages, JSP tag libraries, and Java objects, including EJBs (Enterprise JavaBeans), into your ColdFusion application.	37
COM and CORBA objects	The `cfobject` tag lets you use COM (Component Object Model) or DCOM (Distributed Component Object Model) and CORBA (Common Object Request Broker) objects in your ColdFusion applications.	38
E-mail messages	You can add interactive e-mail features to your ColdFusion applications using the `cfmail` and `cfpop` tags.	39
HTTP and FTP	The `cfhttp` and `cfftp` tags provide simple methods of using HTTP (Hypertext Transfer Protocol) and FTP (File Transfer Protocol) communications in your application.	40
File and directory access	You can use the `cffile`, `cfdirectory`, and `cfcontent` tags to read, write, and manage files and directories on the server.	41
Event gateways	ColdFusion event gateways let ColdFusion applications asynchronously react to or generate external events or messages.	42-45
Instant messaging event gateways	Your ColdFusion application can communicate with XMPP (Jabber) or IBM Sametime instant messaging clients using the ColdFusion IM gateways.	43

Feature	Description	Chapters
SMS event gateway	Your ColdFusion MX application can communicate with short message service (SMS) devices, such as mobile phones, using the ColdFusion SMS event gateway.	44
Creating gateways	You can write your own event gateways in Java and integrate them into ColdFusion MX.	45

Accessing the ColdFusion MX documentation

The ColdFusion MX documentation is designed to provide support for the complete spectrum of participants.

Documentation set

The ColdFusion MX 7 documentation set includes the following titles:

Book	Description
Installing and Using ColdFusion MX	Describes system installation and basic configuration for Windows, Solaris, and Linux. To see this manual, go to www.macromedia.com/go/livedocs_cfmx7docs_installing.
Configuring and Administering ColdFusion MX	Part I describes how to manage the ColdFusion environment, including connecting to your data sources and configuring security for your applications. Part II describes Verity search tools and utilities that you can use for configuring the Verity Search Server engine, as well as creating, managing, and troubleshooting Verity collections. To see this manual, go to www.macromedia.com/go/livedocs_cfmx7docs_configadmin.
ColdFusion MX Developer's Guide	Describes how to develop your dynamic web applications, including retrieving and updating your data, using structures, and forms. This manual includes two volumes. To see this manual, go to www.macromedia.com/go/livedocs_cfmx7docs_dev.
Getting Started Building ColdFusion MX Applications	Contains an overview of ColdFusion features and application development procedures. Includes a tutorial that guides you through the process of developing an example ColdFusion application. To see this manual, go to www.macromedia.com/go/livedocs_cfmx7docs_gs.
CFML Reference	Provides descriptions, syntax, usage, and code examples for all ColdFusion tags, functions, and variables. This manual includes two volumes. To see this manual, go to www.macromedia.com/go/livedocs_cfmx7docs__cfml_reference.
CFML Quick Reference	Provides a brief guide that shows the syntax of ColdFusion tags, functions, and variables.

Viewing online documentation

All ColdFusion MX documentation is available online in HTML and Adobe Acrobat Portable Document Format (PDF) files. Go to the documentation home page for ColdFusion MX on the Macromedia website: www.macromedia.com.

CHAPTER 1
Introducing ColdFusion MX

This chapter describes Macromedia ColdFusion MX and the role it plays in developing dynamic Internet applications. This chapter also introduces the topics discussed in this manual.

Contents

About Internet applications and web application servers 19
About ColdFusion MX 22
About J2EE and the ColdFusion architecture. 24

About Internet applications and web application servers

With ColdFusion MX, you develop Internet applications that run on web application servers. The following sections introduce Internet applications and web application servers. Later sections explain the specific role that ColdFusion MX plays in this environment.

About web pages and Internet applications

The Internet has evolved from a collection of static HTML pages to an application deployment platform. First, the Internet changed from consisting of static web pages to providing dynamic, interactive content. Rather than providing unchanging content where organizations merely advertise goods and services, dynamic pages enable companies to conduct business ranging from e-commerce to managing internal business processes. For example, a static HTML page lets a bookstore publish its location, list services such as the ability to place special orders, and advertise upcoming events like book signings. A dynamic website for the same bookstore lets customers order books online, write reviews of books they read, and even get suggestions for purchasing books based on their reading preferences.

More recently, the Internet has become the underlying infrastructure for a wide variety of applications. With the arrival of technologies such as XML, web services, J2EE (Java 2 Platform, Enterprise Edition), and Microsoft .NET, the Internet has become a multifaceted tool for integrating business activities. Now, enterprises can use the Internet to integrate distributed activities, such as customer service, order entry, order fulfillment, and billing.

ColdFusion MX is a rapid application development environment that lets you build dynamic websites and Internet applications quickly and easily. It lets you develop sophisticated websites and Internet applications without knowing the details of many complex technologies, yet it lets advanced developers take advantage of the full capabilities of many of the latest Internet technologies.

About web application servers

To understand ColdFusion, you must first understand the role of web application servers. Typically, web browsers make requests, and web servers, such as Microsoft Internet Information Server (IIS) and the Apache web server, fulfill those requests by returning the requested information to the browser. This information includes, but is not limited to, HTML and Macromedia Flash files.

A web server's capabilities are limited because all it does is wait for requests to arrive and attempt to fulfill those requests as soon as possible. A web server does not let you do the following tasks:

- Interact with a database, other resource, or other application.
- Serve customized information based on user preferences or requests.
- Validate user input.

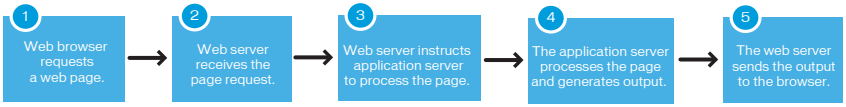
A web server, basically, locates information and returns it to a web browser.

To extend the capabilities of a web server, you use a *web application server*, a software program that extends the web server's capabilities to do tasks such as those in the preceding list.

How a web server and web application server work together

The following steps explain how a web server and web application server work together to process a page request:

1. The user requests a page by typing a URL in a browser, and the web server receives the request.
2. The web server looks at the file extension to determine whether a web application server must process the page. Then, one of the following actions occur:
   - If the user requests a file that is a simple web page (often one with an HTM or HTML extension), the web server fulfills the request and sends the file to the browser.
   - If the user requests a file that is a page that a web application server must process (one with a CFM, CFML, or CFC extension for ColdFusion requests), the web server passes the request to the web application server. The web application server processes the page and sends the results to the web server, which returns those results to the browser. The following figure shows this process:



Because web application servers interpret programming instructions and generate output that a web browser can interpret, they let web developers build highly interactive and data-rich websites, which can do tasks such as the following:

- Query other database applications for data.
- Dynamically populate form elements.
- Dynamically generate Flash application data.
- Provide application security.
- Integrate with other systems using standard protocols such as HTTP, FTP, LDAP, POP, and SMTP.
- Create shopping carts and e-commerce websites.
- Respond with an e-mail message immediately after a user submits a form.
- Return the results of keyword searches.

About ColdFusion MX

ColdFusion MX is a rapid scripting environment server for creating dynamic Internet Applications. ColdFusion Markup Language (CFML) is an easy-to-learn tag-based scripting language, with connectivity to enterprise data and powerful built-in search and charting capabilities. ColdFusion MX enables developers to easily build and deploy dynamic websites, content publishing systems, self-service applications, commerce sites, and more.

ColdFusion MX consists of the following core components:

- ColdFusion scripting environment
- CFML
- ColdFusion MX Administrator
- Verity Search Server

The following sections describe these core components in more detail.

The ColdFusion scripting environment

The ColdFusion scripting environment provides an efficient development model for Internet applications. At the heart of the ColdFusion scripting environment is the ColdFusion Markup Language (CFML), a tag-based programming language that encapsulates many of the low-level details of web programming in high-level tags and functions.

ColdFusion Markup Language

ColdFusion Markup Language (CFML) is a tag-based language, similar to HTML, that uses special tags and functions. With CFML, you can enhance standard HTML files with database commands, conditional operators, high-level formatting functions, and other elements to rapidly produce easy-to-maintain web applications. However, CFML is not limited to enhancing HTML. For example, you can create Macromedia Flash MX applications that consist entirely of Flash elements and CFML. Similarly, you can use CFML to create web services for use by other applications.

The following sections briefly describe basic CFML elements. For more information, see Chapter 2, “Elements of CFML,” on page 27.

CFML tags

CFML looks similar to HTML—it includes starting and, in most cases, ending tags, and each tag is enclosed in angle brackets. All ending tags are preceded with a forward slash (/) and all tag names are preceded with `cf`; for example:

```
<cftagname>
  tag body text and CFML
</cftagname>
```

CFML increases productivity by providing a layer of abstraction that hides many low-level details involved with Internet application programming. At the same time, CFML is extremely powerful and flexible. ColdFusion lets you easily build applications that integrate files, databases, legacy systems, mail servers, FTP servers, objects, and components.

CFML includes approximately 110 tags. ColdFusion tags serve many functions. They provide programming constructs, such as conditional processing and loop structures. They also provide services, such as charting and graphing, full-text search, access to protocols such as FTP, SMTP/POP, and HTTP, and much more. The following table lists a few examples of commonly used ColdFusion tags:

Tag	Purpose
`cfquery`	Establishes a connection to a database (if one does not exist), executes a query, and returns results to the ColdFusion environment.
`cfoutput`	Displays output that can contain the results of processing ColdFusion functions, variables, and expressions.
`cfset`	Sets the value of a ColdFusion variable.
`cfmail`	Lets an application send SMTP mail messages using application variables, query results, or server files. (Another tag, `cfpop`, gets mail.)
`cfchart`	Converts application data or query results into graphs, such as bar charts or pie charts, in Flash, JPG, or PNG format.
`cfobject`	Invokes objects written in other programming languages, including COM (Component Object Model) components, Java objects such as Enterprise JavaBeans, or Common CORBA (Object Request Broker Architecture) objects.

CFML Reference describes the CFML tags in detail.

CFML functions and CFScript

CFML includes approximately 280 built-in functions. These functions perform a variety of roles, including string manipulation, data management, and system functions. CFML also includes a built-in scripting language, CFScript, that lets you write code in a manner that is familiar to programmers and JavaScript writers.

CFML extensions

You can extend CFML further by creating custom tags or user-defined functions (UDFs), or by integrating COM, C++, and Java components (such as JSP tag libraries). You can also create ColdFusion components (CFCs), which encapsulate related functions and properties and provide a consistent interface for accessing them.

All these features let you easily create reusable functionality that is customized to the types of applications or websites that you are building.

CFML development tools

Macromedia Dreamweaver MX 2004 helps you develop ColdFusion applications efficiently. It includes many features that simplify and enhance ColdFusion development, including tools for debugging CFML. Because CFML is written in an HTML-like text format, and you often use HTML in ColdFusion pages, you can also use an HTML editor or a text editor, such as Notepad, to write ColdFusion applications.

Verity Search Server

The Verity Search Server (also called the Verity search engine) provides full text search capability for documents and data on a ColdFusion MX site.

ColdFusion MX Administrator

ColdFusion MX Administrator configures and manages the ColdFusion application server. It is a secure web-based application that you can access using any web browser, from any computer with an Internet connection.

For more information about ColdFusion MX Administrator, see *Configuring and Administering ColdFusion MX*.

About J2EE and the ColdFusion architecture

As the Internet software market has matured, the infrastructure services required by distributed Internet applications, including ColdFusion applications, have become increasingly standardized. The most widely adopted standard today is the Java 2 Platform, Enterprise Edition (J2EE) specification. J2EE provides a common set of infrastructure services for accessing databases, protocols, and operating system functionality, across multiple operating systems.

About ColdFusion MX and the J2EE platform

ColdFusion MX is implemented on the Java technology platform and uses a J2EE application server for many of its base services, including database connectivity, naming and directory services, and other runtime services. ColdFusion MX can be configured to use an embedded J2EE server (in the *server configuration*) or it can be deployed as a J2EE application on an independent J2EE application server (in the *multiserver configuration* or the *J2EE configuration*). ColdFusion MX Enterprise includes a fully featured version of the Macromedia JRun J2EE application server, or can be deployed on third-party J2EE servers such as IBM WebSphere and BEA WebLogic.

For more information on ColdFusion MX configurations, see *Installing and Using ColdFusion MX*.

By implementing the ColdFusion scripting environment on top of the J2EE platform, ColdFusion MX takes advantage of the power of the J2EE platform while also providing an easy-to-use scripting environment and built-in services. Moreover, because ColdFusion is built on a J2EE platform, you can easily integrate J2EE and Java functionality into your ColdFusion application. As a result, ColdFusion pages can do any of the following:

- Share session data with JSPs (Java Server Pages) and Java servlets.
- Import custom JSP tag libraries and use them like ColdFusion custom tags.
- Integrate with Java objects, including the J2EE Java API, JavaBeans, and Enterprise JavaBeans.

For more information on using J2EE features in ColdFusion, see Chapter 37, “Integrating J2EE and Java Elements in CFML Applications,” on page 917.

PART I
The CFML Programming Language

This part describes the elements of the CFML programming language. It tells you how to use CFML tags, functions, variables and expressions, the CFScript scripting language, and regular expressions.

The following chapters are included:

Chapter 2: Elements of CFML. 27
Chapter 3: Using ColdFusion Variables 43
Chapter 4: Using Expressions and Number Signs. 75
Chapter 5: Using Arrays and Structures 97
Chapter 6: Extending ColdFusion Pages with CFML Scripting 123
Chapter 7: Using Regular Expressions in Functions 141

CHAPTER 2
Elements of CFML

This chapter provides an overview of the basic elements of CFML, including tags, functions, constants, variables, expressions, and CFScript. The chapters in Part I of this manual describe these topics in detail.

Contents

CFML Basics 28
Comments 28
Tags 29
Functions 31
ColdFusion components 32
Constants 32
Variables 33
Expressions 34
Data types 35
Flow control 35
Character case 39
Special characters 39
Reserved words 39
CFScript 40

CFML Basics

This chapter introduces and describes the basic elements of CFML. These elements make CFML a powerful tool for developing interactive web applications. Because CFML is a dynamic application development tool, it has many of the features of a programming language, including the following:

- Functions
- Expressions
- Variables and constants
- Flow-control constructs such as if-then and loops

CFML also has a "language within a language," CFScript, which enables you to use a syntax similar to JavaScript for many operations.

This chapter introduces these elements and other basic CFML entities such as comments, data types, escape characters, and reserved words.

The remainder of Part I of this manual provides more detailed information on many of the basic CFML elements. The rest of this manual helps you use these elements effectively in your applications.

Comments

ColdFusion comments have a similar format to HTML comments. However, they use three dash characters instead of two; for example:

```
<!--- This is a ColdFusion Comment. Browsers do not receive it. --->
```

The ColdFusion server removes all ColdFusion comments from the page before returning it to the web server. As a result, the page that a user browser receives does not include the comment, and users cannot see it even if they view the page source.

You can embed CFML comments in begin tags (not just tag bodies), functions calls, and variable text in number signs. ColdFusion ignores the text in comments such as the following:

```
<cfset MyVar = var1 <!--- & var2 --->>
<cfoutput>#Dateformat(now() <!---, "dddd, mmmm yyyy" --->)#</cfoutput>
```

This technique can be useful if you want to temporarily comment out parts of expressions or optional attributes or arguments.

You can also nest comments, as the following example shows:

```
<!--- disable this code
<!--- display error message --->
<cfset errormessage1="Oops!">
<cfoutput>
#errormessage1#
</cfoutput>
--->
```

This is useful if you want to temporarily disable a section of code while you test your application.

You can embed comments within comments, however, you should use this technique carefully.

Note: You cannot embed comments inside a tag name or function name, such as `<cf_My<!--- New --->CustomTag>`. You also cannot embed comments inside strings, as in the following example: `IsDefined("My<!--- New --->Variable")`.

Tags

ColdFusion ***tags*** tell the ColdFusion server that it must process information. The ColdFusion server only processes tag contents; it returns text outside of ColdFusion to the web server unchanged. Macromedia ColdFusion MX provides a wide variety of built-in tags and lets you create custom tags.

Tag syntax

ColdFusion tags have the same format as HTML tags. They are enclosed in angle brackets (< and >) and can have zero or more named attributes. Many ColdFusion tags have bodies; that is, they have beginning and end tags with text to be processed between them. For example:

```
<cfoutput>
  Hello #YourName#! <br>
</cfoutput>
```

Other tags, such as `cfset` and `cfftp`, never have bodies; all the required information goes between the beginning (<) character and the ending (>) character, as in the following example:

```
<cfset YourName="Bob">
```

Sometimes, although the tag can have a body, you do not need to put anything in it because the attributes specify all the required information. You can omit the end tag and put a forward slash character before the closing (>) character, as in the following example:

```
<cfexecute name="C:\winNT\System32\netstat.exe" arguments = "-e"
  outputfile="C:\Temp\out.txt" timeout = "1" />
```

Note: The `cfset` tag differs from other tags in that it has neither a body nor arguments. Instead, the tag encloses an assignment statement that assigns a value to a variable. The `cfset` tag can also call a function without assigning a value to a result variable.

Built-in tags

Over 110 built-in tags make up the heart of ColdFusion. These tags have many uses, including the following:

- Manipulating variables
- Creating interactive forms
- Accessing and manipulating databases
- Displaying data
- Controlling the flow of execution on the ColdFusion page
- Handling errors
- Processing ColdFusion pages
- Managing the CFML application framework
- Manipulating files and directories

- Using external tools and objects, including Verity collections, COM, Java, and CORBA objects, and executable programs
- Using protocols, such as mail, http, ftp, and pop

Much of this document describes how to use these tags effectively. *CFML Reference* documents each tag in detail.

Custom tags

ColdFusion lets you create custom tags. You can create two types of custom tags:

- CFML custom tags that are ColdFusion pages
- CFX tags that you write in a programing language such as Java or C++

Custom tags can encapsulate frequently used business logic or display code. These tags enable you to place frequently used code in one place and call it from many places. Custom tags also let you abstract complex logic into a single, simple interface. They provide an easy way to distribute your code to others; you can even distribute encrypted versions of the tags to prevent access to the tag logic.

You can access a variety of free and commercial custom tags on the Macromedia developer's exchange (www.macromedia.com/cfusion/exchange/index.cfm). They perform tasks ranging from checking if Cookies and JavaScript are enabled on the client's browser to moving items from one list box to another. Many of these tags are free and include source code.

CFML custom tags

When you write a custom tag in CFML, you can take advantage of all the features of the ColdFusion language, including all built-in tags and even other custom tags. CFML custom tags can include body sections and end tags. Because they are written in CFML, you do not need to know a programming language such as Java. CFML custom tags provide more capabilities than user-defined functions, but are less efficient.

For more information on CFML custom tags, see Chapter 11, "Creating and Using Custom CFML Tags," on page 241. For information about, and comparisons among, ways to reuse ColdFusion code, including CFML custom tags, user-defined functions, and CFX tags, see Chapter 8, "Creating ColdFusion Elements," on page 161.

CFX Tags

CFX tags are ColdFusion custom tags that you write in a programming language such as Java or C++. These tags can take full advantage of all the tools and resources provided by these languages, including their access to runtime environments. CFX tags also generally execute faster than CFML custom tags because they are compiled. CFX tags can be cross-platform, but are often platform-specific, for example if they take advantage of COM objects or the Windows API.

For more information on CFX tags, see Chapter 12, "Building Custom CFXAPI Tags," on page 259.

Functions

Functions typically manipulate data and return a result. CFML includes over 280 built-in functions. You can also create user-defined functions (UDFs), sometimes referred to as custom functions.

Functions have the following general form:

```
functionName([argument1[, argument2]]...)
```

Some functions, such as the `Now` function take no arguments. Other functions require one or more comma-separated arguments and can have additional optional arguments. All ColdFusion functions return a value. For example, `Round(3.14159)` returns the value 3.

Built-in functions

ColdFusion built-in functions perform a variety of tasks, including, but not limited to, the following:

- Creating and manipulating complex data variables, such as arrays, lists, and structures
- Creating and manipulating queries
- Creating, analyzing, manipulating, and formatting strings and date and time values
- Evaluating the values of dynamic data
- Determining the type of a variable value
- Converting data between formats
- Performing mathematical operations
- Getting system information and resources

For alphabetical and categorized lists of ColdFusion functions, see Chapter 3, "ColdFusion Functions" in *CFML Reference*.

You use built-in functions throughout ColdFusion pages. Built-in functions are frequently used in a `cfset` or `cfoutput` tag to prepare data for display or further use. For example, the following line displays today's date in the format October 12, 2001:

```
<cfoutput>#DateFormat(Now(), "mmmm d, yyyy")#</cfoutput>
```

Note that this code uses two *nested* functions. The `Now` function returns a ColdFusion date-time value representing the current date and time. The `DateFormat` function takes the value returned by the `Now` function and converts it to the desired string representation.

Functions are also valuable in CFScript scripts. ColdFusion does not support ColdFusion tags in CFScript, so you must use functions to access ColdFusion functionality in scripts.

User-defined functions

You can write your own functions, *user-defined functions* (UDFs). You can use these functions in ColdFusion expressions or in CFScript. You can call a user-defined function anywhere you can use a built-in CFML function. You create UDFs using the `cffunction` tag or the CFScript `function` statement. UDFs that you create using the `cffunction` tag can include ColdFusion tags and functions. UDFs that you create in CFScript can only include functions. You can create stand-alone UDFs or encapsulate them in a ColdFusion component.

User-defined functions let you encapsulate logic and operations that you use frequently in a single unit. This way, you can write the code once and use it multiple times. UDFs ensure consistency of coding and enable you to structure your CFML more efficiently.

Typical user-defined functions include mathematical routines, such as a function to calculate the logarithm of a number; string manipulation routines, such as a function to convert a numeric monetary value to a string such as "two dollars and three cents"; and can even include encryption and decryption routines.

Note: The Common Function Library Project at www.cflib.org includes a number of free libraries of user-defined functions.

For more information on user-defined functions, see Chapter 9, "Writing and Calling User-Defined Functions," on page 171.

ColdFusion components

ColdFusion components encapsulate multiple, related, functions. A ColdFusion component is essentially a set of related user-defined functions and variables, with additional functionality to provide and control access to the component contents. ColdFusion components can make their data private, so that it is available to all functions (also called methods) in the component, but not to any application that uses the component.

ColdFusion components have the following features:

- They are designed to provide related services in a single unit.
- They can provide web services and make them available over the Internet.
- They can provide ColdFusion services that Flash clients can call directly.
- They have several features that are familiar to object-oriented programmers, including data hiding, inheritance, packages, and introspection.

For more information on ColdFusion components, see Chapter 10, "Building and Using ColdFusion Components," on page 201.

Constants

The value of a *constant* does not change during program execution. Constants are simple scalar values that you can use within expressions and functions, such as "Robert Trent Jones" and 123.45. Constants can be integers, real numbers, time and date values, Boolean values, or text strings. ColdFusion does not allow you to give names to constants.

Variables

Variables are the most frequently used operands in ColdFusion expressions. Variable values can be set and reset, and can be passed as attributes to CFML tags. Variables can be passed as parameters to functions, and can replace most constants.

ColdFusion has a number of built-in variables that provide information about the server and are returned by ColdFusion tags. For a list of the ColdFusion built-in variables, see Chapter 1, "Reserved Words and Variables" in *CFML Reference*.

The following two characteristics classify a variable:

- The *scope* of the variable, which indicates where the information is available and how long the variable persists
- The *data type* of the variable's value, which indicates the kind of information a variable represents, such as number, string, or date

The following section lists and briefly describes the variable scopes. "Data types" on page 35 lists data types (which also apply to constant values). For detailed information on ColdFusion variables, including data types, scopes, and their use, see Chapter 3, "Using ColdFusion Variables," on page 43.

Variable scopes

The following table describes ColdFusion variable scopes:

Scope	Description
Variables (local)	The default scope for variables of any type that are created with the `cfset` and `cfparam` tags. A local variable is available only on the page on which it is created and any included pages.
Form	The variables passed from a form page to its action page as the result of submitting the form.
URL	The parameters passed to the current page in the URL that is used to call it.
Attributes	The values passed by a calling page to a custom tag in the custom tag's attributes. Used only in custom tag pages.
Caller	A reference, available in a custom tag, to the Variables scope of the page that calls the tag. Used only in custom tag pages.
ThisTag	Variables that are specific to a custom tag, including built-in variables that provide information about the tag. Used only in custom tag pages. A nested custom tag can use the `cfassociate` tag to return values to the calling tag's ThisTag scope.
Request	Variables that are available to all pages, including custom tags and nested custom tags, that are processed in response to an HTTP request. Used to hold data that must be available for the duration of one HTTP request.
CGI	Environment variables identifying the context in which a page was requested. The variables available depend on the browser and server software.
Cookie	Variables maintained in a user's browser as cookies.

Scope	Description
Client	Variables that are associated with one client. Client variables let you maintain state as a user moves from page to page in an application and are available across browser sessions.
Session	Variables that are associated with one client and persist only as long as the client maintains a session.
Application	Variables that are associated with one, named, application on a server. The Application.cfc initialization code or the `cfapplication` tag `name` attribute specifies the application name.
Server	Variables that are associated with the current ColdFusion server. This scope lets you define variables that are available to all your ColdFusion pages, across multiple applications.
Flash	Variables sent by a Macromedia Flash movie to ColdFusion and returned by ColdFusion to the movie.
Arguments	Variables passed in a call to a user-defined function or ColdFusion component method.
This	Variables that are declared inside a ColdFusion component or in a `cffunction` tag that is not part of a ColdFusion component.
function local	Variables that are declared in a user-defined function and exist only while the function executes.

Expressions

ColdFusion *expressions* consist of *operands* and *operators*. Operands are comprised of constants and variables, such as "Hello" or MyVariable. Operators, such as the string concatenation operator (&) or the division operator (/) are the verbs that act on the operands. ColdFusion functions also act as operators.

The simplest expression consists of a single operand with no operators. Complex expressions consist of multiple operands and operators. For example, the following statements are all ColdFusion expressions:

```
12
MyVariable
(1 + 1)/2
"father" & "Mother"
Form.divisor/Form.dividend
Round(3.14159)
```

For detailed information on using variables, see Chapter 3, “Using ColdFusion Variables,” on page 43. For detailed information on expressions and operators, see Chapter 4, “Using Expressions and Number Signs,” on page 75.

Data types

ColdFusion is considered *typeless* because you do not explicitly specify variable *data types*.

However, ColdFusion data, the constants and the data that variables represent, *do* have data types, which correspond to the ways the data is stored on the computer.

ColdFusion data belongs to the following type categories:

Category	Description and types
Simple	Represents one value. You can use simple data types directly in ColdFusion expressions. ColdFusion simple data types are: • strings A sequence of alphanumeric characters enclosed in single or double quotation marks, such as "This is a test." • integers A sequence of numbers written without quotation marks, such as 356. • real numbers, such as -3.14159 • Boolean values Use True, Yes, or 1 for true and False, No, or 0 for false. Boolean values are not case-sensitive. • date-time values ColdFusion supports a variety of data formats. For more information, see "Date-Time values" on page 49.
Complex	A container for data. Complex variables generally represent more than one value. ColdFusion built-in complex data types are: • arrays • structures • queries
Binary	Raw data, such as the contents of a GIF file or an executable program file
Object	COM, CORBA, Java, web services, and ColdFusion Component objects: Complex objects that you create and access using the `cfobject` tag and other specialized tags.

For more information on ColdFusion data types, see Chapter 3, "Using ColdFusion Variables," on page 43.

Flow control

ColdFusion provides several tags that let you control how a page gets executed. These tags generally correspond to programming language flow control statements, such as if, then, and else. The following tags provide ColdFusion flow control:

Tags	Purpose
`cfif`, `cfelseif`, `cfelse`	Select sections of code based on whether expressions are True or False.
`cfswitch`, `cfcase`, `cfdefaultcase`	Select among sections of code based on the value of an expression. Case processing is not limited to True and False conditions.
`cfloop`, `cfbreak`	Loop through code based on any of the following values: entries in a list, keys in a structure or external object, entries in a query column, an index, or the value of a conditional expression.
`cfabort`, `cfexit`	End processing of a ColdFusion page or custom tag.

This section provides a basic introduction to using flow-control tags. CFScript also provides a set of flow-control statements. For information on using flow-control statements in CFScript, see Chapter 6, "Extending ColdFusion Pages with CFML Scripting," on page 123. For more details on using flow-control tags, see the reference pages for these tags in *CFML Reference*.

cfif, cfelseif, and cfelse

The `cfif`, `cfelseif`, and `cfelse` tags provide if-then-else conditional processing, as follows:

1. The `cfif` tag tests a condition and executes its body if the condition is True.
2. If the preceding `cfif` (or `cfelseif`) test condition is False, the `cfelseif` tag tests another condition and executes its body if that condition is True.
3. The `cfelse` tag can optionally follow a `cfif` tag and zero or more `cfelseif` tags. Its body executes if all the preceding tags' test conditions are False.

The following example shows the use of the `cfif`, `cfelseif`, and `cfelse` tags. If the value of the type variable is "Date," the date displays; if the value is "Time," the time displays; otherwise, both the time and date display.

```
<cfif type IS "Date">
  <cfoutput>#DateFormat(Now())#</cfoutput>
<cfelseif type IS "Time">
  <cfoutput>#TimeFormat(Now())#</cfoutput>
<cfelse>
  <cfoutput>#TimeFormat(Now())#, #DateFormat(Now())#</cfoutput>
</cfif>
```

cfswitch, cfcase, and cfdefaultcase

The `cfswitch`, `cfcase`, and `cfdefaultcase` tags let you select among different code blocks based on the value of an expression. ColdFusion processes these tags as follows:

1. The `cfswitch` tag evaluates an expression. The `cfswitch` tag body contains one or more `cfcase` tags and optionally includes `cfdefaultcase` tag.
2. Each `cfcase` tag in the `cfswitch` tag body specifies a value or set of values. If a value matches the value determined by the expression in the `cfswitch` tag, ColdFusion runs the code in the body of the `cfcase` tag and then exits the `cfswitch` tag. If two `cfcase` tags have the same condition, ColdFusion generates an error.
3. If none of the `cfcase` tags match the value determined by the `cfswitch` tag, and the `cfswitch` tag body includes a `cfdefaultcase` tag, ColdFusion runs the code in the `cfdefaultcase` tag body.

Note: Although the `cfdefaultcase` tag does not have to follow all `cfcase` tags, it is good programming practice to put it at the end of the `cfswitch` statement.

The `cfswitch` tag provides better performance than a `cfif` tag with multiple `cfelseif` tags, and is easier to read. Switch processing is commonly used when different actions are required based on a a string variable such as a month or request identifier.

The following example shows switch processing:

```
<cfoutput query = "GetEmployees">
<cfswitch expression = #Department#>
  <cfcase value = "Sales">
    #FirstName# #LastName# is in <b>Sales</b><br><br>
  </cfcase>
  <cfcase value = "Accounting">
    #FirstName# #LastName# is in <b>Accounting</b><br><br>
  </cfcase>
  <cfcase value = "Administration">
    #FirstName# #LastName# is in <b>Administration</b><br><br>
  </cfcase>
  <cfdefaultcase>#FirstName# #LastName# is not in Sales,
    Accounting, or Administration.<br>
  </cfdefaultcase>
</cfswitch>
</cfoutput>
```

cfloop and cfbreak

The `cfloop` tag loops through the tag body zero or more times based on a condition specified by the tag attributes. The `cfbreak` tag exits a `cfloop` tag.

cfloop

The `cfloop` tag provides five types of loops:

Loop type	Description
Index	Loops through the body of the tag and increments a counter variable by a specified amount after each loop until the counter reaches a specified value.
Conditional	Checks a condition and runs the body of the tag if the condition is True.
Query	Loops through the body of the tag once for each row in a query.
List	Loops through the body of the tag once for each entry in a list.
Collection	Loops through the body of the tag once for each key in a ColdFusion structure or item in a COM/DCOM object.

The following example shows a simple index loop:

```
<cfloop index = "LoopCount" from = 1 to = 5>
The loop index is <cfoutput>#LoopCount#</cfoutput>.<br>
</cfloop>
```

The following example shows a simple conditional loop. The code does the following:

1. Sets up a ten-element array with the word “kumquats” in the fourth entry.
2. Loops through the array until it encounters an array element containing “kumquats” or it reaches the end of the array.
3. Prints out the value of the Boolean variable that indicates whether it found the word *kumquats* and the array index at which it exited the loop.

```
<cfset myArray = ArrayNew(1)>
<!--- Use ArraySet to initialize the first ten elements to 123 --->
<cfset ArraySet(myArray, 1, 10, 123)>
<cfset myArray[4] = "kumquats">

<cfset foundit = False>
<cfset i = 0>
<cfloop condition = "(NOT foundit) AND (i LT ArrayLen(myArray))">
  <cfset i = i + 1>
  <cfif myArray[i] IS "kumquats">
    <cfset foundit = True>
  </cfif>
</cfloop>
<cfoutput>
i is #i#<br>
foundit is #foundit#<br>
</cfoutput>
```

Note: You can get an infinite conditional loop if you do not force an end condition. In this example, the loop is infinite if you omit the `<cfset i = i + 1>` statement. To end an infinite loop, stop the ColdFusion application server.

cfbreak

The `cfbreak` tag exits the `cfloop` tag. You typically use it in a `cfif` tag to exit the loop if a particular condition occurs. The following example shows the use of a `cfbreak` tag in a query loop:

```
<cfloop query="fruitOrder">
  <cfif fruit IS "kumquat">
    <cfoutput>You cannot order kumquats!<br></cfoutput>
    <cfbreak>
  </cfif>
  <cfoutput>You have ordered #quantity# #fruit#.<br></cfoutput>
</cfloop>
```

cfabort and cfexit

The `cfabort` tag stops processing of the current page at the location of the `cfabort` tag. ColdFusion returns to the user or calling tag everything that was processed before the `cfabort` tag. You can optionally specify an error message to display. You can use the `cfabort` tag as the body of a `cfif` tag to stop processing a page when a condition, typically an error, occurs.

The `cfexit` tag controls the processing of a custom tag, and can only be used in ColdFusion custom tags. For more information see, "Terminating tag execution" on page 253 and *CFML Reference*.

Character case

ColdFusion is case-insensitive. For example, the following all represent the `cfset` tag: `cfset`, `CFSET`, `CFSet`, and even `cfsEt`. However, you should get in the habit of consistently using the same case rules in your programs; for example:

- Develop consistent rules for case use, and stick to them. If you use lowercase characters for some tag names, use them for all tag names.
- Always use the same case for a variable. For example, do not use both myvariable and MyVariable to represent the same variable on a page.

Follow these rules to prevent errors on application pages where you use both CFML and case-sensitive languages, such as JavaScript.

Special characters

The double-quotation marks ("), single-quotation mark ('), and number sign (#) characters have special meaning to ColdFusion. To include any of them in a string, double the character; for example, use ## to represent a single # character.

The need to escape the single- and double-quotation marks is context-sensitive. Inside a double-quoted string, you do not need to escape single-quotation mark (apostrophe) characters. Inside a single-quoted string, you do not escape double-quotation mark characters.

The following example illustrates escaping special characters, including the use of mixed single- and double-quotation marks:

```
<cfset mystring = "We all said ""For He's a jolly good fellow.""">
<cfset mystring2 = 'Then we said "For She''s a jolly good fellow".'>
<cfoutput>
  #mystring#<br>
  #mystring2#<br>
  Here is a number sign: ##
</cfoutput>
```

The output looks like this:

```
We all said "For He's a jolly good fellow."
Then we said "For She's a jolly good fellow."
Here is a number sign: #
```

Reserved words

As with any programming tool, you cannot use just any word or name for ColdFusion variables, UDFs and custom tags. You must avoid using any name that can be confused with a ColdFusion element. In some cases, if you use a word that ColdFusion uses—for example, a built-in structure name—you can overwrite the ColdFusion data.

The following list indicates words you must not use for ColdFusion variables, user-defined function names, or custom tag names. While some of these words can be used safely in some situations, you can prevent errors by avoiding them entirely. For a complete list of reserved words, see *CFML Reference*.

- Built-in function names, such as Now or Hash
- Scope names, such as Form or Session
- Any name starting with cf. However, when you call a CFML custom tag directly, you prefix the custom tag page name with cf_.
- Operators, such as NE or IS
- The names of any built-in data structures, such as Error or File
- The names of any built-in variables, such as RecordCount or CGI variable names
- CFScript language element names such as for, default, or continue

You must also not create form field names ending in any of the following, except to specify a form field validation rule using a hidden form field name. (For more information on form field validation, see Chapter 26, "Introduction to Retrieving and Formatting Data," on page 609.)

- _integer
- _float
- _range
- _date
- _time
- _eurodate

Remember that ColdFusion is not case-sensitive. For example, all of the following are reserved words: IS, Is, iS, and is.

CFScript

CFScript is a language within a language. CFScript is a scripting language that is similar to JavaScript but is simpler to use. Also, unlike JavaScript, CFScript only runs on the ColdFusion server; it does not run on the client system. A CFScript script can use all ColdFusion functions and all ColdFusion variables that are available in the script's scope.

CFScript provides a compact and efficient way to write ColdFusion logic. Typical uses of CFScript include:

- Simplifying and speeding variable setting
- Building compact flow control structures
- Encapsulating business logic in user-defined functions

The following sample script populates an array and locates the first array entry that starts with the word "key". It shows several of the elements of CFScript, including setting variables, loop structures, script code blocks, and function calls. Also, the code uses a `cfoutput` tag to display its results. Although you can use CFScript for output, the `cfoutput` tag is usually easier to use.

```
<cfscript>
strings = ArrayNew(1);
strings[1]="the";
strings[2]="key to our";
strings[4]="idea";
for( i=1 ; i LE 4 ; i = i+1 )
```

```
{
  if(Find("key",strings[i],1))
    break; }
</cfscript>
<cfoutput>Entry #i# starts with "key"</cfoutput><br>
```

You use CFScript to create user-defined functions.

For more information on CFScript, see Chapter 6, “Extending ColdFusion Pages with CFML Scripting,” on page 123. For more information on user-defined functions, see Chapter 9, “Writing and Calling User-Defined Functions,” on page 171.

CHAPTER 3
Using ColdFusion Variables

Macromedia ColdFusion variables are the most frequently used operands in ColdFusion expressions. Variable values can be set and reset, and can be passed as attributes to CFML tags. Variables can be passed as parameters to functions, and can replace most constants.

This chapter describes how to create and use ColdFusion variables. It includes the following information:

- How variables can represent different types of data
- How the data types get converted
- How variables exist in different scopes
- How the scopes are used
- How to use variables correctly

Contents

Creating variables 44
Variable characteristics 45
Data types 45
Using periods in variable references 56
Data type conversion 59
About scopes 64
Ensuring variable existence 70
Validating data 73
Passing variables to custom tags and UDFs 73

Creating variables

You create most ColdFusion variables by assigning them values. (You must use the `ArrayNew` function to create arrays.) Most commonly, you create variables by using the `cfset` tag. You can also use the `cfparam` tag, and assignment statements in CFScript. Tags that create data objects also create variables. For example, the `cfquery` tag creates a query object variable.

ColdFusion automatically creates some variables that provide information about the results of certain tags or operations. ColdFusion also automatically generates variables in certain scopes, such as Client and Server. For information on these special variables, see Chapter 1, "Reserved Words and Variables" in *CFML Reference* and the documentation of the CFML tags that create these variables.

ColdFusion generates an error when it tries to use a variable before it is created. This can happen, for example, when processing data from an incompletely filled form. To prevent such errors, test for the variable's existence before you use it. For more information on testing for variable existence, see "Ensuring variable existence" on page 70.

For more information on how to create variables, see "Creating and using variables in scopes" on page 67.

Variable naming rules

ColdFusion variable names, including form field names and custom function and ColdFusion component argument names, must conform to Java naming rules and the following guidelines:

- A variable name must begin with a letter, underscore, or Unicode currency symbol.
- The initial character can by followed by any number of letters, numbers, underscore characters, and Unicode currency symbols.
- A variable name cannot contain spaces.
- A query result is a type of variable, so it overwrites a local variable with the same name.
- ColdFusion variables are not case-sensitive. However, consistent capitalization makes the code easier to read.
- When creating a form with fields that are used in a query, match form field names with the corresponding database field names.
- Periods separate the components of structure or object names. They also separate a variable scope from the variable name. You cannot use periods in simple variable names, with the exception of variables in the Cookie and Client scopes. For more information on using periods, see "Using periods in variable references" on page 56.

The following rule applies to variable names, but does not apply to form field and argument names:

- Prefix each variable's name with its scope. Although some ColdFusion programmers do not use the Variables prefix for local variable names, you should use prefixes for all other scopes. Using scope prefixes makes variable names clearer and increases code efficiency. In many cases, you must prefix the scope. For more information, see "About scopes" on page 64.

Note: In some cases, when you use an existing variable name, you must enclose it with number signs (#) to allow ColdFusion to distinguish it from string or HTML text, and to insert its value, as opposed to its name. For more information, see Chapter 4, "Using number signs," on page 81.

Variable characteristics

You can classify a variable using the following characteristics:

- The data type of the variable value, which indicates the kind of information a variable represents, such as number, string, or date
- The scope of the variable, which indicates where the information is available and how long the variable persists

The following sections provide detailed information on Data types and scopes.

Data types

ColdFusion is often referred to as *typeless* because you do not assign types to variables and ColdFusion does not associate a type with the variable name. However, the data that a variable represents does have a type, and the data type affects how ColdFusion evaluates an expression or function argument. ColdFusion can automatically convert many data types into others when it evaluates expressions. For simple data, such as numbers and strings, the data type is unimportant until the variable is used in an expression or as a function argument.

ColdFusion variable data belongs to one of the following type categories:

- **Simple** One value. Can use directly in ColdFusion expressions. Include numbers, strings, Boolean values, and date-time values.
- **Complex** A container for data. Generally represent more than one value. ColdFusion built-in complex data types include arrays, structures, queries, and XML document objects.

  You cannot use a complex variable, such as an array, directly in a ColdFusion expression, but you can use simple data type elements of a complex variable in an expression.

  For example, with a one-dimensional array of numbers called myArray, you cannot use the expression myArray * 5. However, you could use an expression myArray[3] * 5 to multiply the third element in the array by five.
- **Binary** Raw data, such as the contents of a GIF file or an executable program file.
- **Objects** Complex constructs. Often encapsulate both data and functional operations. The following table lists the types of objects that ColdFusion can use, and identifies the chapters that describe how to use them:

Object type	See
Component Object Model (COM)	Chapter 38, "Integrating COM and CORBA Objects in CFML Applications," on page 945
Common Object Request Broker Architecture (CORBA)	Chapter 38, "Integrating COM and CORBA Objects in CFML Applications," on page 945
Java	Chapter 37, "Integrating J2EE and Java Elements in CFML Applications," on page 917

Object type	See
ColdFusion component	Chapter 10, "Building and Using ColdFusion Components," on page 201
Web service	Chapter 36, "Using Web Services," on page 885

Data type notes

Although ColdFusion variables do not have types, it is often convenient to use "variable type" as a shorthand for the type of data that the variable represents.

ColdFusion can validate the type of data contained in form fields and query parameters. For more information, see Chapter 26, "SELECT * FROM Departmt WHERE Dept_Name IN ('Marketing','Sales')," on page 630 and Chapter 20, "Using cfqueryparam," on page 473.

The `cfdump` tag displays the entire contents of a variable, including ColdFusion complex data structures. It is an excellent tool for debugging complex data and the code that handles it.

ColdFusion provides the following functions for identifying the data type of a variable:

- `IsArray`
- `IsBinary`
- `IsBoolean`
- `IsObject`
- `IsQuery`
- `IsSimpleValue`
- `IsStruct`
- `IsXmlDoc`

ColdFusion also includes the following functions for determining whether a string can be represented as or converted to another data type:

- `IsDate`
- `IsNumeric`
- `IsXML`

ColdFusion does not use a null data type. However, if ColdFusion receives a null value from an external source such as a database, a Java object, or some other mechanism, it maintains the null value until you use it as a simple value. At that time, ColdFusion converts the null to an empty string (""). Also, you can use the `JavaCast` function in a call to a Java object to convert a ColdFusion empty string to a Java null.

Numbers

ColdFusion supports integers and real numbers. You can intermix integers and real numbers in expressions; for example, 1.2 + 3 evaluates to 4.2.

Integers

ColdFusion supports integers between -2,147,483,648 and 2,147,483,647 (32-bit signed integers). You can assign a value outside this range to a variable, but ColdFusion initially stores the number as a string. If you use it in an arithmetic expression, ColdFusion converts it into a floating point value, preserving its value, but losing precision as the following example shows:

```
<cfset mybignum=12345678901234567890>
<cfset mybignumtimes10=(mybignum * 10)>
<cfoutput>mybignum is: #mybignum#</cfoutput><br>
<cfoutput>mybignumtimes10 is: #mybignumtimes10# </cfoutput><br>
```

This example generates the following output:

mybignum is: 12345678901234567890
mybignumtimes10 is: 1.23456789012E+020

Real numbers

Real numbers, numbers with a decimal part, are also known as floating point numbers. ColdFusion real numbers can range from approximately -10^{300} to approximately 10^{300}. A real number can have up to 12 significant digits. As with integers, you can assign a variable a value with more digits, but the data is stored as a string. The string is converted to a real number, and can lose precision, when you use it in an arithmetic expression.

You can represent real numbers in scientific notation. This format is *x*E*y*, where *x* is a positive or negative real number in the range 1.0 (inclusive) to 10 (exclusive), and *y* is an integer. The value of a number in scientific notation is x times 10^{y}. For example, 4.0E2 is 4.0 times 10^{2}, which equals 400. Similarly, 2.5E-2 is 2.5 times 10^{-2}, which equals 0.025. Scientific notation is useful for writing very large and very small numbers.

Strings

In ColdFusion, text values are stored in *strings*. You specify strings by enclosing them in either single- or double-quotation marks. For example, the following two strings are equivalent:

"This is a string"
'This is a string'

You can write an empty string in the following ways:

- "" (a pair of double-quotation marks with nothing in between)
- '' (a pair of single-quotation marks with nothing in between)

Strings can be any length, limited by the amount of available memory on the ColdFusion server. However, the default size limit for long text retrieval (CLOB) is 64K. The ColdFusion MX Administrator lets you increase the limit for database string transfers, but doing so can reduce server performance. To change the limit, select the Enable retrieval of long text option on the Advanced Settings page for the data source.

Escaping quotation marks and number signs

To include a single-quotation character in a string that is single-quoted, use two single-quotation marks (known as escaping the single-quotation mark). The following example uses escaped single-quotation marks:

```
<cfset myString='This is a single-quotation mark: '' This is a double-quotation
  mark: "'>
<cfoutput>#mystring#</cfoutput><br>
```

To include a double-quotation mark in a double-quoted string, use two double-quotation marks (known as escaping the double-quotation mark). The following example uses escaped double-quotation marks:

```
<cfset myString="This is a single-quotation mark: ' This is a double-quotation
  mark: """>
<cfoutput>#mystring#</cfoutput><br>
```

Because strings can be in either double-quotation marks or single-quotation marks, both of the preceding examples display the same text:

This is a single-quotation mark: ' This is a double-quotation mark: "

To insert a number sign (#) in a string, you must escape the number sign, as follows:

```
"This is a number sign ##"
```

Lists

ColdFusion includes functions that operate on lists, but it does not have a list data type. In ColdFusion, a *list* is just a string that consists of multiple entries separated by delimiter characters.

The default delimiter for lists is the comma. If you use any other character to separate list elements, you must specify the delimiter in the list function. You can also specify multiple delimiter characters. For example, you can tell ColdFusion to interpret a comma or a semicolon as a delimiter, as the following example shows:

```
<cfset MyList="1,2;3,4;5">
<cfoutput>
List length using ; and , as delimiters: #listlen(Mylist, ";,")#<br>
List length using only , as a delimiter: #listlen(Mylist)#<br>
</cfoutput>
```

This example displays the following output:

List length using ; and , as delimiters: 5
List length using only , as a delimiter: 3

Each delimiter must be a single character. For example, you cannot tell ColdFusion to require two hyphens in a row as a delimiter.

If a list has two delimiters in a row, ColdFusion ignores the empty element. For example, if MyList is "1,2,,3,,4,,,5" and the delimiter is the comma, the list has five elements and list functions treat it the same as "1,2,3,4,5".

Boolean values

A *Boolean* value represents whether something is true or false. ColdFusion has two special constants—True and False—to represent these values. For example, the Boolean expression 1 IS 1 evaluates to True. The expression `"Monkey" CONTAINS "Money"` evaluates to False.

You can use Boolean constants directly in expressions, as in the following example:

```
<cfset UserHasBeenHere = True>
```

In Boolean expressions, True, nonzero numbers, and the string “Yes” are equivalent, and False, 0, and the string “No” are equivalent.

In Boolean expressions, True, nonzero numbers, and the strings “Yes”, “1|”, “True” are equivalent; and False, 0, and the strings “No”, “0”, and “False” are equivalent.

Boolean evaluation is not case-sensitive. For example, True, TRUE, and true are equivalent.

Date-Time values

ColdFusion can perform operations on date and time values. Date-time values identify a date and time in the range 100 AD to 9999 AD. Although you can specify just a date or a time, ColdFusion uses one data type representation, called a date-time object, for date, time, and date and time values.

ColdFusion provides many functions to create and manipulate date-time values and to return all or part of the value in several different formats.

You can enter date and time values directly in a `cfset` tag with a constant, as follows:

```
<cfset myDate = "October 30, 2001">
```

When you do this, ColdFusion stores the information as a string. If you use a date-time function, ColdFusion stores the value as a date-time object, which is a separate simple data type. When possible, use date-time functions such as `CreateDate` and `CreateTime` to specify dates and times, because these functions can prevent you from specifying the date or time in an invalid format and they create a date-time object immediately.

Date and time formats

You can directly enter a date, time, or date and time, using standard U.S. date formats. ColdFusion processes the two-digit-year values 0 to 29 as twenty-first century dates; it processes the two-digit-year values 30 to 99 as twentieth century dates. Time values can include units down to seconds. The following table lists valid date and time formats:

To specify	Use these formats
Date	October 30, 2003 Oct 30, 2003 Oct. 30, 2003 10/30/03 2003-10-30 10-30-2003
Time	02:34:12 2:34a 2:34am 02:34am 2am
Date and Time	Any combination of valid date and time formats, such as these: October 30, 2003 02:34:12 Oct 30, 2003 2:34a Oct. 30, 2001 2:34am 10/30/03 02:34am 2003-10-30 2am 10-30-2003 2am

Locale-specific dates and times

ColdFusion provides several functions that let you input and output dates and times (and numbers and currency values) in formats that are specific to the current locale. A *locale* identifies a language and locality, such as English (US) or French (Swiss). Use these functions to input or output dates and times in formats other than the U.S. standard formats. (Use the `SetLocale` function to specify the locale.) The following example shows how to do this:

```
<cfset oldlocale = SetLocale("French (Standard)")>
<cfoutput>#LSDateFormat(Now(), "ddd, dd mmmm, yyyy")#</cfoutput>
```

This example outputs a line like the following:

mar., 03 juin, 2003

For more information on international functions, see Chapter 17, "Developing Globalized Applications," on page 403 and *CFML Reference*.

How ColdFusion stores dates and times

ColdFusion stores and manipulates dates and times as *date-time objects*. Date-time objects store data on a time line as real numbers. This storage method increases processing efficiency and directly mimics the method used by many popular database systems. In date-time objects, one day is equal to the difference between two successive integers. The time portion of the date-and-time value is stored in the fractional part of the real number. The value 0 represents 12:00 AM 12/30/1899.

Although you can use arithmetic operations to manipulate date-and-time values directly, this method can result in code that is difficult to understand and maintain. Use the ColdFusion date-time manipulation functions instead. For information on these functions, see the *CFML Reference*.

Binary data type and binary encoding

Binary data (also referred to as a *binary object*) is raw data, such as the contents of a GIF file or an executable program file. You do not normally use binary data directly, but you can use the `cffile` tag to read a binary file into a variable, typically for conversion to a string binary encoding before transmitting the file using e-mail.

A string binary encoding represents a binary value in a string format that can be transmitted over the web. ColdFusion MX supports three binary encoding formats:

Encoding	Format
Base64	Encodes the binary data in the lowest six bits of each byte. It ensures that binary data and non-ANSI character data can be transmitted using e-mail without corruption. The Base64 algorithm is specified by IETF RFC 2045, at www.ietf.org/rfc/rfc2045.txt.
Hex	Uses two characters in the range 0-9 and A-F represent the hexadecimal value of each byte; for example, 3A.
UU	Uses the UNIX UUencode algorithm to convert the data.

ColdFusion provides the following functions that convert among string data, binary data, and string encoded binary data:

Function	Description
`BinaryDecode`	Converts a string that contains encoded binary data to a binary object.
`BinaryEncode`	Converts binary data to an encoded string.
`CharsetDecode`	Converts a string to binary data in a specified character encoding.
`CharsetEncode`	Converts a binary object to a string in a specified character encoding.
`ToBase64`	Converts string and binary data to Base64 encoded data Macromedia recommends using the `BinaryEncode` function, instead of this function, to convert binary data to Base64 in new applications.

Function	Description
ToBinary	Converts Base64 encoded data to binary data. Macromedia recommends using the BinaryDecode function, instead of this function, in new applications.
ToString	Converts most simple data types to string data. It can convert numbers, date-time objects, and boolean values. (It converts date-time objects to ODBC timestamp strings.) Macromedia recommends that you use the CharsetEncode function to convert binary data to a string in new applications.

Complex data types

Arrays, structures, and queries are ColdFusion built-in complex data types. Structures and queries are sometimes referred to as objects, because they are containers for data, not individual data values.

For details on using arrays and structures, see Chapter 5, "Using Arrays and Structures," on page 97.

Arrays

Arrays are a way of storing multiple values in a table-like format that can have one or more dimensions. To create an array and specify its initial dimensions, use the ColdFusion ArrayNew function. For example, the following line creates an empty two-dimensional array:

```
<cfset myarray=ArrayNew(2)>
```

You reference elements using numeric indexes, with one index for each dimension. For example, the following line sets one element of a two-dimensional array to the current date and time:

```
<cfset myarray[1][2]=Now()>
```

The ArrayNew function can create arrays with up to three dimensions. However, there is no limit on array size or maximum dimension. To create arrays with more than three dimensions, create arrays of arrays.

After you create an array, you can use functions or direct references to manipulate its contents.

When you assign an existing array to a new variable, ColdFusion creates a new array and copies the old array's contents to the new array. The following example creates a copy of the original array:

```
<cfset newArray=myArray>
```

For more information on using arrays, see Chapter 5, "Using Arrays and Structures," on page 97.

Structures

ColdFusion *structures* consist of key-value pairs, where the keys are text strings and the values can be any ColdFusion data type, including other structures. Structures let you build a collection of related variables that are grouped under a single name. To create a structure, use the ColdFusion StructNew function. For example, the following line creates a new, empty, structure called *depts*:

```
<cfset depts=StructNew()>
```

You can also create a structure by assigning a value in the structure. For example, the following line creates a new structure called *MyStruct* with a key named *MyValue*, equal to 2:

```
<cfset MyStruct.MyValue=2>
```

Note: In previous ColdFusion versions, this line created a Variables scope variable named "MyStruct.MyValue" with the value 2.

After you create a structure, you can use functions or direct references to manipulate its contents, including adding key-value pairs.

You can use either of the following methods to reference elements stored in a structure:

- *StructureName.KeyName*
- *StructureName*["*KeyName*"]

The following examples show these methods:

```
depts.John="Sales"
depts["John"]="Sales"
```

When you assign an existing structure to a new variable, ColdFusion does *not* create a new structure. Instead, the new variable accesses the same data (location) in memory as the original structure variable. In other words, both variables are references to the same object.

For example, the following line creates a new variable, myStructure2, that is a reference to the same structure as the myStructure variable:

```
<cfset myStructure2=myStructure>
```

When you change the contents of myStructure2, you also change the contents of myStructure. To copy the contents of a structure, use the ColdFusion `Duplicate` function, which copies the contents of structures and other complex data types.

Structure key names can be the names of complex data objects, including structures or arrays. This lets you create arbitrarily complex structures.

For more information on using structures, see Chapter 5, “Using Arrays and Structures,” on page 97.

Queries

A *query object*, sometimes referred to as a query, query result, or record set, is a complex ColdFusion data type that represents data in a set of named columns, similar to the columns of a database table. The following ColdFusion tags can create query objects:

- `cfquery`
- `cfdirectory`
- `cfhttp`
- `cfldap`
- `cfpop`
- `cfprocresult`

In these tags, the `name` attribute specifies the query object’s variable name. The `QueryNew` function also creates query objects.

When you assign a query to a new variable, ColdFusion does *not* copy the query object. Instead, both names point to the same record set data. For example, the following line creates a new variable, myQuery2, that references the same record set as the myQuery variable:

```
<cfset myQuery2 = myQuery>
```

If you make changes to data in myQuery, myQuery2 also shows those changes.

You reference query columns by specifying the query name, a period, and the column name; for example:

```
myQuery.Dept_ID
```

When you reference query columns inside tags, such as `cfoutput` and `cfloop`, in which you specify the query name in a tag attribute, you do not have to specify the query name.

You can access query columns as if they are one-dimensional arrays. For example, the following line assigns the contents of the Employee column in the second row of the myQuery query to the variable myVar:

```
<cfset myVar = myQuery.Employee[2]>
```

Note: You cannot use array notation to refer to a row (of all columns) of a query. For example, myQuery[2] does not refer to the second row of the myQuery query object.

Working with structures and queries

Because structure variables and query variables are references to objects, the rules in the following sections apply to both types of data.

Multiple references to an object

When multiple variables refer to a structure or query object, the object continues to exist as long as at least one reference to the object exists. The following example shows how this works:

```
<cfscript> depts = structnew();</cfscript>
<cfset newStructure=depts>
<cfset depts.John="Sales">
<cfset depts=0>
<cfoutput>
  #newStructure.John#<br>
  #depts#
</cfoutput>
```

This example displays the following output:

Sales
0

After the `<cfset depts=0>` tag executes, the depts variable does not refer to a structure; it is a simple variable with the value 0. However, the variable newStructure still refers to the original structure object.

Assigning objects to scopes

You can give a query or structure a different scope by assigning it to a new variable in the other scope. For example, the following line creates a server variable, Server.SScopeQuery, using the local myquery variable:

```
<cfset Server.SScopeQuery = myquery>
```

To clear the server scope query variable, reassign the query object, as follows:

```
<cfset Server.SScopeQuery = 0>
```

This deletes the reference to the object from the server scope, but does not remove any other references that might exist.

Copying and duplicating objects

You can use the `Duplicate` function to make a true copy of a structure or query object. Changes to the copy do not affect the original.

Using a query column

When you are not inside a `cfloop`, `cfoutput`, or `cfmail` tag that has a `query` attribute, you can treat a query column as an array. However, query column references do not always behave as you might expect. This section explains the behavior of references to query columns using the results of the following `cfquery` tag in its examples:

```
<cfquery dataSource="cfdocexamples" name="myQuery">
  SELECT FirstName, LastName
  FROM Employee
</cfquery>
```

To reference elements in a query column, use the row number as an array index. For example, both of the following lines display the word "ben":

```
<cfoutput> #myQuery.Firstname[1]# </cfoutput><br>
<cfoutput> #myQuery["Firstname"][1]# </cfoutput><br>
```

ColdFusion behavior is less straightforward, however, when you use the query column references myQuery.Firstname and myQuery["Firstname"] without using an array index. The two reference formats produce different results.

If you refer to myQuery.Firstname, ColdFusion automatically converts it to the first row in the column. For example, the following lines print the word "ben":

```
<cfset myCol = myQuery.Firstname >
<cfoutput>#mycol#</cfoutput>
```

But the following lines display an error message:

```
<cfset myCol = myQuery.Firstname >
<cfoutput>#mycol[1]#</cfoutput><br>
```

If you refer to `Query["Firstname"]`, ColdFusion does not automatically convert it to the first row of the column. For example, the following line results in an error message indicating that ColdFusion cannot convert a complex type to a simple value:

```
<cfoutput> #myQuery['Firstname']# </cfoutput><br>
```

Similarly, the following lines print the name "marjorie", the value of the second row in the column:

```
<cfset myCol = myQuery["Firstname"]>
<cfoutput>#mycol[2]#</cfoutput><br>
```

However, when you make an assignment that requires a simple value, ColdFusion automatically converts the query column to the value of the first row. For example, the following lines display the name "ben" twice:

```
<cfoutput> #myQuery.Firstname# </cfoutput><br>
<cfset myVar= myQuery['Firstname']>
<cfoutput> #myVar# </cfoutput><br>
```

Using periods in variable references

ColdFusion uses the period (.) to separate elements of a complex variable such as a structure, query, XML document object, or external object, as in MyStruct.KeyName. A period also separates a variable scope identifier from the variable name, as in Variables.myVariable or CGI.HTTP_COOKIE.

With the exception of Cookie and Client scope variables, which must always be simple variable types, you cannot normally include periods in simple variable names. However, ColdFusion makes some exceptions that accommodate legacy and third-party code that does not conform to this requirement.

For more information, see "About scopes" on page 64, Chapter 5, "Using Arrays and Structures," on page 97, and Chapter 35, "Using XML and WDDX," on page 843.

Understanding variables and periods

The following descriptions use a sample variable named MyVar.a.b to explain how ColdFusion uses periods when getting and setting the variable value.

Getting a variable

ColdFusion can correctly get variable values even if the variable name includes a period. For example, the following set of steps shows how ColdFusion gets MyVar.a.b, as in `<cfset Var2 = myVar.a.b>` or `IsDefined(myVar.a.b)`:

1. Looks for myVar in an internal table of names (the symbol table).
2. If myVar is the name of a complex object, including a scope, looks for an element named a in the object.

   If myVar is not the name of a complex object, checks whether myVar.a is the name of a complex object and skips step 3.
3. If myVar is the name of a complex object, checks whether a is a complex object.

4. If a or myVar.a is the name of a complex object, checks whether b is the name of a simple variable, and returns the value of b.

   If myVar is a complex object but a is not a complex object, checks whether a.b is the name of a simple variable and returns its value.

   If myVar.a is not a complex object, checks whether myVar.a.b is the name of a simple variable and returns its value.

This way, ColdFusion correctly resolves the variable name and can get its value.

You can also use array notation to get a simple variable with a name that includes periods. In this form of array notation, you use the scope name (or the complex variable that contains the simple variable) as the "array" name. You put the simple variable name, in single- or double-quotation marks, inside the square brackets.

Using array notation is more efficient than using plain dot notation because ColdFusion does not have to analyze and look up all the possible key combinations. For example, both of the following lines write the value of myVar.a.b, but the second line is more efficient than the first:

```
<cfoutput>myVar.a.b is: #myVar.a.b#<br></cfoutput>
<cfoutput>myVar.a.b is: #Variables["myVar.a.b"]#<br></cfoutput>
```

Setting a variable

ColdFusion cannot be as flexible when it sets a variable value as when it gets a variable, because it must determine the type of variable to create or set. Therefore, the rules for variable names that you set are stricter. Also, the rules vary depending on whether the first part of the variable name is the Cookie or Client scope identifier.

For example, assume you have the following code:

```
<cfset myVar.a.b = "This is a test">
```

If a variable myVar does not exist, it does the following:

1. Creates a structure named myVar.
2. Creates a structure named a in the structure myVar.
3. Creates a key named b in myVar.a.
4. Gives it the value "This is a test".

If either myVar or myVar.a exist and neither one is a structure, ColdFusion generates an error.

In other words, ColdFusion uses the same rules as for getting a variable to resolve the variable name until it finds a name that does not exist yet. It then creates any structures that are needed to create a key named b inside a structure, and assigns the value to the key.

However, if the name before the first period is either Cookie or Client, ColdFusion uses a different rule. It treats all the text (including any periods) that follow the scope name as the name of a simple variable, because Cookie and Client scope variables must be simple. If you have the following code, you see that ColdFusion creates a single, simple Client scope variable named myVar.a.b:

```
<cfset Client.myVar.a.b = "This is a test">
<cfdump var=#Client.myVar.a.b#>
```

Creating variables with periods

You should avoid creating the names of variables (except for dot notation in structures) that include periods. However, ColdFusion provides mechanisms for handling cases where you must do so, for example, to maintain compatibility with names of variables in external data sources or to integrate your application with existing code that uses periods in variable names. The following sections describe how to create simple variable names that include periods.

Using brackets to create variables with periods

You can create a variable name that includes periods by using associative array structure notation, as described in Chapter 5, "Structure notation," on page 109. To do so, you must do the following:

- Refer to the variable as part of a structure. You can always do this, because ColdFusion considers all scopes to be structures. For more information on scopes, see "About scopes" on page 64.
- Put the variable name that must include a period inside square brackets and single- or double-quotation marks.

The following example shows this technique:

```
<cfset Variables['My.Variable.With.Periods'] = 12>
<cfset Request["Another.Variable.With.Periods"] = "Test variable">
<cfoutput>
  My.Variable.With.Periods is: #My.Variable.With.Periods#<br>
  Request.Another.Variable.With.Periods is:
    #Request.Another.Variable.With.Periods#<br>
</cfoutput>
```

Creating Client and Cookie variables with periods

To create a Client or Cookie variable with a name that includes one or more periods, simply assign the variable a value. For example, the following line creates a Cookie named User.Preferences.CreditCard:

```
<cfset Cookie.User.Preferences.CreditCard="Discover">
```

Data type conversion

ColdFusion automatically converts between data types to satisfy the requirements of an expression's operations, including a function's argument requirements. As a result, you generally don't need to be concerned about compatibility between data types and the conversions from one data type to another. However, understanding how ColdFusion evaluates data values and converts data between types can help you prevent errors and create code more effectively.

Operation-driven evaluation

Conventional programming languages enforce strict rules about mixing objects of different types in expressions. For example, in a language such as C++ or Basic, the expression `("8" * 10)` produces an error because the multiplication operator requires two numerical operands and "8" is a string. When you program in such languages, you must convert between data types to ensure error-free program execution. For example, the previous expression might have to be written as `(ToNumber("8") * 10)`.

In ColdFusion, however, the expression `("8" * 10)` evaluates to the number 80 without generating an error. When ColdFusion processes the multiplication operator, it automatically attempts to convert its operands to numbers. Since "8" can be successfully converted to the number 8, the expression evaluates to 80.

ColdFusion processes expressions and functions in the following sequence:

1. For each operator in an expression, it determines the required operands. (For example, the multiplication operator requires numeric operands and the CONTAINS operator requires string operands.)

   For functions, it determines the type required for each function argument. (For example, the `Min` function requires two numbers as arguments and the `Len` function requires a string.)
2. It evaluates all operands or function arguments.
3. It converts all operands or arguments whose types differ from the required type. If a conversion fails, it reports an error.

Conversion between types

Although the expression evaluation mechanism in ColdFusion is very powerful, it cannot automatically convert all data. For example, the expression `"eight" * 10` produces an error because ColdFusion cannot convert the string "eight" to the number 8. Therefore, you must understand the rules for conversion between data types.

The following table explains how conversions are performed. The first column shows values to convert. The remaining columns show the result of conversion to the listed data type.

Value	As Boolean	As number	As date-time	As string
"Yes"	True	1	Error	"Yes"
"No"	False	0	Error	"No"
True	True	1	Error	"Yes"

Value	As Boolean	As number	As date-time	As string
False	False	0	Error	"No"
Number	True if Number is not 0; False otherwise.	Number	See "Date-time values" earlier in this chapter.	String representation of the number (for example, "8").
String	If "Yes", True If "No", False If it can be converted to 0, False If it can be converted to any other number, True	If it represents a number (for example, "1,000" or "12.36E-12"), it is converted to the corresponding number. If it represents a date-time (see next column), it is converted to the numeric value of the corresponding date-time object.	If it is an ODBC date, time, or timestamp (for example "{ts '2001-06-14 11:30:13'}", or if it is expressed in a standard U.S. date or time format, including the use of full or abbreviated month names, it is converted to the corresponding date-time value. Days of the week or unusual punctuation result in an error. Dashes, forward-slashes, and spaces are generally allowed.	String
Date	Error	The numeric value of the date-time object.	Date	An ODBC timestamp.

ColdFusion cannot convert complex types, such as arrays, queries, and COM objects, to other types. However, it can convert simple data elements of complex types to other simple data types.

Type conversion considerations

The following sections detail specific rules and considerations for converting between types.

The cfoutput tag

The `cfoutput` tag always displays data as a string. As a result, when you display a variable using the `cfoutput` tag, ColdFusion applies the type conversion rules to any non-string data before displaying it. For example, the `cfoutput` tag displays a date-time value as an ODBC timestamp.

Case-insensitivity and Boolean conversion

Because ColdFusion expression evaluation is not case-sensitive, Yes, YES, and yes are equivalent; False, FALSE, and false are equivalent; No, NO, and no are equivalent; and True, TRUE, and true are equivalent.

Converting binary data

ColdFusion cannot automatically convert binary data to other data types. To convert binary data, use the `ToBase64` and `ToString` functions. For more information, see "Binary data type and binary encoding" on page 51.

Converting date and time data

To ensure that a date and time value is expressed as a real number, add zero to the variable. The following example shows this:

```
<cfset mynow = now()>
Use cfoutput to display the result of the now function:<br>
<cfoutput>#mynow#</cfoutput><br>
Now add 0 to the result and display it again:<br>
<cfset mynow = mynow + 0>
<cfoutput>#mynow#</cfoutput>
```

At 1:06 PM on June 6, 2003, its output looked like this:

```
Use cfoutput to display the result of the now function:
{ts '2003-06-03 13:06:44'}
Now add 0 to the result and display it again:
37775.5463426
```

Converting numeric values

When ColdFusion evaluates an expression that includes both integers and real numbers, the result is a real number. To convert a real number to an integer, use a ColdFusion function. The `Int`, `Round`, `Fix`, and `Ceiling` functions convert real numbers to integers, and differ in their treatment of the fractional part of the number.

If you use a hidden form field with a name that has the suffix `_integer` or `_range` to validate a form input field, ColdFusion truncates real numbers entered into the field and passes the resulting integer to the action page.

If you use a hidden form field with a name that has the suffix `_integer`, `_float`, or `_range` to validate a form input field, and the entered data contains a dollar amount (including a dollar sign) or a numeric value with commas, ColdFusion considers the input to be valid, removes the dollar sign or commas from the value, and passes the resulting integer or real number to the action page.

Evaluation and type conversion issues

The following sections explain several issues that you might encounter with type evaluation and conversion.

Comparing variables to True or False

You might expect the following two `cfif` tag examples to produce the same results:

```
<cfif myVariable>
  <cfoutput>myVariable equals #myVariable# and is True
  </cfoutput>
</cfif>
<cfif myVariable IS True>
  <cfoutput>myVariable equals #myVariable# and is True
  </cfoutput>
</cfif>
```

However, if myVariable has a numeric value such as 12, only the first example produces a result. In the second case, the value of myVariable is not converted to a Boolean data type, because the IS operator does not require a specific data type and just tests the two values for identity. Therefore, ColdFusion compares the value 12 with the constant True. The two are not equal, so nothing is printed. If myVariable is 1, "Yes", or True, however, both examples print the same result, because ColdFusion considers these to be identical to Boolean True.

If you use the following code, the output statement does display, because the value of the variable, 12, is not equal to the Boolean value False:

```
<cfif myVariable IS NOT False>
  <cfoutput>myVariable equals #myVariable# and IS NOT False
  </cfoutput>
</cfif>
```

As a result, you should use the test `<cfif testvariable>`, and not use the `IS` comparison operator when testing whether a variable is True or False. This issue is a case of the more general problem of ambiguous type expression evaluation, described in the following section.

Ambiguous type expressions and strings

When ColdFusion evaluates an expression that does not require strings, including all comparison operations, such as `IS` or `GT`, it checks whether it can convert each string value to a number or date-time object. If so, ColdFusion converts it to the corresponding number or date-time value (which is stored as a number). It then uses the number in the expression.

Short strings, such as 1a and 2P, can produce unexpected results. ColdFusion can interpret a single "a" as AM and a single "P" as PM. This can cause ColdFusion to interpret strings as date-time values in cases where this was not intended.

Similarly, if the strings can be interpreted as numbers, you might get unexpected results.

For example, ColdFusion interprets the following expressions as shown:

Expression	Interpretation
`<cfif "1a" EQ "01:00">`	If 1:00am is 1:00am.
`<cfif "1P" GT "2A">`	If 1:00pm is later than 2:00am.
`<cfset age="4a">` `<cfset age=age + 7>`	Treat the variable age as 4:00 am, convert it to the date-time value 0.16666666667, and add 7 to make it 7.16666666667.
`<cfif "0.0" is "0">`	If 0 is 0.

To prevent such ambiguities when you compare strings, use the ColdFusion string comparison functions `Compare` and `CompareNoCase`, instead of the comparison operators.

You can also use the `IsDate` function to determine whether a string can be interpreted as a date-time value, or to add characters to a string before comparison to avoid incorrect interpretation.

Date-time functions and queries when ODBC is not supported

Many CFML functions, including the `Now`, `CreateDate`, `CreateTime`, and `CreateDateTime` functions, return date-time objects. ColdFusion creates Open Database Connectivity (ODBC) timestamp values when it converts date-time objects to strings. As a result, you might get unexpected results when using dates with a database driver that does not support ODBC escape sequences, or when you use SQL in a query of queries.

If you use SQL to insert data into a database or in a WHERE clause to select data from a database, and the database driver does not support ODBC-formatted dates, use the `DateFormat` function to convert the date-time value to a valid format for the driver. This rule also applies to queries of queries.

For example, the following SQL statement uses the `DateFormat` function in a query of queries to select rows that have MyDate values in the future:

```
<cfquery name="MyQofQQ" dbtype="query">
SELECT *
FROM DateQuery
WHERE MyDate >= '#DateFormat(Now())#'
</cfquery>
```

The following query of queries fails with the error message "Error: {ts is not a valid date," because the ColdFusion `Now` function returns an ODBC timestamp:

```
<cfquery name="MyQofQQ" dbtype="query">
SELECT *
FROM DateQuery
WHERE MyDate >= '#now()#'
</cfquery>
```

Using JavaCast with overloaded Java methods

You can overload Java methods so a class can have several identically named methods that differ only in parameter data types. At runtime, the Java virtual machine attempts to resolve the specific method to use, based on the types of the parameters passed in the call. Because ColdFusion does not use explicit types, you cannot predict which version of the method the virtual machine will use.

The ColdFusion `JavaCast` function helps you ensure that the right method executes by specifying the Java type of a variable, as in the following example:

```
<cfset emp.SetJobGrade(JavaCast("int", JobGrade))>
```

The `JavaCast` function takes two parameters: a string representing the Java data type and the variable whose type you are setting. You can specify the following Java data types: bool, int, long, float, double, and String.

For more information on the `JavaCast` function, see *CFML Reference*.

Using quotation marks

To ensure that ColdFusion properly interprets string data, surround strings in single- or double-quotation marks. For example, ColdFusion evaluates “10/2/2001” as a string that can be converted into a date-time object. However, it evaluates 10/2/2001 as a mathematical expression, 5/2001, which evaluates to 0.00249875062469.

Examples of type conversion in expression evaluation

The following examples demonstrate ColdFusion expression evaluation.

Example 1

```
2 * True + "YES" - ('y' & "es")
```

Result value as string: "2"

Explanation: (2*True) is equal to 2; ("YES"- "yes") is equal to 0; 2 + 0 equals 2.

Example 2

```
"Five is " & 5
```

Result value as string: "Five is 5"

Explanation: 5 is converted to the string "5".

Example 3

```
DateFormat("October 30, 2001" + 1)
```

Result value as string: "31-Oct-01"

Explanation: The addition operator forces the string "October 30, 2001" to be converted to a date-time object, and then to a number. The number is incremented by one. The DateFormat function requires its argument to be a date-time object; thus, the result of the addition is converted to a date-time object. One is added to the date-time object, moving it ahead by one day to October 31, 2001.

About scopes

Variables differ in how they are set (by your code or by ColdFusion), the places in your code where they are meaningful, and how long their values persist. These considerations are generally referred to as a variable’s *scope*. Commonly used scopes include the Variables scope, the default scope for variables that you create, and the Request scope, which is available for the duration of an HTTP request.

Note: User-defined functions also belong to scopes. For more information, see Chapter 9, “Specifying the scope of a function,” on page 194.

Scope types

The following table describes ColdFusion scopes:

Scope	Description
Variables (local)	The default scope for variables of any type that are created with the `cfset` and `cfparam` tags. A local variable is available only on the page on which it is created and any included pages (see also the Caller scope).
Form	Contains variables passed from a Form page to its action page as the result of submitting the form. (If you use the HTML `form` tag, you must use `method="post"`.) For more information, see Chapter 26, "Introduction to Retrieving and Formatting Data," on page 609.
URL	Contains parameters passed to the current page in the URL that is used to call it. The parameters are appended to the URL in the format ?variablename = value[&variablename=value...]; for example www.MyCompany.com/inputpage.cfm?productCode=A12CD1510&quantity=3. ***Note:*** If a URL includes multiple parameters with the same name, the resulting variable in the ColdFusion URL scope consists of all parameter values separated by commas. For example, a URL of the form http://localhost/urlparamtest.cfm?param=1&param=2&param=3 results in a URL.param variable value of 1,2,3 on the ColdFusion page.
Attributes	Used only in custom tag pages. Contains the values passed by the calling page in the custom tag's attributes. For more information, see Chapter 11, "Creating and Using Custom CFML Tags," on page 241.
Caller	Used only in custom tag pages. The custom tag's Caller scope is a reference to the calling page's Variables scope. Any variables that you create or change in the custom tag page using the Caller scope are visible in the calling page's Variables scope. For more information, see Chapter 11, "Creating and Using Custom CFML Tags," on page 241.
ThisTag	Used only in custom tag pages. The ThisTag scope is active for the current invocation of the tag. If a custom tag contains a nested tag, any ThisTag scope values you set before calling the nested tag are preserved when the nested tag returns to the calling tag. The ThisTag scope includes three built-in variables that identify the tag's execution mode, contain the tag's generated contents, and indicate whether the tag has an end tag. A nested custom tag can use the `cfassociate` tag to return values to the calling tag's ThisTag scope. For more information, see "Accessing tag instance data" on page 250.
Request	Used to hold data that must be available for the duration of one HTTP request. The Request scope is available to all pages, including custom tags and nested custom tags, that are processed in response to the request. This scope is useful for nested (child/parent) tags. This scope can often be used in place of the Application scope, to avoid the need for locking variables. Several chapters discuss using the Request scope.
CGI	Contains environment variables identifying the context in which a page was requested. The variables available depend on the browser and server software. For a list of the commonly used CGI variables, see *Chapter 1, "Reserved Words and Variables,"* in *CFML Reference*.

Scope	Description
Cookie	Contains variables maintained in a user's browser as cookies. Cookies are typically stored in a file on the browser, so they are available across browser sessions and applications. You can create memory-only Cookie variables, which are not available after the user closes the browser. Cookie scope variable names can include periods.
Client	Contains variables that are associated with one client. Client variables let you maintain state as a user moves from page to page in an application, and are available across browser sessions. By default, Client variables are stored in the system registry, but you can store them in a cookie or a database. Client variables cannot be complex data types and can include periods in their names. For more information, see Chapter 15, "Using Persistent Data and Locking," on page 339.
Session	Contains variables that are associated with one client and persist only as long as the client maintains a session. They are stored in the server's memory and can be set to time out after a period of inactivity. For more information, see Chapter 15, "Using Persistent Data and Locking," on page 339.
Application	Contains variables that are associated with one, named application on a server. The `cfapplication` tag `name` attribute or the Application.cfc This.name variable setting specifies the application name. For more information, see Chapter 15, "Using Persistent Data and Locking," on page 339.
Server	Contains variables that are associated with the current ColdFusion server. This scope lets you define variables that are available to all your ColdFusion pages, across multiple applications. For more information, see Chapter 15, "Using Persistent Data and Locking," on page 339.
Flash	Variables sent by a Macromedia Flash movie to ColdFusion and returned by ColdFusion to the movie. For more information, see Chapter 33, "Using the Flash Remoting Service," on page 807.
Arguments	Variables passed in a call to a user-defined function or ColdFusion component method. For more information, see "About the Arguments scope" on page 181.
This	Exists only in ColdFusion components or `cffunction` tags that are part of a containing object such as a ColdFusion Struct. Exists for the duration of the component instance or containing object. Data in the This scope is accessible from outside the component or container by using the instance or object name as a prefix.
function local	Contains variables that are declared inside a user-defined function or ColdFusion component method and exist only while a function executes. For more information, see Chapter 9, "Writing and Calling User-Defined Functions," on page 171.

Caution: To prevent data corruption, you lock code that uses Session, Application, or Server scope variables. For more information, see Chapter 15, "Using Persistent Data and Locking," on page 339.

Creating and using variables in scopes

The following table shows how you create and refer to variables in different scopes in your code. For more information on the mechanisms for creating variables in most scopes, see "Creating variables" on page 44.

Scope prefix (type)	Prefix required to reference	Where available	Created by
Variables (Local)	No	On the current page. Cannot be accessed by a form's action page (unless the form page is also the action page). Variables in this scope used on a page that calls a custom tag can be accessed in the custom tag by using its Caller scope; however, they are not available to any nested custom tags.	Specifying the prefix Variables, or using no prefix, when you create the variable.
Form	No	On the action page of a form and in custom tags called by the action page; cannot be used on a form page that is not also the action page.	A `form` or `cfform` tag. Contains the values of form field tags (such as input) in the form body when the form is submitted. The variable name is the name of the form field.
URL	No	On the target page of the URL.	The system. Contains the parameters passed in the URL query string used to access the page.
Attributes	Yes	On the custom tag page.	The calling page passing the values to a custom tag page in the custom tag's attributes.
Caller	On the custom tag page, Yes. On the calling page, No (Variables prefix is optional).	On the custom tag page, by using the Caller scope prefix. On the page that calls the custom tag, as local variables (Variables scope).	On the custom tag page, by specifying the prefix Caller when you create the variable. On the calling page, by specifying the prefix Variables, or using no prefix, when you create the variable.
ThisTag	Yes	On the custom tag page.	Specifying the prefix ThisTag when you create the variable in the tag or using the `cfassociate` tag in a nested custom tag.
Request	Yes	On the creating page and in any pages invoked during the current HTTP request after the variable is created, including in custom tags and nested custom tags.	Specifying the prefix Request when you create the variable.

Scope prefix (type)	Prefix required to reference	Where available	Created by
CGI	No	On any page. Values are specific to the latest browser request.	The web server. Contains the server environment variables that result from the browser request.
Cffile	Yes	Following an invocation of `cffile`.	A `cffile` tag.
Cookie	No	For one client in one or more applications and pages, over multiple browser sessions.	A `cfcookie` tag. You can also set memory-only cookies by specifying the prefix Cookie when you create the variable.
Client	No	For one client in one application, over multiple browser sessions.	Specifying the prefix Client when you create the variable.
Session	Yes	For one client in one application and one browser session. Surround code that uses Session scope variables in `cflock` blocks.	Specifying the prefix Session when you create the variable.
Application	Yes	For multiple clients in one application over multiple browser sessions. Surround code that uses application variables in `cflock` blocks.	Specifying the prefix Application when you create the variable.
Server	Yes	To any page on the ColdFusion server. Surround all code that uses server variables in `cflock` blocks.	Specifying the prefix Server when you create the variable.
Flash	Yes	A ColdFusion page or ColdFusion component called by a Flash client.	The ColdFusion Client access. You assign a value to Flash.You can assign values to the Flash.result and Flash.pagesize variables.
Arguments	No	Within the body of a user-defined function or ColdFusion component method.	The calling page passing an argument in the function call.

Scope prefix (type)	Prefix required to reference	Where available	Created by
This	Yes	Within a ColdFusion component or the body of a user-defined function that was created using the `cffunction` tag and put in an object, structure, or scope. In the containing page, through the component instance or containing object.	Within the component or function by specifying the prefix This when you create the variable. In the containing page, by specifying the component instance or object that contains the function as a prefix when you create the variable.
(function local, no prefix)	Prohibited	Within the body of a user-defined function or ColdFusion component method, only while the function executes.	In the function or method definition, a `var` keyword in a `cfset` tag or a CFScript `var` statement.

Using scopes

The following sections provide details on how you can create and use variables in different scopes.

Evaluating unscoped variables

If you use a variable name without a scope prefix, ColdFusion checks the scopes in the following order to find the variable:

1. Function local (UDFs and CFCs only)
2. Arguments
3. Variables (local scope)
4. CGI
5. Cffile
6. URL
7. Form
8. Cookie
9. Client

Because ColdFusion must search for variables when you do not specify the scope, you can improve performance by specifying the scope for all variables.

To access variables in all other scopes, you must prefix the variable name with the scope identifier.

Scopes and CFX tags

ColdFusion scopes do not apply to ColdFusion Extension (CFX) tags, custom tags that you write in a programming language such as C++ or Java. The ColdFusion page that calls a CFX tag must use tag attributes to pass data to the CFX tag. The CFX tag must use the Java Request and Response interfaces or the C++ Request class to get and return data.

The Java `setVariable` Response interface method and C++ `CCFX::SetVariable` method return data to the Variables scope of the calling page. Therefore, they are equivalent to setting a Caller scope variable in a custom ColdFusion tag.

Using scopes as structures

ColdFusion makes all named scopes available as structures. You cannot access the function-local scope for user defined functions (UDFs) that you define using CFScript as a structure. (In ColdFusion 4.5 and 5, the following scopes are *not* available as structures: Variables, Caller, Client, and Server.)

You can reference the variables in named scopes as elements of a structure. To do so, specify the scope name as the structure name and the variable name as the key. For example, if you have a MyVar variable in the Request scope, you can refer to it in either of the following ways:

```
Request.MyVar
Request["MyVar"]
```

Similarly, you can use CFML structure functions to manipulate the contents of the scope. For more information on using structures, see Chapter 5, "Using Arrays and Structures," on page 97.

Caution: Do not call `StructClear(Session)` to clear session variables. This deletes the `SessionID`, `CFID`, and `CFtoken` built-in variables, effectively ending the session. If you want to use `StructClear` to delete your application variables, put those variables in a structure in the Session scope, and then clear that structure. For example, put all your application variables in Session.MyVars and then call `StructClear(Session.MyVars)` to clear the variables.

Ensuring variable existence

ColdFusion generates an error if you try to use a variable value that does not exist. Therefore, before you use any variable whose value is assigned dynamically, you must ensure that a variable value exists. For example, if your application has a form, it must use some combination of requiring users to submit data in fields, providing default values for fields, and checking for the existence of field variable values before they are used.

There are several ways to ensure that a variable exists before you use it, including the following:

- You can use the `IsDefined` function to test for the variable's existence.
- You can use the `cfparam` tag to test for a variable and set it to a default value if it does not exist.
- You can use a `cfform input` tag with a `hidden` attribute to tell ColdFusion to display a helpful message to any user who does not enter data in a required field. For more information on this technique, see Chapter 26, "Requiring users to enter values in form fields," on page 616.

Testing for a variable's existence

Before relying on a variable's existence in an application page, you can test to see if it exists by using the `IsDefined` function. To check whether a specific key exists in a structure, use the `StructKeyExists` function.

For example, if you submit a form with an unsettled check box, the action page does not get a variable for the check box. The following example from a form action page makes sure the Contractor check box Form variable exists before using it:

```
<cfif IsDefined("Form.Contractor")>
  <cfoutput>Contractor: #Form.Contractor#</cfoutput>
  </cfif>
```

You must always enclose the argument passed to the `IsDefined` function in double-quotation marks. For more information on the `IsDefined` function, see *CFML Reference*.

If you attempt to evaluate a variable that you did not define, ColdFusion cannot process the page and displays an error message. To help diagnose such problems, turn on debugging in the ColdFusion MX Administrator or use the debugger in your editor. The Administrator debugging information shows which variables are being passed to your application pages.

Variable existence considerations

If a variable is part of a scope that is available as a structure, you might get a minor performance increase by testing the variable's existence using the `StructKeyExists` function instead of the `IsDefined` function.

You can also determine which Form variables exist by inspecting the contents of the `Form.fieldnames` built-in variable. This variable contains a list of all the fields submitted by the form. Remember, however, that form text fields are always submitted to the action page, and might contain an empty string if the user did not enter data.

The `IsDefined` function always returns False if you specify an array or structure element using bracket notation. For example, `IsDefined("myArray[3]")` always returns False, even if the array element `myArray[3]` has a value. To check for the existence of an array element, use `cftry`, as in the following example:

```
<cfset items=ArrayNew(2)>
<cfset items[1][1] = "Dog">
<cfset items[2][2] = "Cat">
<cftry>
  <cfset temp=items[1][2]>
<cfcatch type="any">
<  cfoutput>Items[1][2] does not exist</cfoutput>
</cfcatch>
</cftry>
```

Using the cfparam tag

You can ensure that a variable exists by using the `cfparam` tag, which tests for the variable's existence and optionally supplies a default value if the variable does not exist. The `cfparam` tag has the following syntax:

```
<cfparam name="VariableName"
  type="data_type"
  default="DefaultValue">
```

Note: For information on using the `type` attribute to validate the parameter data type, see *CFML Reference*.

There are two ways to use the cfparam tag to test for variable existence, depending on how you want the validation test to proceed:

- With only the name attribute to test that a required variable exists. If it does not exist, the ColdFusion server stops processing the page and displays an error message.
- With the name and default attributes to test for the existence of an optional variable. If the variable exists, processing continues and the value is not changed. If the variable does not exist, it is created and set to the value of the default attribute, and processing continues.

The following example shows how to use the cfparam tag to check for the existence of an optional variable and to set a default value if the variable does not already exist:

```
<cfparam name="Form.Contract" default="Yes">
```

Example: testing for variables

Using the cfparam tag with the name attribute is one way to clearly define the variables that a page or a custom tag expects to receive before processing can proceed. This can make your code more readable, as well as easier to maintain and debug.

For example, the following cfparam tags indicate that this page expects two form variables named StartRow and RowsToFetch:

```
<cfparam name="Form.StartRow">
  <cfparam name="Form.RowsToFetch">
```

If the page with these tags is called without either one of the form variables, an error occurs and the page stops processing. By default, ColdFusion displays an error message; you can also handle the error as described in Chapter 14, “Handling Errors,” on page 307.

Example: setting default values

The following example uses the cfparam tag to see if optional variables exist. If they do exist, processing continues. If they do not exist, the ColdFusion server creates them and sets them to the default values.

```
<cfparam name="Cookie.SearchString" default="temple">
<cfparam name="Client.Color" default="Grey">
<cfparam name="ShowExtraInfo" default="No">
```

You can use the cfparam tag to set default values for URL and Form variables, instead of using conditional logic. For example, you could include the following code on the action page to ensure that a SelectedDepts variable exists:

```
<cfparam name="Form.SelectedDepts" default="Marketing,Sales">
```

Validating data

It is often not sufficient that input data merely exists; it must also have the right format. For example, a date field must have data in a date format. A salary field must have data in a numeric or currency format. There are many ways to ensure the validity of data, including the following methods:

- Use the `cfparam` tag with the `type` attribute to validate a variable.
- Use the `IsValid` function to validate a variable.
- Use the `cfqueryparam` tag in a SQL WHERE clause to validate query parameters.
- Use `cfform` controls that have validation attributes.
- Use a form `input` tag with a `hidden` attribute to validate the contents of a form input field.

Note: Data validation using the `cfparam`, `cfqueryparam`, and form tags is done by the server. Validation using `cfform` tags and hidden fields is done using JavaScript in the user's browser, before any data is sent to the server.

For detailed information on validating data in forms and variables, see Chapter 28, "Validating Data," on page 659. For detailed information on validating query parameters, see "Using cfqueryparam" on page 473.

Passing variables to custom tags and UDFs

The following sections describe rules for how data gets passed to custom tags and user-defined functions that are written in CFML, and to CFX custom tags that are written in Java or C++.

Passing variables to CFML tags and UDFs

When you pass a variable to a CFML custom tag as an attribute, or to a user-defined function as an argument, the following rules determine whether the custom tag or function receives its own private copy of the variable or only gets a reference to the calling page's variable:

- Simple variables and arrays are passed as copies of the data. If your argument is an expression that contains multiple simple variables, the result of the expression evaluation is copied to the function or tag.
- Structures, queries, and `cfobject` objects are passed as references to the object.

If the tag or function gets a copy of the calling page's data, changes to the variable in the custom tag or function do not change the value of the variable on the calling page. If the variable is passed by reference, changes to the variable in the custom tag or function also change the value of the variable in the calling page.

To pass a variable to a custom tag, you must enclose the variable name in number signs. To pass a variable to a function, do *not* enclose the variable name in number signs. For example, the following code calls a user-defined function using three Form variables:

```
<cfoutput>
TOTAL INTEREST: #TotalInterest(Form.Principal, Form.AnnualPercent,
  Form.Months)#<br>
</cfoutput>
```

The following example calls a custom tag using two variables, MyString and MyArray:

```
<cf_testTag stringval=#MyString# arrayval=#MyArray#>
```

Passing variables to CFX tags

You cannot pass arrays, structures, or `cfobject` objects to CFX tags. You can pass a query to a CFX tag by using the `query` attribute when calling the tag. ColdFusion normally converts simple data types to strings when passing them to CFX tags; however, the Java Request Interface `getIntAttribute` method lets you get a passed integer value.

CHAPTER 4
Using Expressions and Number Signs

This chapter discusses how to use expressions in CFML. It discusses the elements of Macromedia ColdFusion expressions and how to create expressions. It also describes the correct use of number signs to indicate expressions in ColdFusion tags such as `cfoutput`, in strings, and in expressions. Finally, it describes how to use variables in variable names and strings to create dynamic expressions, and dynamic variables.

Contents

Expressions 75

Using number signs 81

Dynamic expressions and dynamic variables. 84

Expressions

ColdFusion expressions consist of *operands* and *operators*. Operands are comprised of constants and variables. Operators, such as the multiplication symbol, are the verbs that act on the operands; functions are a form of operator.

The simplest expression consists of a single operand with no operators. Complex expressions have multiple operators and operands. The following are all ColdFusion expressions:

```
12
MyVariable
(1 + 1)/2
"father" & "Mother"
Form.divisor/Form.dividend
Round(3.14159)
```

Operators act on the operands. Some operators, such as functions with a single argument, take a single operand. Many operators, including most arithmetic and logical operators, take two operands. The following is the general form of a two-operand expression:

```
Expression Operator Expression
```

Note that the operator is surrounded by expressions. Each expression can be a simple operand (variable or constant) or a *subexpression* consisting of more operators and expressions. Complex expressions are built up using subexpressions. For example, in the expression (1 + 1)/2, 1 + 1 is a subexpression consisting of an operator and two operands.

Operator types

ColdFusion has four types of operators:

- Arithmetic
- Boolean
- Decision (or comparison)
- String

Functions also can be viewed as operators because they act on operands.

Arithmetic operators

The following table describes the arithmetic operators:

Operator	Description
+ - * /	Basic arithmetic: Addition, subtraction, multiplication, and division. In division, the right operand cannot be zero.
+ -	Unary arithmetic: Set the sign of a number.
MOD	Modulus: Return the remainder after a number is divided by a divisor. The result has the same sign as the divisor. The right should be an integer; using a non-numeric value causes an error, and if you specify a real number, ColdFusion ignores the fractional part (for example, 11 MOD 4 is 3).
\	Integer division: Divide an integer by another integer. Use the backslash character (\) to separate the integers. The right operand cannot be zero. For example, 9\4 is 2.
^	Exponentiation: Return the result of a number raised to a power (exponent). Use the caret character (^) to separate the number from the power; for example, 2^3 is 8. Real and negative numbers are allowed for both the base and the exponent. However, any expression that equates to an imaginary number, such -1^.5 results in the string "-1.#IND. ColdFusion does not support imaginary or complex numbers.

Boolean operators

Boolean, or logical, operators perform logical connective and negation operations. The operands of Boolean operators are Boolean (True/False) values. The following table describes the Boolean operators:

Operator	Description
NOT	Reverse the value of an argument. For example, NOT True is False and vice versa.
AND	Return True if both arguments are True; return False otherwise. For example, True AND True is True, but True AND False is False.

Operator	Description
OR	Return True if any of the arguments is True; return False otherwise. For example, True OR False is True, but False OR False is False.
XOR	Exclusive or: Return True if one of the values is True and the other is False. Return False if both arguments are True or both are False. For example, True XOR True is False, but True XOR False is True.
EQV	Equivalence: Return True if both operands are True or both are False. The EQV operator is the opposite of the XOR operator. For example, True EQV True is True, but True EQV False is False.
IMP	Implication: The statement A IMP B is the equivalent of the logical statement "If A Then B." A IMP B is False only if A is True and B is False. It is True in all other cases.

Decision operators

The ColdFusion decision, or comparison, operators produce a Boolean True/False result. The following table describes the decision operators:

Operator	Description
IS	Perform a case-insensitive comparison of two values. Return True if the values are identical.
IS NOT	Opposite of IS. Perform a case-insensitive comparison of two values. Return True if the values are not identical.
CONTAINS	Return True if the value on the left contains the value on the right.
DOES NOT CONTAIN	Opposite of CONTAINS. Return True if the value on the left does not contain the value on the right.
GREATER THAN	Return True if the value on the left is greater than the value on the right.
LESS THAN	Opposite of GREATER THAN. Return True if the value on the left is smaller than the value on the right.
GREATER THAN OR EQUAL TO	Return True if the value on the left is greater than or equal to the value on the right.
LESS THAN OR EQUAL TO	Return True if the value on the left is less than or equal to the value on the right.

Alternative notation for decision operators

You can replace some decision operators with alternative notations to make your CFML more compact, as shown in the following table:

Operator	Alternative name(s)
IS	EQUAL, EQ
IS NOT	NOT EQUAL, NEQ
GREATER THAN	GT

Operator	Alternative name(s)
LESS THAN	LT
GREATER THAN OR EQUAL TO	GTE, GE
LESS THAN OR EQUAL TO	LTE, LE

Decision operator rules

The following rules apply to decision operators:

- When ColdFusion evaluates an expression that contains a decision operator other than CONTAINS or DOES NOT CONTAIN, it first determines if the data can be converted to numeric values. If they can be converted, it performs a numeric comparison on the data. If they cannot be converted, it performs a string comparison. This can sometimes result in unexpected results. For more information on this behavior, see "Evaluation and type conversion issues" on page 61.
- When ColdFusion evaluates an expression with CONTAINS or DOES NOT CONTAIN it does a string comparison. The expression A CONTAINS B evaluates to True if B is a substring of A. Therefore an expression such as the following evaluates as True:

  ```
  123.45 CONTAINS 3.4
  ```

- When a ColdFusion decision operator compares strings, it ignores the case. As a result, the following expression is True:

  ```
  "a" IS "A"
  ```

- When a ColdFusion decision operator compares strings, it evaluates the strings from left to right, comparing the characters in each position according to their sorting order. The first position where the characters differ determines the relative values of the strings. As a result, the following expressions are True:

  ```
  "ab" LT "aba"
  "abde" LT "ac"
  ```

String operators

There is one string operator, which is the concatenation operator.

Operator	Description
&	Concatenates strings.

Note: In a Query of Queries, you use || as the concatenation operator.

Operator precedence and evaluation ordering

The order of precedence controls the order in which operators in an expression are evaluated. The order of precedence is as follows:

```
Unary +, Unary -
^
*, /
\
MOD
+, -
&
EQ, NEQ, LT, LTE, GT, GTE, CONTAINS, DOES NOT CONTAIN
NOT
AND
OR
XOR
EQV
IMP
```

To enforce a non-standard order of evaluation, you must parenthesize expressions. For example:

- 6 - 3 * 2 is equal to 0
- (6 - 3) * 2 is equal to 6

You can nest parenthesized expressions. When in doubt about the order in which operators in an expression will be evaluated, use parentheses to force the order of evaluation.

Using functions as operators

Functions are a form of operator. Because ColdFusion functions return values, you can use function results as operands. Function arguments are expressions. For example, the following are valid expressions:

- `Rand()`
- `UCase("This is a text: ") & ToString(123 + 456)`

Function syntax

The following table shows function syntax and usage guidelines:

Usage	Example
No arguments	`Function()`
Basic format	`Function(Data)`
Nested functions	`Function1(Function2(Data))`
Multiple arguments	`Function(Data1, Data2, Data3)`
String arguments	`Function('This is a demo')` `Function("This is a demo")`
Arguments that are expressions	`Function1(X*Y, Function2("Text"))`

All functions return values. In the following example, the `cfset` tag sets a variable to the value returned by the `Now` function:

```
<cfset myDate = DateFormat(Now(), "mmmm d, yyyy")>
```

You can use the values returned by functions directly to create more complex expressions, as in the following example:

```
Abs(Myvar)/Round(3.14159)
```

For more information on how to insert functions in expressions, see "Using number signs" on page 81.

Optional function arguments

Some functions take optional arguments after their required arguments. If omitted, all optional arguments default to a predefined value. For example:

- `Replace("Eat and Eat", "Eat", "Drink")` returns "Drink and Eat"
- `Replace("Eat and Eat", "Eat", "Drink", "All")` returns "Drink and Drink"

The difference in the results is because the `Replace` function takes an optional fourth argument that specifies the scope of replacement. The default value is "One," which explains why only the first occurrence of "Eat" was replaced with "Drink" in the first example. In the second example, a fourth argument causes the function to replace all occurrences of "Eat" with "Drink".

Expression evaluation and functions

It is important to remember that ColdFusion evaluates function attributes as expressions before it executes the function. As a result, you can use any ColdFusion expression as a function attribute. For example, consider the following lines:

```
<cfset firstVariable = "we all need">
<cfset myStringVar = UCase(firstVariable & " more sleep!")>
```

When ColdFusion server executes the second line, it does the following:

1. Determines that there is an expression with a string concatenation.
2. Evaluates the firstVariable variable as the string "we all need".
3. Concatenates "we all need" with the string " more sleep!" to get "we all need more sleep!".
4. Passes the string "we all need more sleep!" to the `UCase` function.
5. Executes the `UCase` function on the string argument "we all need more sleep!" to get "WE ALL NEED MORE SLEEP!".
6. Assigns the string value "WE ALL NEED MORE SLEEP!" to the variable myStringVar.

ColdFusion completes steps 1-3 before invoking the function.

Using number signs

Number signs (#) have a special meaning in CFML. When the ColdFusion server encounters number signs in CFML text, such as the text in a `cfoutput` tag body, it checks to see if the text between the number signs is either a variable or a function.

Tip: Number signs are also called pound signs.

Is so, it replaces the text and surrounding number signs with the variable value or the result of the function. Otherwise, ColdFusion generates an error.

For example, to output the current value of a variable named `Form.MyFormVariable`, you delimit (surround) the variable name with number signs:

```
<cfoutput>Value is #Form.MyFormVariable#</cfoutput>
```

In this example, the variable `Form.MyFormVariable` is replaced with the value assigned to it.

Follow these guidelines when using number signs:

- Use number signs to distinguish variables or functions from plain text.
- Surround only a single variable or function in number signs; for example, #Variables.myVar# or #Left(myString, position)#. (However, a function in number signs can contain nested functions, such as #Left(trim(myString), position)#.
- Do not put complex expressions, such as `1 + 2` in number signs. Although this is allowed in a cfoutput block, such as <cfoutput>One plus one is #1 + 1#</cfoutput>, doing so mixes logic and presentation.
- Use number signs *only* where necessary, because unneeded number signs slow processing.

The following sections provide more details on how to use number signs in CFML. For a description of using number signs to create variable names, see "Using number signs to construct a variable name in assignments" on page 85.

Using number signs in ColdFusion tag attribute values

You can put variables, functions, or expressions inside tag attributes by enclosing the variable or expression with number signs. For example, if the variable CookieValue has the value "MyCookie", the following line sets the `cfcookie value` attribute to "The value is MyCookie":

```
<cfcookie name="TestCookie" value="The value is #CookieValue#">
```

You can optionally omit quotation marks around variables used as attribute values as shown in the following example:

```
<cfcookie name = TestCookie value = #CookieValue#>
```

However, surrounding all attribute values in quotation marks is more consistent with HTML coding style.

If you use string expressions to construct an attribute value, as shown in the following example, the strings inside the expression use single quotation marks (') to differentiate the quotation marks from the quotation marks that surround the attribute value.

```
<cfcookie name="TestCookie2" value="The #CookieValue & 'ate the cookie!'#">
```

Note: You do not need to use number signs when you use the `cfset` tag to assign one variable's value to another value. For example, the following tag assigns the value of the `oldVar` variable to the new variable, `newVar`: `<cfset newVar = oldVar>`.

Using number signs in tag bodies

You can put variables or functions freely inside the bodies of the following tags by enclosing each variable or expression with number signs:

- `cfoutput`
- `cfquery`
- `cfmail`

For example:

```
<cfoutput>
  Value is #Form.MyTextField#
</cfoutput>

<cfoutput>
  The name is #FirstName# #LastName#.
</cfoutput>

<cfoutput>
  The value of Cos(0) is #Cos(0)#
</cfoutput>
```

If you omit the number signs, the text, rather than the value, appears in the output generated by the `cfoutput` statement.

Two expressions inside number signs can be adjacent to one another, as in the following example:

```
<cfoutput>
  "Mo" and "nk" is #Left("Moon", 2)##Mid("Monkey", 3, 2)#
</cfoutput>
```

This code displays the following text:

"Mo" and "nk" is Monk

ColdFusion does not interpret the double number sign as an escaped # character.

Using number signs in strings

You can put variables or functions freely inside strings by enclosing each variable or expression with number signs; for example:

```
<cfset TheString = "Value is #Form.MyTextField#">
<cfset TheString = "The name is #FirstName# #LastName#.">
<cfset TheString = "Cos(0) is #Cos(0)#">
```

ColdFusion automatically replaces the text with the value of the variable or the value returned by the function. For example, the following pairs of `cfset` statements produce the same result:

```
<cfset TheString = "Hello, #FirstName#!">
<cfset TheString = "Hello, " & FirstName & "!">
```

If number signs are omitted inside the string, the text, rather than the value, appears in the string. For example, the following pairs of `cfset` statements produce the same result:

```
<cfset TheString = "Hello, FirstName!">
<cfset TheString = "Hello, " & "First" & "Name!">
```

As with the `cfoutput` statement, two expressions can be adjacent to each other in strings, as in the following example:

```
<cfset TheString = "Monk is #Left("Moon", 2)##Mid("Monkey", 3, 2)#">
```

The double-quotation marks around "Moon" and "Monkey" do *not* need to be escaped (as in ""Moon"" and ""Monkey""). This is because the text between the number signs is treated as an expression; it is evaluated before its value is inserted inside the string.

Nested number signs

In a few cases, you can nest number signs in an expression. The following example uses nested number signs:

```
<cfset Sentence = "The length of the full name is
  #Len("#FirstName# #LastName#")#">
```

In this example, number signs are nested so that the values of the variables FirstName and LastName are inserted in the string whose length the `Len` function calculates.

Nested number signs imply a complex expression that can typically be written more clearly and efficiently without the nesting. For example, you can rewrite the preceding code example without the nested number signs, as follows:

```
<cfset Sentence2 = "The length of the full name is #Len(FirstName & " "
  & LastName)#">
```

The following achieves the same results and can further improve readability:

```
<cfset FullName = "#FirstName# #LastName#">
<cfset Sentence = "The length of the full name
  is #Len(FullName)#">
```

A common mistake is to put number signs around the arguments of functions, as in:

```
<cfset ResultText = "#Len(#TheText#)#">
<cfset ResultText = "#Min(#ThisVariable#, 5 + #ThatVariable#)#">
<cfset ResultText = "#Len(#Left("Some text", 4)#)#">
```

These statements result in errors. As a general rule, *never* put number signs around function arguments.

Using number signs in expressions

Use number signs in expressions only when necessary, because unneeded number signs reduce clarity and can increase processing time. The following example shows the preferred method for referencing variables:

```
<cfset SomeVar = Var1 + Max(Var2, 10 * Var3) + Var4>
```

In contrast, the following example uses number signs unnecessarily and is less efficient than the previous statement:

```
<cfset #SomeVar# = #Var1# + #Max(Var2, 10 * Var3)# + #Var4#>
```

Dynamic expressions and dynamic variables

This section discusses the advanced topics of dynamic expressions, dynamic evaluation, and dynamic variable naming. Many ColdFusion programmers never encounter or need to use dynamic expressions. However, dynamic variable naming is important in situations where the variable names are not known in advance, such as in shopping cart applications.

This section also discusses the use of the `IIf` function, which is most often used without dynamic expressions. This function dynamically evaluates its arguments, and you must often use the `DE` function to prevent the evaluation. For more information on using the `IIF` function, see "Using the IIF function" on page 90.

Note: This section uses several tools and techniques that are documented in later chapters. If you are unfamiliar with using ColdFusion forms, structures, and arrays, you should learn about these tools before reading this section.

About dynamic variables

Dynamic variables are variables that are named dynamically, typically by creating a variable name from a static part and a variable part. For example, the following example dynamically constructs the variable name from a variable prefix and a static suffix:

```
<cfset "#flavor#_availability" = "out of stock">
```

Using dynamic variables in this manner does not require dynamic evaluation.

About dynamic expressions and dynamic evaluation

In a *dynamic expression*, the actual expression, not just its variable values, is determined at execution time. In other words, in a dynamic expression the structure of the expression, such as the names of the variables, not just the values of the variables, gets built at runtime.

You create dynamic expressions using *string expressions*, which are expressions contained in strings, (that is, surrounded with quotation marks). *Dynamic evaluation* is the process of evaluating a string expression. The `Evaluate` and `IIf` functions, and only these functions, perform dynamic evaluation.

When ColdFusion performs dynamic evaluation it does the following:

1. Takes a string expression and treats it as a standard expression, as if the expression was not a string.
2. Parses the expression to determine the elements of the expression and validate the expression syntax.
3. Evaluates the expression, looking up any variables and replacing them with their values, calling any functions, and performing any required operations.

This process enables ColdFusion to interpret dynamic expressions with variable parts. However, it incurs a substantial processing overhead.

Dynamic expressions were important in early versions of ColdFusion, before it supported arrays and structures, and they still can be useful in limited circumstances. However, the ability to use structures and the ability to use associative array notation to access structure elements provide more efficient and easier methods for dynamically managing data. For information on using arrays and structures, see Chapter 5, "Using Arrays and Structures," on page 97.

Selecting how to create variable names

The following two examples describes cases when you need dynamic variable names:

- Form applications where the number and names of fields on the form vary dynamically. In this case, the form posts only the names and values of its fields to the action page. The action page does not know all the names of the fields, although it does know how the field names (that is, the variable names) are constructed.
- If the following are true:
  - ColdFusion calls a custom tag multiple times.
  - The custom tag result must be returned to different variables each time.
  - The calling code can specify the variable in which to return the custom tag result.

  In this case, the custom tag does not know the return variable name in advance, and gets it as an attribute value.

In both cases, it might appear that dynamic expressions using the `Evaluate` function are needed to construct the variable names. However, you can achieve the same ends more efficiently by using dynamic variable naming, as shown in "Example: a dynamic shopping cart" on page 92.

This does not mean that you must always avoid dynamic evaluation. However, given the substantial performance costs of dynamic evaluation, you should first ensure that one of the following techniques cannot serve your purpose:

- An array (using index variables)
- Associative array references containing expressions to access structure elements
- Dynamically generated variable names

Dynamic variable naming without dynamic evaluation

While ColdFusion does not always allow you to construct a variable name in-line from variable pieces, it does let you to do so in the most common uses, as described in the following sections.

Using number signs to construct a variable name in assignments

You can combine text and variable names to construct a variable name on the left side of a `cfset` assignment. For example, the following code sets the value of the variable Product12 to the string "Widget":

```
<cfset ProdNo = 12>
<cfset "Product#ProdNo#" = "Widget">
```

To construct a variable name this way, all the text on the left side of the equal sign must be in quotation marks.

This usage is less efficient than using arrays. The following example has the same purpose as the previous one, but requires less processing:

```
<cfset MyArray=ArrayNew(1)>
<cfset prodNo = 12>
<cfset myArray[prodNo] = "Widget">
```

Dynamic variable limitation

When you use a dynamic variable name in quotation marks on the left side of an assignment, the name must be either a simple variable name or a complex name that uses object.property notation (such as MyStruct.#KeyName#). You cannot use an array as part of a dynamic variable name. For example, the following code generates an error:

```
<cfset MyArray=ArrayNew(1)>
<cfset productClassNo = 1>
<cfset productItemNo = 9>
<cfset "myArray[#productClassNo##productItemNo#]" = "Widget">
```

However, you can construct an array index value dynamically from variables without using quotation marks on the left side of an assignment. For example, the preceding sample code works if you replace the final line with the following line:

```
<cfset myArray[#productClassNo# & #productItemNo#] = "Widget">
```

Dynamically constructing structure references

The ability to use associative array notation to reference structures provides a way for you to use variables to dynamically create structure references. (For a description of associative array notation, see "Structure notation" on page 109.) Associative array structure notation allows you to use a ColdFusion expression inside the index brackets. For example, if you have a productName structure with keys of the form product_1, product_2 and so on, you can use the following code to display the value of productName.product_3:

```
<cfset prodNo = 3>
<cfoutput>
  Product_3 Name: #ProductName["product_" & prodNo]#
<cfoutput>
```

For an example of using this format to manage a shopping cart, see "Example: a dynamic shopping cart" on page 92.

Using dynamic evaluation

The following sections describe how to use dynamic evaluation and create dynamic expressions.

ColdFusion dynamic evaluation functions

The following table describes the functions that perform dynamic evaluation and are useful in evaluating dynamic expressions:

Function	Purpose
`DE`	Escapes any double-quotation marks in the argument and wraps the result in double-quotation marks. The `DE` function is particularly useful with the `IIF` function, to prevent the function from evaluating a string to be output. For an example of using the `DE` function with the `IIF` function, see "Using the IIF function" on page 90.
`Evaluate`	Takes one or more string expressions and dynamically evaluates their contents as expressions from left to right. (The results of an evaluation to the left can have meaning in an expression to the right.) Returns the result of evaluating the rightmost argument. For more information on this function see "About the Evaluate function" on page 88.
`IIf`	Evaluates a boolean condition expression. Depending on whether this expression is True or False, dynamically evaluates one of two string expressions and returns the result of the evaluation. The `IIF` function is convenient for incorporating a `cfif` tag in-line in HTML. For an example of using this function, see "Using the IIF function" on page 90.
`SetVariable`	Sets a variable identified by the first argument to the value specified by the second argument. This function is no longer required in well-formed ColdFusion pages; see "SetVariable function considerations" on page 90.

Function argument evaluation considerations

It is important to remember that ColdFusion always evaluates function arguments *before* the argument values are passed to a function:

For example, consider the following `DE` function:

```
<cfoutput>#DE("1" & "2")#</cfoutput>
```

You might expect this line to display """1"" & ""2""". Instead, it displays "12", because ColdFusion processes the line as follows:

1. Evaluates the expression "1" & "2" as the string "12".
2. Passes the string "12" (without the quotation marks) to the `DE` function.
3. Calls the `DE` function, which adds literal quotation marks around the 12.

Similarly, if you use the expression DE(1 + 2), ColdFusion evaluates 1 + 2 as the integer 3 and passes it to the function. The function converts it to a string and surrounds the string in literal quotation marks: "3".

About the Evaluate function

The `Evaluate` function takes one or more string expressions, dynamically evaluates their contents as expressions from left to right, and returns the result of evaluating the rightmost argument.

The following example shows the `Evaluate` function and how it works with ColdFusion variable processing:

```
<cfset myVar2="myVar">
<cfset myVar="27/9">
<cfoutput>
  #myVar2#<br>
  #myVar#<br>
  #Evaluate("myVar2")#<br>
  #Evaluate("myVar")#<br>
  #Evaluate(myVar2)#<br>
  #Evaluate(myVar)#<br>
</cfoutput>
```

Reviewing the code

The following table describes how ColdFusion processes this code:

Code	Description
`<cfset myVar2="myVar">` `<cfset myVar="27/9">`	Sets the two variables to the following strings: myVar 27/9
`<cfoutput>` `  #myVar2# ` `  #myVar# `	Displays the values assigned to the variables, myVar and 27/9, respectively.
`#Evaluate("myVar2")# `	Passes the string "myvar2" (without the quotation marks) to the Evaluate function, which does the following: 1 Evaluates it as the variable myVar2. 2 Returns the value of the myVar2 variable, the string "myvar" (without the quotation marks).
`#Evaluate("myVar")# `	Passes the string "myvar" (without the quotation marks) to the Evaluate function, which does the following: 1 Evaluates it as the variable myVar. 2 Returns the value of the myVar variable, the string "27/9" (without the quotation marks).
`#Evaluate(myVar2)# `	Evaluates the variable myVar2 as the string "myVar" and passes the string (without the quotation marks) to the Evaluate function. The rest of the processing is the same as in the previous line.
`#Evaluate(myVar)# ` `</cfoutput>`	Evaluates the variable myVar as the string "27/9" (without the quotation marks), and passes it to the Evaluate function, which does the following: 1 Evaluates the string as the expression 27/9 2 Performs the division. 3 Returns the resulting value, 3

As you can see, using dynamic expressions can result in substantial expression evaluation overhead, and the code can be confusing. Therefore, you should avoid using dynamic expressions wherever a simpler technique, such as using indexed arrays or structures can serve your purposes.

Avoiding the Evaluate function

Using the `Evaluate` function increases processing overhead, and in most cases it is not necessary. The following sections provide examples of cases where you might consider using the `Evaluate` function.

Example 1

You might be inclined to use the `Evaluate` function in code such as the following:

```
<cfoutput>1 + 1 is #Evaluate(1 + 1)#</cfoutput>
```

Although this code works, it is not as efficient as the following code:

```
<cfset Result = 1 + 1>
<cfoutput>1 + 1 is #Result#</cfoutput>
```

Example 2

This example shows how you can use an associative array reference in place of an `Evaluate` function. This technique is powerful because:

- Most ColdFusion scopes are accessible as structures.
- You can use ColdFusion expressions in the indexes of associative array structure references. (For more information on using associative array references for structures, see "Structure notation" on page 109.)

The following example uses the `Evaluate` function to construct a variable name:

```
<cfoutput>
Product Name: #Evaluate("Form.product_#i#")#
</cfoutput>
```

This code comes from an example where a form has entries for an indeterminate number of items in a shopping cart. For each item in the shopping cart there is a product name field. The field name is of the form product_1, product_2, and so on, where the number corresponds to the product's entry in the shopping cart. In this example, ColdFusion does the following:

1. Replaces the variable i with its value, for example 1.
2. concatenates the variable value with "Form.product_", and passes the result (for Form.product_1) to the `Evaluate` function, which does the remaining steps.
3. Parses the variable product_1 and generates an executable representation of the variable. Because ColdFusion must invoke its parser, this step requires substantial processing, even for a simple variable.
4. Evaluates the representation of the variable, for example as "Air popper".
5. Returns the value of the variable.

The following example has the same result as the preceding example and is more efficient:

```
<cfoutput>
ProductName: #Form["product_" & i]#
</cfoutput>
```

In this code, ColdFusion does the following:

1. Evaluates the expression in the associative array index brackets as the string "product_" concatenated with the value of the variable i.
2. Determines the value of the variable i; 1.
3. Concatenates the string and the variable value to get product_1.
4. Uses the result as the key value in the Form structure to get Form[product_1]. This associative array reference accesses the same value as the object.attribute format reference Form.product_1; in this case, Air popper.

This code format does not use any dynamic evaluation, but it achieves the same effect, of dynamically creating a structure reference by using a string and a variable.

SetVariable function considerations

You can avoid using the `SetVariable` function by using a format such as the following to set a dynamically named variable. For example, the following lines are equivalent:

```
<cfset SetVariable("myVar" & i, myVal)>
```

```
<cfset "myVar#i#" = myVal>
```

In the second line, enclosing the myVar#i# variable name in quotation marks tells ColdFusion to evaluate the name and process any text in number signs as a variable or function. ColdFusion replaces the #i# with the value of the variable i, so that if the value of i is 12, this code is equivalent to the line

```
<cfset myVar12 = myVal>
```

For more information on this usage, see "Using number signs to construct a variable name in assignments" on page 85.

Using the IIF function

The `IIf` function is a shorthand for the following code:

```
<cfif argument1>
  <cfset result = Evaluate(argument1)>
<cfelse>
  <cfset result = Evaluate(argument2)>
</cfif>
```

The function returns the value of the result variable. It is comparable to the use of the JavaScript and Java ? : operator, and can result in more compact code. As a result, the `IIF` function can be convenient even if you are not using dynamic expressions.

The IIF function requires the DE function to prevent ColdFusion from evaluating literal strings, as the following example shows:

```
<cfoutput>
#IIf(IsDefined("LocalVar"), "LocalVar", DE("The variable is not
  defined."))#
</cfoutput>
```

If you do not enclose the string "The variable is not defined." in a DE function, the IIF function tries to evaluate the contents of the string as an expression and generates an error (in this case, an invalid parser construct error).

The IIF function is useful for incorporating ColdFusion logic in-line in HTML code, but it entails a processing time penalty in cases where you do not otherwise need dynamic expression evaluation.

The following example shows using IIF to alternate table row background color between white and gray. It also shows the use of the DE function to prevent ColdFusion from evaluating the color strings.

```
<cfoutput>
<table border="1" cellpadding="3">
<cfloop index="i" from="1" to="10">
  <tr bgcolor="#IIF( i mod 2 eq 0, DE("white"), DE("gray") )#">
    <td>
      hello #i#
    </td>
  </tr>
</cfloop>
</table>
</cfoutput>
```

This code is more compact than the following example, which does not use IIF or DE:

```
<cfoutput>
<table border="1" cellpadding="3">
<cfloop index="i" from="1" to="10">
  <cfif i mod 2 EQ 0>
    <cfset Color = "white">
  <cfelse>
    <cfset Color = "gray">
  </cfif>
  <tr bgcolor="#color#">
    <td>
      hello #i#
    </td>
  </tr>
</cfloop>
</table>
</cfoutput>
```

Example: a dynamic shopping cart

The following example dynamically creates and manipulates variable names without using dynamic expression evaluation by using associative array notation.

You need to dynamically generate variable names in applications such as shopping carts, where the required output is dynamically generated and variable. In a shopping cart, you do not know in advance the number of cart entries or their contents. Also, because you are using a form, the action page only receives Form variables with the names and values of the form fields.

The following example shows the shopping cart contents and lets you edit your order and submit it. To simplify things, the example automatically generates the shopping cart contents using CFScript instead of having the user fill the cart. A more complete example would populate a shopping cart as the user selected items. Similarly, the example omits all business logic for committing and making the order.

To create the form:

1. Create a file in your editor.

```
<html>
<head>
  <title>Shopping Cart</title>
</head>
<cfscript>
CartItems=4;
Cart = ArrayNew(1);
for ( i=1; i LE cartItems; i=i+1)
{
  Cart[i]=StructNew();
  Cart[i].ID=i;
  Cart[i].Name="Product " & i;
  Cart[i].SKU=i*100+(2*i*10)+(3*i);
  Cart[i].Qty=3*i-2;
}
</cfscript>

<body>
Your shopping cart has the following items.<br>
You can change your order quantities.<br>
If you don't want any item, clear the item's check box.<br>
When you are ready to order, click submit.<br>
<br>
<cfform name="ShoppingCart" action="ShoppingCartAction.cfm" method="post">
  <table>
  <tr>
    <td>Order?</td>
    <td>Product</td>
    <td>Code</td>
    <td>Quantity</td>
  </tr>
  <cfloop index="i" from="1" to="#cartItems#">
    <tr>
      <cfset productName= "product_" & Cart[i].ID>
      <cfset skuName= "sku_" & Cart[i].ID>
```

```
        <cfset qtyname= "qty_" & Cart[i].ID>
        <td><cfinput type="checkbox" name="itemID" value="#Cart[i].ID#"
    checked>
         </td>
        <td><cfinput type="text" name="#productName#" value="#Cart[i].Name#"
         passThrough = "readonly = 'True'"></td>
        <td><cfinput type="text" name="#skuName#" value="#Cart[i].SKU#"
         passThrough = "readonly = 'True'"></td>
        <td><cfinput type="text" name="#qtyName#" value="#Cart[i].Qty#">
         </td>
      </tr>
    </cfloop>
    </table>
    <input type="submit" name="submit" value="submit">
  </cfform>

  </body>
  </html>
```

2. Save the page as ShoppingCartForm.cfm.

Reviewing the code

The following table describes the code:

Code	Description

`<cfscript>`
`CartItems=4;`
`Cart = ArrayNew(1);`
`for ( i=1; i LE #cartItems#; i=i+1)`
`{`
`  Cart[i]=StructNew();`
`  Cart[i].ID=i;`
`  Cart[i].Name="Product " & i;`
`  Cart[i].SKU=i*100+(2*i*10)+(3*i);`
`  Cart[i].Qty=3*i-2;`
`}`
`</cfscript>`
	Create a shopping cart as an array of structures, with each structure containing the cart item ID, product name, SKU number, and quantity ordered for one item in the cart. Populate the shopping cart by looping CartItems times and setting the structure variables to arbitrary values based on the loop counter. A real application would set the Name, SKU, and Quantity values on other pages.

`<cfform name="ShoppingCart"`
`  action="ShoppingCartAction.cfm"`
`  method="post">`
`<table>`
`<tr>`
`  <td>Order?</td>`
`  <td>Product</td>`
`  <td>Code</td>`
`  <td>Quantity</td>`
`</tr>`
	Start the form and its embedded table. When the user clicks the submit button, post the form data to the ShoppingCartAction.cfm page.
	The table formats the form neatly. The first table row contains the column headers. Each following row has the data for one cart item.

Code	Description
<cfloop index="i" from="1" to="#cartItems#"> <tr> <cfset productName= "product_" & Cart[i].ID> <cfset skuName= "sku_" & Cart[i].ID> <cfset qtyname= "qty_" & Cart[i].ID> <td><cfinput type="checkbox" name="itemID" value="#Cart[i].ID#" checked> </td> <td><cfinput type="text" name="#productName#" value="#Cart[i].Name#" passThrough = "readonly = 'True'"> </td> <td><cfinput type="text" name="#skuName#" value="#Cart[i].SKU#" passThrough = "readonly = 'True'"> </td> <td><cfinput type="text" name="#qtyName#" value="#Cart[i].Qty#"> </td> </tr> </cfloop> </table>	Loop through the shopping cart entries to generate the cart form dynamically. For each loop, generate variables used for the form field name attributes by appending the cart item ID (Cart[i].ID) to a field type identifier, such as "sku_". Use a single name, "itemID", for all check boxes. This way, the itemID value posted to the action page is a list of all the check box field values. The check box field value for each item is the cart item ID. Each column in a row contains a field for a cart item structure entry. The `passthrough` attribute sets the product name and SKU fields to read-only; note the use of single-quotation marks. (For more information on the `cfinput` tag `passthrough` attribute, see *CFML Reference*.) The check boxes are selected by default.
<input type="submit" name="submit" value="Submit"> </form>	Create the Submit button and end the form.

To create the Action page:

1. Create a file in your editor.
2. Enter the following text:

```
<html>
<head>
  <title>Your Order</title>
</head>
<body>
<cfif isDefined("Form.submit")>
  <cfparam name="Form.itemID" default="">
  <cfoutput>
    You have ordered the following items:<br>
    <br>
    <cfloop index="i" list="#Form.itemID#">
      ProductName: #Form["product_" & i]#<br>
      Product Code: #Form["sku_" & i]#<br>
      Quantity: #Form["qty_" & i]#<br>
      <br>
    </cfloop>
  </cfoutput>
</cfif>
</body>
</html>
```

3. Save the file as ShoppingCartAction.cfm
4. Open ShoppingCartform.cfm in your browser, change the check box and quantity values, and click Submit.

Reviewing the code

The following table describes the code:

Code	Description
`<cfif isDefined("Form.submit")>`	Run the CFML on this page only if it is called by submitting a form. This is not needed if there are separate form and action pages, but is required if the form and action page were one ColdFusion page.
`<cfparam name="Form.itemID" default="">`	Set the default Form.itemID to the empty string. This prevents ColdFusion from displaying an error if the user clears all check boxes before submitting the form (so no product IDs are submitted).
`<cfoutput>` `You have ordered the following items: ` ` `<cfloop index="i" list="#Form.itemID#">` `ProductName: #Form["product_" & i]# ` `Product Code: #Form["sku_" & i]# ` `Quantity: #Form["qty_" & i]# ` ` `</cfloop>` `</cfoutput>` `</cfif>`	Display the name, SKU number, and quantity for each ordered item. The form page posts Form.itemID as a list containing the value attributes of all the check boxes. These attributes contain the shopping cart item IDs for the selected cart items. Use the list values to index a loop that outputs each ordered item. Use associative array notation to access the Form scope as a structure and use expressions in the array indexes to construct the form variable names. The expressions consist of a string containing the field name's field type prefix (for example, "sku_"), concatenated with the variable i, which contains the shopping cart ItemID number (which is also the loop index variable).

CHAPTER 5
Using Arrays and Structures

Macromedia ColdFusion supports dynamic multidimensional arrays. This chapter explains the basics of creating and handling arrays. It also provides several examples showing how arrays can enhance your ColdFusion application code.

ColdFusion also supports structures for managing lists of key-value pairs. Because structures can contain other structures or complex data types as it values, they provide a flexible and powerful tool for managing complex data. This chapter explains the basics of creating and working with structures.

Contents

About arrays 97

Basic array techniques 100

Populating arrays with data 105

Array functions 108

About structures 108

Creating and using structures 111

Structure examples 117

Structure functions 121

About arrays

Traditionally, an *array* is a tabular structure used to hold data, much like a spreadsheet table with clearly defined limits and dimensions.

In ColdFusion, you typically use arrays to temporarily store data. For example, if your site lets users order goods online, you can store their shopping cart contents in an array. This lets you make changes easily without committing the information, which the user can change before completing the transaction, to a database.

Basic array concepts

The following terms will help you understand subsequent discussions of ColdFusion arrays:

- **Array dimension** The relative complexity of the array structure.
- **Index** The position of an element in a dimension, ordinarily surrounded by square brackets: my1Darray[1], my2Darray[1][1], my3Darray[1][1][1].
- **Array element** Data stored at an array index.

The simplest array is a one-dimensional array, similar to a row in a table. A one-dimensional array has a *name* (the variable name) and a numerical index. The index number references a single entry, or cell, in the array, as the following figure shows:



Thus, the following statement sets the value of the fifth entry in the one-dimensional array MyArray to “Robert”:

```
<cfset MyArray[5] = "Robert">
```

A basic two-dimensional (2D) array is like a simple table. A three-dimensional (3D) array is like a cube of data, and so on. ColdFusion lets you directly create arrays with up to three dimensions. You can use multiple statements to create arrays with more than three dimensions.

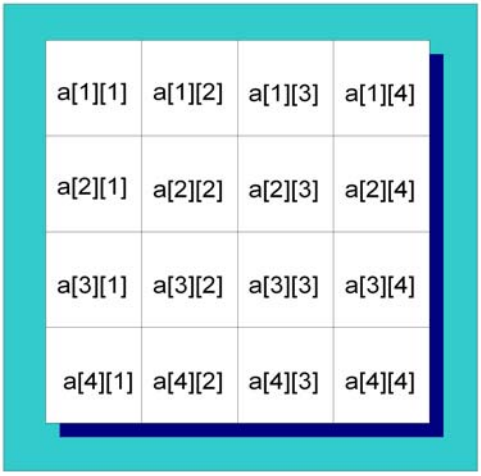
The syntax `my2darray[1][3]="Paul"` is the same as saying 'My2dArray is a two-dimensional array and the value of the array element index [1][3] is "Paul"'.

About ColdFusion arrays

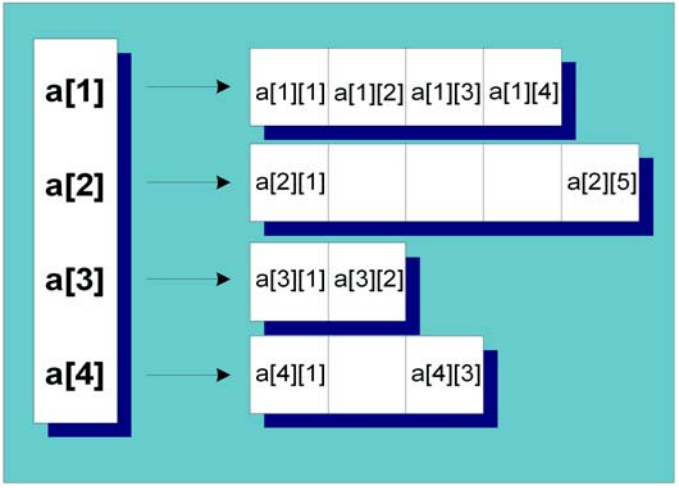
ColdFusion arrays differ from traditional arrays, because they are dynamic. For example, in a conventional array, array size is constant and symmetrical, whereas in a ColdFusion array, you can have rows of differing lengths based on the data that has been added or removed.

The following figures show the differences between traditional arrays and ColdFusion arrays using 2D arrays. The differences between traditional and ColdFusion 3D arrays are similar, but much harder to show on a page.

A conventional 2D array is like a fixed-size table made up of individual cells, as the following figure shows:



The following figure represents a ColdFusion 2D array:



A ColdFusion 2D array is actually a one-dimensional array that contains a series of additional 1D arrays. Each of the arrays that make up a row can expand and contract independently of any other column. Similarly, a ColdFusion 3D array is essentially three nested sets of 1D arrays.

Dynamic arrays expand to accept data you add to them and contract as you remove data from them.

Basic array techniques

The following sections describe how to reference array elements, create arrays, add and remove array elements, and copy arrays.

Referencing array elements

You reference array elements by enclosing the index with brackets: *arrayName*[*x*] where *x* is the index that you want to reference. In ColdFusion, array indexes are counted starting with position 1, which means that position 1 in the firstname array is referenced as firstname[1]. For 2D arrays, you reference an index by specifying two coordinates: `myarray[1][1]`.

You can use ColdFusion variables and expressions inside the square brackets to reference an index, as the following example shows:

```
<cfset myArray=ArrayNew(1)>
<cfset myArray[1]="First Array Element">
<cfset myArray[1 + 1]="Second Array" & "Element">
<cfset arrayIndex=3>
<cfset arrayElement="Third Array Element">
<cfset myArray[arrayIndex]=arrayElement>
<cfset myArray[arrayIndex + 1]="Fourth Array Element">
<cfdump var=#myArray#>
```

Note: The `IsDefined` function does not test the existence of array elements. Instead, put any code that might try to access an undefined array element in a try block and use a catch block to handle exceptions that arise if elements do not exist.

Creating arrays

In ColdFusion, you declare an array by assigning a variable name to the new array and specifying its dimensions, as follows:

```
<cfset mynewarray=ArrayNew(x)>
```

where *x* is the number of dimensions (from 1 to 3) in the array that you want to create.

Once you declare an array, you can add array elements, which you can then reference using the elements' indexes.

For example, suppose you declare a 1D array called firstname:

```
<cfset firstname=ArrayNew(1)>
```

The array firstname holds no data and is of an unspecified length. Next you add data to the array:

```
<cfset firstname[1]="Coleman">
<cfset firstname[2]="Charlie">
<cfset firstname[3]="Dexter">
```

After you add these names to the array, it has a length of 3.

Creating complex multidimensional arrays

ColdFusion supports dynamic multidimensional arrays. When you declare an array with the `ArrayNew` function, you specify the number of dimensions. You can create an asymmetrical array or increase an existing array's dimensions by nesting arrays as array elements.

It is important to know that when you assign one array (array1) to an element of another array (array2), array1 is copied into array2. The original copy of array1 still exists, independent of array2. You can then change the contents of the two arrays independently.

The best way to understand an asymmetrical array is by looking at it. The following example creates an asymmetric, multidimensional array and the `cfdump` tag displays the resulting array structure. Several array elements do not yet contain data.

```
<cfset myarray=ArrayNew(1)>
<cfset myotherarray=ArrayNew(2)>
<cfset biggerarray=ArrayNew(3)>

<cfset biggerarray[1][1][1]=myarray>
<cfset biggerarray[1][1][1][10]=3>
<cfset biggerarray[2][1][1]=myotherarray>
<cfset biggerarray[2][1][1][4][2]="five deep">

<cfset biggestarray=ArrayNew(3)>
<cfset biggestarray[3][1][1]=biggerarray>
<cfset biggestarray[3][1][1][2][3][1]="This is complex">
<cfset myarray[3]="Can you see me">

<cfdump var=#biggestarray#><br>
<cfdump var=#myarray#>
```

Note: The `cfdump` tag displays the entire contents of an array. It is an excellent tool for debugging arrays and array-handling code.

Reviewing the code

The following table describes the code:

Code	Description

`<cfset myarray=ArrayNew(1)>`
`<cfset myotherarray=ArrayNew(2)>`
`<cfset biggerarray=ArrayNew(3)>`
	Create three empty arrays, a 1D array, a 2D array, and a 3D array.

`<cfset biggerarray[1][1][1]=myarray>` `<cfset biggerarray[1][1][1][10]=3>`	Make element [1][1][1] of the 3D biggerarray array be a copy of the 1D array. Assign 3 to the [1][1][1][10] element of the resulting array. The biggerarray array is now asymmetric. For example, it does not have a [1][1][2][1] element.
`<cfset biggerarray[2][1][1]=myotherarray>` `<cfset biggerarray[2][1][1][4][2]="five deep">`	Make element [2][1][1] of the 3D array be the 2D array, and assign the [2][1][1][4][2] element the value "five deep". The biggerarray array is now even more asymmetric.

`<cfset biggestarray=ArrayNew(3)>`
`<cfset biggestarray[3][1][1]=biggerarray>`
`<cfset biggestarray[3][1][1][2][3][1]="This is complex">`
	Create a second 3D array. Make the [3][1][1] element of this array be a copy of the bigerarray array, and assign element [3][1][1][2][3][1].
	The resulting array is very complex and asymmetric.

Code	Description
`<cfset myarray[3]="Can you see me">`	Assign a value to element [3] of myarray.
`<cfdump var=#biggestarray#> <cfdump var=#myarray#>`	Use `cfdump` to view the structure of biggestarray and myarray. Notice that the "Can you see me" entry appears in myarray, but not in biggestarray, because biggestarray has a copy of the original myarray values and is not affected by the change to myarray.

Adding elements to an array

You can add an element to an array by assigning the element a value or by using a ColdFusion function.

Adding an array element by assignment

You can add elements to an array by defining the value of an array element, as shown in the following `cfset` tag:

```
<cfset myarray[5]="Test Message">
```

If an element does not exist at the specified index, ColdFusion creates it. If an element already exists at the specified index, ColdFusion replaces it with the new value. To prevent existing data from being overwritten, use the `ArrayInsertAt` function, as described in the next section.

If elements with lower-number indexes do not exist, they remain undefined. You must assign values to undefined array elements before you can use them. For example, the following code creates an array and an element at index 4. It outputs the contents of element 4, but generates an error when it tries to output the (nonexistent) element 3.

```
<cfset myarray=ArrayNew(1)>
<cfset myarray[4]=4>
<cfoutput>
  myarray4: #myarray[4]#<br>
  myarray3: #myarray[3]#<br>
</cfoutput>
```

Adding an array element with a function

You can use the following array functions to add data to an array:

Function	Description
`ArrayAppend`	Creates a new array element at the end of the array.
`ArrayPrepend`	Creates a new array element at the beginning of the array.
`ArrayInsertAt`	Inserts an array element at the specified index position.

Because ColdFusion arrays are dynamic, if you add or delete an element from the array, any higher-numbered index values all change. For example, the following code creates a two element array and displays the array contents. It then uses `ArrayPrepend` to insert a new element at the beginning of the array and displays the result. The data that was originally in indexes 1 and 2 is now in indexes 2 and 3.

```
<!--- Create an array with three elements --->
<cfset myarray=ArrayNew(1)>
<cfset myarray[1]="Original First Element">
<cfset myarray[2]="Original Second Element">
<!--- Use cfdump to display the array structure --->
<cfdump var=#myarray#>
<br>
<!--- Add a new element at the beginning of the array --->
<cfscript>
  ArrayPrepend(myarray, "New First Element");
</cfscript>
<!--- Use cfdump to display the new array structure --->
<cfdump var=#myarray#>
```

For more information about these array functions, see *CFML Reference*.

Deleting elements from an array

Use the `ArrayDeleteAt` function to delete data from the array at a particular index, instead of setting the data value to zero or an empty string. If you remove data from an array, the array resizes dynamically, as the following example shows:

```
<!--- Create an array with three elements --->
<cfset firstname=ArrayNew(1)>
<cfset firstname[1]="Robert">
  <cfset firstname[2]="Wanda">
  <cfset firstname[3]="Jane">
<!--- Delete the second element from the array --->
<cfset temp=ArrayDeleteAt(firstname, 2)>
<!--- Display the array length (2) and its two entries,
  which are now "Robert" and "Jane" --->
<cfoutput>
  The array now has #ArrayLen(firstname)# indexes<br>
  The first entry is #firstname[1]#<br>
  The second entry is #firstname[2]#<br>
  </cfoutput>
```

The `ArrayDeleteAt` function removed the original second element and resized the array so that it has two entries, with the second element now being the original third element.

Copying arrays

You can copy arrays of simple variables (numbers, strings, Boolean values, and date-time values) by assigning the original array to a new variable name. You do not have to use `ArrayNew` to create the new array first. When you assign the existing array to a new variable, ColdFusion creates a new array and copies the old array's contents to the new array. The following example creates and populates a two-element array. It then copies the original array, changes one element of the copied array and dumps both arrays. As you can see, the original array is unchanged and the copy has a new second element.

```
<cfset myArray=ArrayNew(1)>
<cfset myArray[1]="First Array Element">
<cfset myArray[2]="Second Array Element">
<cfset newArray=myArray>
<cfset newArray[2]="New Array Element 2">
<cfdump var=#myArray#><br>
<cfdump var=#newArray#>
```

If your array contains complex variables (structures, query objects, or external objects such as COM objects) assigning the original array to a new variable does not make a complete copy of the original array. The array structure is copied; however, the new array does not get its own copy of the complex data, only references to it. To demonstrate this behavior, run the following code:

```
Create an array that contains a structure.<br>
<cfset myStruct=StructNew()>
<cfset myStruct.key1="Structure key 1">
<cfset myStruct.key2="Structure key 2">
<cfset myArray=ArrayNew(1)>
<cfset myArray[1]=myStruct>
<cfset myArray[2]="Second array element">
<cfdump var=#myArray#><br>
<br>
Copy the array and dump it.<br>
<cfset myNewArray=myArray>
<cfdump var=#myNewArray#><br>
<br>
Change the values in the new array.<br>
<cfset myNewArray[1].key1="New first array element">
<cfset myNewArray[2]="New second array element">
<br>
Contents of the original array after the changes:<br>
<cfdump var=#myArray#><br>
Contents of the new array after the changes:<br>
<cfdump var=#myNewArray#>
```

The change to the new array also changes the contents of the structure in the original array.

To make a complete copy of an array that contains complex variables, use the `Duplicate` function.

Populating arrays with data

Array elements can store any values, including queries, structures, and other arrays. You can use a number of functions to populate an array with data, including `ArraySet`, `ArrayAppend`, `ArrayInsertAt`, and `ArrayPrepend`. These functions are useful for adding data to an existing array.

In particular, you should master the following basic techniques:

- Populating an array with the `ArraySet` function
- Populating an array with the `cfloop` tag
- Populating an array from a query

The following sections describe these techniques.

Populating an array with the ArraySet function

You can use the `ArraySet` function to populate a 1D array, or one dimension of a multidimensional array, with some initial value, such as an empty string or zero. This can be useful if you need to create an array of a certain size, but do not need to add data to it right away. One reason to do this is so that you can refer to all the array indexes. If you refer to an array index that does not contain some value, such as an empty string, you get an error.

The `ArraySet` function has the following form:

```
ArraySet (arrayname, startrow, endrow, value)
```

The following example initializes the array myarray, indexes 1 to 100, with an empty string:

```
ArraySet (myarray, 1, 100, "")
```

Populating an array with the cfloop tag

The `cfloop` tag provides a common and very efficient method for populating an array. The following example uses a `cfloop` tag and the `MonthAsString` function to populate a simple 1D array with the names of the months. A second `cfloop` outputs data in the array to the browser.

```
<cfset months=arraynew(1)>

<cfloop index="loopcount" from=1 to=12>
  <cfset months[loopcount]=MonthAsString(loopcount)>
</cfloop>

<cfloop index="loopcount" from=1 to=12>
    <cfoutput>
      #months[loopcount]#<br>
    </cfoutput>
</cfloop>
```

Using nested loops for 2D and 3D arrays

To output values from 2D and 3D arrays, you must employ nested loops to return array data. With a one-dimensional (1D) array, a single `cfloop` is sufficient to output data, as in the previous example. With arrays of dimension greater than one, you need to maintain separate loop counters for each array level.

Nesting cfloop tags for a 2D array

The following example shows how to handle nested `cfloop` tags to output data from a 2D array. It also uses nested `cfloop` tags to populate the array:

```
<cfset my2darray=arraynew(2)>
<cfloop index="loopcount" from=1 to=12>
  <cfloop index="loopcount2" from=1 to=2>
    <cfset my2darray[loopcount][loopcount2]=(loopcount * loopcount2)>
  </cfloop>
</cfloop>

<p>The values in my2darray are currently:</p>

<cfloop index="OuterCounter" from="1" to="#ArrayLen(my2darray)#">
  <cfloop index="InnerCounter" from="1"
      to="#ArrayLen(my2darray[OuterCounter])#">
    <cfoutput>
      <b>[#OuterCounter#][#InnerCounter#]</b>:
      #my2darray[OuterCounter][InnerCounter]#<br>
    </cfoutput>
  </cfloop>
</cfloop>
```

Nesting cfloop tags for a 3D array

For 3D arrays, you simply nest an additional `cfloop` tag. (This example does not set the array values first to keep the code short.)

```
<cfloop index="Dim1" from="1" to="#ArrayLen(my3darray)#">
  <cfloop index="Dim2" from="1" to="#ArrayLen(my3darray[Dim1])#">
    <cfloop index="Dim3" from="1"
        to="#ArrayLen(my3darray[Dim1][Dim2])#">
      <cfoutput>
        <b>[#Dim1#][#Dim2#][#Dim3#]</b>:
        #my3darray[Dim1][Dim2][Dim3]#<br>
      </cfoutput>
    </cfloop>
  </cfloop>
</cfloop>
```

Populating an array from a query

When populating an array from a query, keep the following things in mind:

- You cannot add query data to an array all at once. A looping structure is generally required to populate an array from a query.
- You can reference query column data using array-like syntax. For example, myquery.col_name[1] references data in the first row in the col_name column of the myquery query.
- Inside a `cfloop query=` loop, you do not have to specify the query name to reference the query's variables.

You can use a `cfset` tag with the following syntax to define values for array indexes:

```
<cfset arrayName[index]=queryColumn[row]>
```

In the following example, a `cfloop` tag places four columns of data from a sample data source into an array, myarray.

```
<!--- Do the query --->
<cfquery name="test" datasource="cfdocexamples">
  SELECT Emp_ID, LastName, FirstName, Email
  FROM Employees
</cfquery>

<!--- Declare the array --->
<cfset myarray=arraynew(2)>

<!--- Populate the array row by row --->
<cfloop query="test">
  <cfset myarray[CurrentRow][1]=Emp_ID>
  <cfset myarray[CurrentRow][2]=LastName>
  <cfset myarray[CurrentRow][3]=FirstName>
  <cfset myarray[CurrentRow][4]=Email>
</cfloop>

<!--- Now, create a loop to output the array contents --->
<cfset total_records=test.recordcount>
<cfloop index="Counter" from=1 to="#Total_Records#">
  <cfoutput>
    ID: #MyArray[Counter][1]#,
    LASTNAME: #MyArray[Counter][2]#,
    FIRSTNAME: #MyArray[Counter][3]#,
    EMAIL: #MyArray[Counter][4]# <br>
  </cfoutput>
</cfloop>
```

This example uses the query object built-in variable CurrentRow to index the first dimension of the array.

Array functions

The following functions are available for creating, editing, and handling arrays:

Function	Description
ArrayAppend	Appends an array element to the end of a specified array.
ArrayAvg	Returns the average of the values in the specified array.
ArrayClear	Deletes all data in a specified array.
ArrayDeleteAt	Deletes an element from a specified array at the specified index and resizes the array.
ArrayInsertAt	Inserts an element (with data) in a specified array at the specified index and resizes the array.
ArrayIsEmpty	Returns True if the specified array is empty of data.
ArrayLen	Returns the length of the specified array.
ArrayMax	Returns the largest numeric value in the specified array.
ArrayMin	Returns the smallest numeric value in the specified array.
ArrayNew	Creates a new array of specified dimension.
ArrayPrepend	Adds an array element to the beginning of the specified array.
ArrayResize	Resets an array to a specified minimum number of elements.
ArraySet	Sets the elements in a 1D array in a specified range to a specified value.
ArraySort	Returns the specified array with elements sorted numerically or alphanumerically.
ArraySum	Returns the sum of values in the specified array.
ArraySwap	Swaps array values in the specified indexes.
ArrayToList	Converts the specified 1D array to a list, delimited with the character you specify.
IsArray	Returns True if the value is an array.
ListToArray	Converts the specified list, delimited with the character you specify, to an array.

For more information about each of these functions, see *CFML Reference*.

About structures

ColdFusion *structures* consist of key-value pairs. Structures let you build a collection of related variables that are grouped under a single name. You can define ColdFusion structures dynamically.

You can use structures to refer to related values as a unit, rather than individually. To maintain employee lists, for example, you can create a structure that holds personnel information such as name, address, phone number, ID numbers, and so on. Then you can refer to this collection of information as a structure called *employee* rather than as a collection of individual variables.

A structure's *key* must be a string. The *values* associated with the key can be any valid ColdFusion value or object. It can be a string or integer, or a complex object such as an array or another structure. Because structures can contain any kind of data they provide a very powerful and flexible mechanism for representing complex data.

Structure notation

ColdFusion supports two types of notation for referencing structure contents. The notation that you use depends on your requirements.

Notation	Description
Object.property	You can refer to a property, prop, of an object, obj, as obj.prop. This notation, also called dot notation, is useful for simple assignments, as in this example: `depts.John="Sales"` Use this notation only when you know the property names (keys) in advance and they are strings, with no special characters, numbers, or spaces. You cannot use the dot notation when the property, or key, is dynamic.

Associative arrays	If you do not know the key name in advance, or it contains spaces, numbers, or special characters, you can use associative array notation. This notation uses structures as arrays with string indexes; for example:
	`depts["John"]="Sales"`
	`depts[employeeName]="Sales"`
	You can use a variable (such as employeeName) as an associative array index. Therefore, you must enclose any literal key names in quotation marks.
	For information on using associative array references containing variables, see Chapter 4, "Dynamically constructing structure references," on page 86.

Referencing complex structures

When a structure contains another structure, you reference the data in the nested structure by extending either object.property or associative array notation. You can even use a mixture of both notations.

For example, if structure1 has a key key1 whose value is a structure that has keys struct2key1, struct2key2, and so on, you can use any of the following references to access the data in the first key of the embedded structure:

```
Structure1.key1.Struct2key1
Structure1["key1"].Struct2key1
Structure1.key1["Struct2key1"]
Structure1["key1"]["Struct2key1"]
```

The following example shows various ways you can reference the contents of a complex structure:

```
<cfset myArray=ArrayNew(1)>
<cfset myArray[1]="2">
<cfset myArray[2]="3">
<cfset myStruct2=StructNew()>
<cfset myStruct2.struct2key1="4">
```

```
<cfset myStruct2.struct2key2="5">
<cfset myStruct=StructNew()>
<cfset myStruct.key1="1">
<cfset myStruct.key2=myArray>
<cfset myStruct.key3=myStruct2>
<cfdump var=#myStruct#><br>

<cfset key1Var="key1">
<cfset key2Var="key2">
<cfset key3Var="key3">
<cfset var2="2">

<cfoutput>
Value of the first key<br>
#mystruct.key1#<br>
#mystruct["key1"]#<br>
#mystruct[key1Var]#<br>
<br>
Value of the second entry in the key2 array<br>
#myStruct.key2[2]#<br>
#myStruct["key2"][2]#<br>
#myStruct[key2Var][2]#<br>
#myStruct[key2Var][var2]#<br>
<br>
Value of the struct2key2 entry in the key3 structure<br>
#myStruct.key3.struct2key2#<br>
#myStruct["key3"]["struct2key2"]#<br>
#myStruct[key3Var]["struct2key2"]#<br>
#myStruct.key3["struct2key2"]#<br>
#myStruct["key3"].struct2key2#<br>
<br>
</cfoutput>
```

Reviewing the code

The following table describes the code:

Code	Description

`<cfset myArray=ArrayNew(1)>`
`<cfset myArray[1]="2">`
`<cfset myArray[2]="3">`
`<cfset myStruct2=StructNew()>`
`<cfset myStruct2.struct2key1="4">`
`<cfset myStruct2.struct2key2="5">`
`<cfset myStruct=StructNew()>`
`<cfset myStruct.key1="1">`
`<cfset myStruct.key2=myArray>`
`<cfset myStruct.key3=myStruct2>`
	Create a structure with three entries: a string, an array, and an embedded structure.

`<cfdump var=#myStruct#> `	Display the complete structure.

`<cfset key1Var="key1">`
`<cfset key2Var="key2">`
`<cfset key3Var="key3">`
`<cfset var2="2">`
	Create variables containing the names of the myStruct keys and the number 2.

Code	Description
`<cfoutput>` `Value of the first key ` `#mystruct.key1# ` `#mystruct["key1"]# ` `#mystruct[key1Var]# ` ` `	Output the value of the structure's key1 (string) entry using the following notation: • object.property notation • associative array notation with a constant • associative array notation with a variable
`Value of the second entry in the key2 array ` `#myStruct.key2[2]# ` `#myStruct["key2"][2]# ` `#myStruct[key2Var][2]# ` `#myStruct[key2Var][var2]# ` ` `	Output the value of the second entry in the structure's key2 array using the following notation: • object.property notation • associative array notation with a constant • associative array notation with a variable • associative array notation with variables for both the array and the array index
`Value of the struct2key2 entry in the key3 structure ` `#myStruct.key3.struct2key2# ` `#myStruct["key3"]["struct2key2"]# ` `#myStruct[key3Var]["struct2key2"]# ` `#myStruct.key3["struct2key2"]# ` `#myStruct["key3"].struct2key2# ` ` ` `</cfoutput>`	Output the value of second entry in the structure's key3 embedded structure using the following notation: • object.property notation • associative array notation with two constants • associative array notation with a variable and a constant • object.property notation followed by associative array notation • associative array notation followed by object.property notation

Creating and using structures

This section explains how to create and use structures in ColdFusion. The sample code in this section uses a structure called *employee,* which is used to add new employees to a corporate information system.

Creating structures

You can create a structure by creating a first key-pair or by using the ColdFusion `StructNew` function.

Creating structures by assigning values

You can create a structure by assigning a key-value pair. For example, the following line creates a structure named myStruct with one element, name, that has the value Macromedia.

```
<cfset myStruct.name="Macromedia">
```

Note: You can also create a structure using the *`structname`*["*`keyname`*"] format; for example, `<cfset myStruct[“name”] ="Macromedia">`.

Creating structures using a function

You can create structures by assigning a variable name to the structure with the `StructNew` function as follows:

```
<cfset mystructure=StructNew()>
```

For example, to create a structure named departments, use the following syntax:

```
<cfset departments=StructNew()>
```

This creates an empty structure to which you can add data.

Use this technique to create structures if your application must run on ColdFusion server versions 5 and earlier.

Adding data elements to structures

You add an element to a structure by assigning the element a value or by using a ColdFusion function. It is cleaner and more efficient to use direct assignment, so only this technique is described.

You add structure key-value pairs by defining the value of the structure key, as shown in the following example:

```
<cfset myNewStructure.key1="A new structure with a new key">
<cfdump var=#myNewStructure#>
<cfset myNewStructure.key2="Now I've added a second key">
<cfdump var=#myNewStructure#>
```

Updating values in structures

You can update structure element values by assignment or by using the `StructUpdate` function. Direct assignment results in simpler code than using a function, so only the assignment technique is described.

To update a structure value, assign the key a new value. For example, the following code uses `cfset` and object.property notation to create a new structure element called departments.John, and changes John's department from Sales to Marketing. It then uses associative array notation to change his department to Facilities. Each time the department changes, it displays the results:

```
<cfset departments=structnew()>
<cfset departments.John = "Sales">
<cfoutput>
  Before the first change, John was in the #departments.John# Department<br>
</cfoutput>
<cfset Departments.John = "Marketing">
<cfoutput>
  After the first change, John is in the #departments.John# Department<br>
</cfoutput>
<cfset Departments["John"] = "Facilities">
<cfoutput>
  After the second change, John is in the #departments.John# Department<br>
</cfoutput>
```

Getting information about structures and keys

The following sections describe how to use ColdFusion functions to find information about structures and their keys.

Getting information about structures

To find out if a given value represents a structure, use the `IsStruct` function, as follows:

```
IsStruct(variable)
```

This function returns True if *variable* is a ColdFusion structure. (It also returns True if *variable* is a Java object that implements the java.util.Map interface.)

Structures are not indexed numerically, so to find out how many name-value pairs exist in a structure, use the `StructCount` function, as in the following example:

```
StructCount(employee)
```

To discover whether a specific Structure contains data, use the `StructIsEmpty` function, as follows:

```
StructIsEmpty(structure_name)
```

This function returns True if the structure is empty, and False if it contains data.

Finding a specific key and its value

To determine whether a specific key exists in a structure, use the `StructKeyExists` function, as follows:

```
StructKeyExists(structure_name, "key_name")
```

Do *not* put the name of the structure in quotation marks, but you do put the key name in quotation marks. For example, the following code displays the value of the MyStruct.MyKey only if it exists:

```
<cfif StructKeyExists(myStruct, "myKey")>
<cfoutput> #mystruct.myKey#</cfoutput><br>
</cfif>
```

You can use the `StructKeyExists` function to dynamically test for keys by using a variable to represent the key name. In this case, you do not put the variable in quotation marks. For example, the following code loops through the records of the GetEmployees query and tests the myStruct structure for a key that matches the query's LastName field. If ColdFusion finds a matching key, it displays the Last Name from the query and the corresponding entry in the structure.

```
<cfloop query="GetEmployees">
<cfif StructKeyExists(myStruct, LastName)>
<cfoutput>#LastName#: #mystruct[LastName]#</cfoutput><br>
</cfif>
</cfloop>
```

If the name of the key is known in advance, you can also use the ColdFusion `IsDefined` function, as follows:

```
IsDefined("structure_name.key")>
```

However, if the key is dynamic, or contains special characters, you must use the `StructKeyExists` function.

Note: Using `StructKeyExists` to test for the existence of a structure entry is more efficient than using `IsDefined`. ColdFusion scopes are available as structures and you can improve efficiency by using `StructKeyExists` to test for the existence of variables.

Getting a list of keys in a structure

To get a list of the keys in a CFML structure, you use the `StructKeyList` function, as follows:

```
<cfset temp=StructKeyList(structure_name, [delimiter] )>
```

You can specify any character as the delimiter; the default is a comma.

Use the `StructKeyArray` function to returns an array of keys in a structure, as follows:

```
<cfset temp=StructKeyArray(structure_name)>
```

Note: The `StructKeyList` and `StructKeyArray` functions do not return keys in any particular order. Use the `ListSort` or `ArraySort` functions to sort the results.

Copying structures

ColdFusion provides several ways to copy structures and create structure references. The following table lists these methods and describes their uses:

Technique	Use
`Duplicate` function	Makes a complete copy of the structure. All data is copied from the original structure to the new structure, including the contents of structures, queries, and other objects. As a result changes to one copy of the structure have no effect on the other structure. This function is useful when you want to move a structure completely into a new scope. In particular, if a structure is created in a scope that requires locking (for example, Application), you can duplicate it into a scope that does not require locking (for example, Request), and then delete it in the scope that requires locking.
`StructCopy` function	Makes a shallow copy of a structure. It creates a new structure and copies all simple variable and array values at the top level of the original structure to the new structure. However, it does not make copies of any structures, queries, or other objects that the original structure contains, or of any data inside these objects. Instead, it creates a reference in the new structure to the objects in the original structure. As a result, any change to these objects in one structure also changes the corresponding objects in the copied structure. The Duplicate replaces this function for most, if not all, purposes.
Variable assignment	Creates an additional reference, or alias, to the structure. Any change to the data using one variable name changes the structure that you access using the other variable name. This technique is useful when you want to add a local variable to another scope or otherwise change a variable's scope without deleting the variable from the original scope.

The following example shows the different effects of copying, duplicating, and assigning structure variables:

```
Create a new structure<br>
<cfset myNewStructure=StructNew()>
<cfset myNewStructure.key1="1">
<cfset myNewStructure.key2="2">
<cfset myArray=ArrayNew(1)>
<cfset myArray[1]="3">
<cfset myArray[2]="4">
<cfset myNewStructure.key3=myArray>
<cfset myNewStructure2=StructNew()>
<cfset myNewStructure2.Struct2key1="5">
<cfset myNewStructure2.Struct2key2="6">
<cfset myNewStructure.key4=myNewStructure2>
<cfdump var=#myNewStructure#><br>
<br>
A StructCopy copied structure<br>
<cfset CopiedStruct=StructCopy(myNewStructure)>
<cfdump var=#CopiedStruct#><br>
<br>
A Duplicated structure<br>
<cfset dupStruct=Duplicate(myNewStructure)>
<cfdump var=#dupStruct#><br>
<br>
A new reference to a structure<br>
<cfset structRef=myNewStructure>
<cfdump var=#structRef#><br>

<br>
Change a string, array element, and structure value in the StructCopy copy.<br>
<br>
<cfset CopiedStruct.key1="1A">
<cfset CopiedStruct.key3[2]="4A">
<cfset CopiedStruct.key4.Struct2key2="6A">
Original structure<br>
<cfdump var=#myNewStructure#><br>
Copied structure<br>
<cfdump var=#CopiedStruct#><br>
Duplicated structure<br>
<cfdump var=#DupStruct#><br>
Structure reference
<cfdump var=#structRef#><br>
<br>
Change a string, array element, and structure value in the Duplicate<br>
<br>
<cfset DupStruct.key1="1B">
<cfset DupStruct.key3[2]="4B">
<cfset DupStruct.key4.Struct2key2="6B">
Original structure<br>
<cfdump var=#myNewStructure#><br>
Copied structure<br>
<cfdump var=#CopiedStruct#><br>
Duplicated structure<br>
<cfdump var=#DupStruct#><br>
```

```
Structure reference
<cfdump var=#structRef#><br>
<br>
Change a string, array element, and structure value in the reference<br>
<br>
<cfset structRef.key1="1C">
<cfset structRef.key3[2]="4C">
<cfset structRef.key4.Struct2key2="6C">
Original structure<br>
<cfdump var=#myNewStructure#><br>
Copied structure<br>
<cfdump var=#CopiedStruct#><br>
Duplicated structure<br>
<cfdump var=#DupStruct#><br>
Structure reference
<cfdump var=#structRef#><br>
<br>
Clear the original structure<br>
<cfset foo=structclear(myNewStructure)>
Original structure:<br>
<cfdump var=#myNewStructure#><br>
Copied structure<br>
<cfdump var=#CopiedStruct#><br>
Duplicated structure<br>
<cfdump var=#DupStruct#><br>
Structure reference:<br>
<cfdump var=#structRef#><br>
```

Deleting structure elements and structures

To delete a key and its value from a structure, use the `StructDelete` function, as follows:

```
StructDelete(structure_name, key [, indicateNotExisting ])
```

The *indicateNotExisting* argument tells the function what to do if the specified key does not exist. By default, the function always returns True. However, if you specify True for the *indicateNotExisting* argument, the function returns True if the key exists and False if it does not.

You can also use the `StructClear` function to delete all the data in a structure but keep the structure instance itself, as follows:

```
StructClear(structure_name)
```

If you use `StructClear` to delete a structure that you have copied using the `StructCopy` function, the specified structure is deleted, but the copy is unaffected.

If you use `StructClear` to delete a structure that has a multiple references, the function deletes the contents of the structure and all references point to the empty structure, as shown in the following example:

```
<cfset myStruct.Key1="Macromedia">
Structure before StructClear<br>
<cfdump var="#myStruct#">
<cfset myCopy=myStruct>
<cfset StructClear(myCopy)>
After Clear:<br>
```

```
myStruct: <cfdump var="#myStruct#"><br>
myCopy: <cfdump var="#myCopy#">
```

Looping through structures

You can loop through a structure to output its contents, as shown in the following example:

```
<!--- Create a structure and set its contents --->
<cfset departments=structnew()>

<cfset val=StructInsert(departments, "John", "Sales")>
<cfset val=StructInsert(departments, "Tom", "Finance")>
<cfset val=StructInsert(departments, "Mike", "Education")>

<!--- Build a table to display the contents --->
<cfoutput>
<table cellpadding="2" cellspacing="2">
  <tr>
    <td><b>Employee</b></td>
    <td><b>Department</b></td>
  </tr>
  <!--- Use cfloop to loop through the departments structure.
  The item attribute specifies a name for the structure key. --->
  <cfloop collection=#departments# item="person">
    <tr>
      <td>#person#</td>
      <td>#Departments[person]#</td>
    </tr>
  </cfloop>
</table>
</cfoutput>
```

Structure examples

Structures are particularly useful for grouping together a set of variables under a single name. The example in this section uses structures to collect information from a form, and to submit that information to a custom tag, named `cf_addemployee`. For information on creating and using custom tags, see Chapter 11, “Creating and Using Custom CFML Tags,” on page 241.

Example file newemployee.cfm

The following ColdFusion page shows how to create structures and use them to add data to a database. It calls the `cf_addemployee` custom tag, which is defined in the addemployee.cfm file.

```
<html>
<head>
<title>Add New Employees</title>
</head>

<body>
<h1>Add New Employees</h1>
<!--- Action page code for the form at the bottom of this page --->

<!--- Establish parameters for first time through --->
<cfparam name="Form.firstname" default="">
```

```
<cfparam name="Form.lastname" default="">
<cfparam name="Form.email" default="">
<cfparam name="Form.phone" default="">
<cfparam name="Form.department" default="">

<!--- If at least the firstname form field is passed, create
   a structure named employee and add values --->
<cfif #Form.firstname# eq "">
  <p>Please fill out the form.</p>
<cfelse>
  <cfoutput>
   <cfscript>
    employee=StructNew();
    employee.firstname = Form.firstname;
    employee.lastname = Form.lastname;
    employee.email = Form.email;
    employee.phone = Form.phone;
    employee.department = Form.department;
   </cfscript>

<!--- Display results of creating the structure --->
   First name is #StructFind(employee, "firstname")#<br>
   Last name is #StructFind(employee, "lastname")#<br>
   EMail is #StructFind(employee, "email")#<br>
   Phone is #StructFind(employee, "phone")#<br>
   Department is #StructFind(employee, "department")#<br>
  </cfoutput>

  <!--- Call the custom tag that adds employees --->
  <cf_addemployee empinfo="#employee#">
</cfif>

<!--- The form for adding the new employee information --->
<hr>
<form action="newemployee.cfm" method="Post">
First Name: 
<input name="firstname" type="text" hspace="30" maxlength="30"><br>
Last Name: 
<input name="lastname" type="text" hspace="30" maxlength="30"><br>
EMail: 
<input name="email" type="text" hspace="30" maxlength="30"><br>
Phone: 
<input name="phone" type="text" hspace="20" maxlength="20"><br>
Department: 
<input name="department" type="text" hspace="30" maxlength="30"><br>

<input type="Submit" value="OK">
</form>
<br>
</body>
</html>
```

Reviewing the code

The following table describes the code:

Code	Description

`<cfparam name="Form.firstname" default="">`
`<cfparam name="Form.lastname" default="">`
`<cfparam name="Form.email" default="">`
`<cfparam name="Form.phone" default="">`
`<cfparam name="Form.department" default="">`
	Set default values of all form fields so that they exist the first time this page is displayed and can be tested.

`<cfif #form.firstname# eq "">` `Please fill out the form. `	Test the value of the form's firstname field. This field is required. The test is False the first time the page displays. If there is no data in the Form.firstname variable, display a message requesting the user to fill the form.
`<cfelse>` `<cfoutput>` `<cfscript>` `employee=StructNew();` `employee.firstname = Form.firstname;` `employee.lastname = Form.lastname;` `employee.email = Form.email;` `employee.phone = Form.phone;` `employee.department = Form.department;` `</cfscript>` `First name is #employee.firstname# ` `Last name is #employee.lastname# ` `EMail is #employee.email# ` `Phone is #employee.phone# ` `Department is #employee.department# ` `</cfoutput>`	If Form.firstname contains text, the user submitted the form. Use CFScript to create a new structure named employee and fill it with the form field data. Then display the contents of the structure

Code	Description
<cf_addemployee empinfo="#duplicate(employee)#"> </cfif>	Call the cf_addemployee custom tag and pass it a copy of the employee structure in the empinfo attribute. The duplicate function ensures that the custom tag gets a copy of the employee structure, not the original. While this is not necessary in this example, it is good practice because it prevents the custom tag from modifying the calling page's structure contents.
<form action="newemployee.cfm" method="Post"> First Name: <input name="firstname" type="text" hspace="30" maxlength="30"> Last Name: <input name="lastname" type="text" hspace="30" maxlength="30"> EMail: <input name="email" type="text" hspace="30" maxlength="30"> Phone: <input name="phone" type="text" hspace="20" maxlength="20"> <p>Department: <input name="department" type="text" hspace="30" maxlength="30"> <input type="Submit" value="OK"> </form>	The data form. When the user clicks Submit, the form posts the data to this ColdFusion page.

Example file addemployee.cfm

The following file is an example of a custom tag used to add employees. Employee information is passed through the employee structure (the empinfo attribute). For databases that do not support automatic key generation, you must also add the Emp_ID.

```
<cfif StructIsEmpty(attributes.empinfo)>
  <cfoutput>
    Error. No employee data was passed.<br>
  </cfoutput>
  <cfexit method="ExitTag">
<cfelse>
  <!--- Add the employee --->
  <cfquery name="AddEmployee" datasource="cfdocexamples">
    INSERT INTO Employees
      (FirstName, LastName, Email, Phone, Department)
    VALUES (
    '#attributes.empinfo.firstname#' ,
    '#attributes.empinfo.lastname#' ,
    '#attributes.empinfo.email#' ,
    '#attributes.empinfo.phone#' ,
    '#attributes.empinfo.department#' )
  </cfquery>
</cfif>
```

```
<cfoutput>
  <hr>Employee Add Complete
</cfoutput>
```

Reviewing the code

The following table describes the code:

Code	Description

`<cfif StructIsEmpty(Attributes.empinfo)>`
`  <cfoutput>`
`  Error. No employee data was passed.`
`  </cfoutput>`
`  <cfexit method="ExitTag">`
	If the custom tag was called without an `empinfo` attribute, display an error message and exit the tag.

`<cfelse>`
`  <cfquery name="AddEmployee" datasource="cfdocexamples">`
`  INSERT INTO Employees`
`  (FirstName, LastName, Email, Phone, Department)`
`  VALUES (`
`  '#attributes.empinfo.firstname#' ,`
`  '#attributes.empinfo.lastname#' ,`
`  '#attributes.empinfo.email#' ,`
`  '#attributes.empinfo.phone#' ,`
`  '#attributes.empinfo.department#' )`
`  </cfquery>`
`</cfif>`
	Add the employee data passed in the empinfo structure to the Employees table of the cfdocexamples database.
	Use direct references to the structure entries, not `StructFind` functions.
	If the database does not support automatic generation of the Emp_ID key, you must add an Emp_ID entry to the form and add it to the query.

`<cfoutput>`
`  <hr>Employee Add Complete`
`</cfoutput>`
	Display a completion message. This code does not have to be inside the `cfelse` block because the `cfexit` tag prevents it from being run if the empinfo structure is empty.

Structure functions

You can use the following functions to create and manage structures in ColdFusion applications. The table describes each function's purpose and provides specific, but limited, information that can assist you in determining whether to use the function instead of other techniques:

Function	Description
`Duplicate`	Returns a complete copy of the structure.
`IsStruct`	Returns True if the specified variable is a ColdFusion structure or a Java object that implements the java.util.Map interface.
`StructAppend`	Appends one structure to another.
`StructClear`	Removes all data from the specified structure.
`StructCopy`	Returns a "shallow" copy of the structure. All embedded objects are references to the objects in the original structure. The `Duplicate` function has replaced this function for most purposes.
`StructCount`	Returns the number of keys in the specified structure.

Function	Description
`StructDelete`	Removes the specified item from the specified structure.
`StructFind`	Returns the value associated with the specified key in the specified structure. This function is redundant with accessing structure elements using associative array notation.
`StructFindKey`	Searches through a structure for the specified key name and returns an array containing data on the found key or keys.
`StructFindValue`	Searches through a structure for the specified simple data value (for example, a string or number) and returns an array containing information on the value location in the structure.
`StructGet`	Returns a reference to a substructure contained in a structure at the specified path. This function is redundant with using direct reference to a structure. If you accidentally use this function on a variable that is not a structure, it replaces the value with an empty structure.
`StructInsert`	Inserts the specified key-value pair into the specified structure. Unlike a direct assignment statement, this function generates an error by default if the specified key exists in the structure.
`StructIsEmpty`	Indicates whether the specified structure contains data. Returns True if the structure contains no data, and False if it does contain data.
`StructKeyArray`	Returns an array of keys in the specified structure.
`StructKeyExists`	Returns True if the specified key is in the specified structure. You can use this function in place of the `IsDefined` function to check for the existence of variables in scopes that are available as structures.
`StructKeyList`	Returns a list of keys in the specified structure.
`StructNew`	Returns a new structure.
`StructSort`	Returns an array containing the key names of a structure in the order determined by the sort criteria.
`StructUpdate`	Updates the specified key with the specified value. Unlike a direct assignment statement, this function generates an error if the structure or key does not exist.

All functions except `StructDelete` throw an exception if a referenced key or structure does not exist.

For more information on these functions, see *CFML Reference*.

CHAPTER 6
Extending ColdFusion Pages with CFML Scripting

Macromedia ColdFusion MX offers a server-side scripting language, CFScript, that provides ColdFusion functionality in script syntax. This JavaScript-like language gives developers the same control flow as ColdFusion, but without tags. You can also use CFScript to write user-defined functions that you can use anywhere that a ColdFusion expression is allowed.

This chapter describes the CFScript language's functionality and syntax, and provides information on using CFScript effectively in ColdFusion pages.

Contents

About CFScript 123

The CFScript language 125

Using CFScript statements 128

Handling exceptions 136

CFScript example 137

About CFScript

CFScript is a language within a language. It is a scripting language that is similar to JavaScript but is simpler to use. Also, unlike JavaScript, CFScript only runs on the ColdFusion server; it does not run on the client system. CFScript code can use all the ColdFusion functions and expressions, and has access to all ColdFusion variables that are available in the script's scope.

CFScript provides a compact and efficient way to write ColdFusion logic. Typical uses of CFScript include the following:

- Simplifying and speeding variable setting
- Building compact JavaScript-like flow control structures
- Creating user-defined functions

Because you use functions and expressions directly in CFScript, you do not have to surround each assignment or function in a `cfset` tag. Also, CFScript assignments are often faster than `cfset` tags.

CFScript provides a set of decision and flow-control structures that are more familiar than ColdFusion tags to most programmers.

In addition to variable setting, other operations tend to be slightly faster in CFScript than in tags.

ColdFusion 5 and later releases let you use CFScript to create user-defined functions, or UDFs (also known as custom functions). You call UDFs in the same manner that you call standard ColdFusion functions. UDFs are to ColdFusion built-in functions what custom tags are to ColdFusion built-in tags. Typical uses of UDFs include data manipulation and mathematical calculation routines.

You cannot include ColdFusion tags in CFScript. However, a number of functions and CFScript statements are equivalent to commonly used tags. For more information, see "CFScript functional equivalents to ColdFusion tags" on page 127.

Comparing tags and CFScript

The following examples show how you can use CFML tags and CFScript to do the same thing. Each example takes data submitted from a form and puts it in a structure; if the form does not have a last name and department field, it displays a message.

Using CFML tags

```
<cfif IsDefined("Form.submit")>
  <cfif (Form.lastname NEQ "") AND (Form.department NEQ "")>
    <cfset employee=structnew()>
    <cfset employee.firstname=Form.firstname>
    <cfset employee.lastname=Form.lastname>
    <cfset employee.email=Form.email>
    <cfset employee.phone=Form.phone>
    <cfset employee.department=Form.department>
    <cfoutput>
      Adding #Form.firstname# #Form.lastname#<br>
    </cfoutput>
  <cfelse>
    <cfoutput>
      You must enter a Last Name and Department.<br>
    </cfoutput>
  </cfif>
</cfif>
```

Using CFScript

```
<cfscript>
  if (IsDefined("Form.submit")) {
    if ((Form.lastname NEQ "") AND (Form.department NEQ "")) {
      employee=StructNew();
      employee.firstname=Form.firstname;
      employee.lastname=Form.lastname;
      employee.email=Form.email;
      employee.phone=Form.phone;
      employee.department=Form.department;
      WriteOutput("Adding #Form.firstname# #Form.lastname# <br>");
      }
    else
      WriteOutput("You must enter a Last Name and Department.<br>");
    }
</cfscript>
```

The CFScript language

This section explains the syntax of the CFScript language.

Identifying CFScript

You enclose CFScript regions inside `<cfscript>` and `</cfscript>` tags. No other CFML tags are allowed inside a `cfscript` region. The following lines show a minimal script:

```
<cfscript>
a = 2;
</cfscript>
```

Variables

CFScript variables can be of any ColdFusion type, such as numbers, strings, arrays, queries, and objects. The CFScript code can read and write any variables that are available in the page that contains the script. This includes all common scope variables, such as session, application, and server variables.

Expressions and operators

CFScript supports all CFML expressions. CFML expressions include operators (such as +, -, EQ, and so on), as well as all CFML functions. As in all ColdFusion expressions, you must use CFML operators, such as LT, GT, and EQ. You cannot use JavaScript operators, such as <, >, ==, or ++.

For information about CFML expressions, operators, and functions, see Chapter 4, "Using Expressions and Number Signs," on page 75.

Statements

CFScript supports the following statements:

assignment	for-in	try-catch
function call	while	function (function definition)
if-else	do-while	var (in custom functions only)
switch-case-default	break	return (in custom functions only)
for	continue	

The following rules apply to statements:

- You must put a semicolon at the end of a statement.
- Line breaks are ignored. A single statement can cross multiple lines.
- White space is ignored. For example, it does not matter whether you precede a semicolon with a space character.
- Use curly braces to group multiple statements together into one logical statement unit.
- Unless otherwise indicated, you can use any ColdFusion expression in the body of a statement.

Note: This chapter documents all statements except `function`, `var`, and `return`. For information on these statements, see Chapter 9, "Defining functions in CFScript," on page 173.

Statement blocks

Curly brace characters ({ and }) group multiple CFScript statements together so that they are treated as a single unit or statement. This enables you to create code blocks in conditional statements, such as the following:

```
if(score GT 0) {
  result = "positive";
  Positives = Positives + 1;
}
```

In this example, both assignment statements are executed if the score is greater than 0. If they were not in the code block, only the first line would execute.

You do not have to put brace characters on their own lines in the code. For example, you could put the open brace in the preceding example on the same line as the `if` statement, and some programmers use this style. However, putting at least the ending brace on its own line makes it easier to read the code and separate out code blocks.

Comments

CFScript has two forms of comments: single line and multiline.

A single line comment begins with two forward slashes (//) and ends at the line end; for example:

```
//This is a single-line comment.
//This is a second single-line comment.
```

A multiline comment starts with a /* marker and continues until it reaches a */ marker; for example:

```
/*This is a multiline comment.
  You do not need to start each line with a comment indicator.
  This is the last line in the comment. */
```

The following rules apply to comments:

- Comments do not have to start at the beginning of a line. They can follow active code on a line. For example, the following line is valid:

  ```
  MyVariable = 12; // Set MyVariable to the default value.
  ```

- The end of a multiline comment can be followed on the same line by active code. For example, the following line is valid, although it is poor coding practice:

  ```
  End of my long comment */ foo = "bar";
  ```

- You can use multiline format for a comment on a single line, for example:

  ```
  /*This is a single line comment using multiline format. */
  ```

- You cannot nest /* and */ markers inside other comment lines.
- CFML comments (<!--- and --->) do not work in CFScript.

Reserved words

In addition to the names of ColdFusion functions and words reserved by ColdFusion expressions (such as NOT, AND, IS, and so on), the following words are reserved in CFScript. Do not use these words as variables or identifiers in your scripting code:

break	default	function	switch
case	do	if	try
catch	else	in	var
continue	for	return	while

Differences from JavaScript

Although CFScript and JavaScript are similar, they have several key differences. The following list identifies CFScript features that differ from JavaScript:

- CFScript uses ColdFusion expressions, which are neither a subset nor a superset of JavaScript expressions. For example, there is no < operator in CFScript; you use the LT operator instead.
- Variable declarations are only used in user-defined functions.
- CFScript is case-insensitive.
- All statements end with a semicolon and line breaks in the code are ignored.
- Assignments are statements, not expressions.
- JavaScript objects, such as Window and Document, are not available.
- Only the ColdFusion server processes CFScript. There is no client-side CFScript.

CFScript limitation

You cannot include ColdFusion tags in CFScript. However, you can include `cfscript` blocks inside other ColdFusion tags, such as `cfoutput`.

CFScript functional equivalents to ColdFusion tags

Tag	CFScript equivalent
`cfset`	Direct assignment, such as Myvar=1;
`cfoutput`	`WriteOutput` function
`cfif`, `cfelseif`, `cfelse`	`if` and `else` statements
`cfswitch`, `cfcase`, `cfdefaultcase`	`switch`, `case`, and `default` statements
Indexed `cfloop`	`for` loops
Conditional `cfloop`	`while` loops and `do while` loops
Structure `cfloop`	`for in` loop. (There is no equivalent for queries, lists, or objects.)
`cfbreak`	`break` statement. CFScript also has a `continue` statement that has no equivalent CFML tag.

Tag	CFScript equivalent
cftry, cfcatch	try and catch statements
cfcookie	Direct assignment of Cookie scope memory-only variables. You cannot use direct assignment to set persistent cookies that are stored on the user's system.
cfobject	CreateObject function

For example, the following example loops through a query in CFScript:

```
...
<cfscript>
// Loop through the qGetEmails RecordSet
for (x = 1; x LTE qGetEmails.RecordCount; x=x+1) {
  This_id = qGetEmails.Emails_id[x];
  This_Subject = qGetEmails.Subject[x];
  This_RecFrom = qGetEmails.RecFrom[x];
  This_SentTo = qGetEmails.SentTo[x];
  This_dReceived = qGetEmails.dReceived[x];
   This_Body = qGetEmails.Body[x];
... // More code goes here.
}
</cfscript>
```

Using CFScript statements

The following sections describe how to use these CFScript statements:

- Using assignment statements and functions
- Using conditional processing statements
- Using looping statements

Using assignment statements and functions

CFScript assignment statements are the equivalent of the cfset tag. These statements have the following form:

```
lval = expression;
```

lval is any ColdFusion variable reference; for example:

```
x = "positive";
y = x;
a[3]=5;
structure.member=10;
ArrayCopy=myArray;
```

You can use ColdFusion function calls, including UDFs, directly in CFScript. For example, the following line is a valid CFScript statement:

```
StructInsert(employee,"lastname",FORM.lastname);
```

Using conditional processing statements

CFScript includes the following conditional processing statements:

- `if` and `else` statements, which serve the same purpose as the `cfif`, `cfelseif`, and `cfelse` tags
- `switch`, `case`, and `default` statements, which are the equivalents of the `cfswitch`, `cfcase`, and `cfdefaultcase` tags

Using if and else statements

The `if` and `else` statements have the following syntax:

```
if(expr) statement [else statement]
```

In its simplest form, an if statement looks like this:

```
if(value EQ 2700)
  message = "You've reached the maximum";
```

A simple if-else statement looks like the following:

```
if(score GT 1)
  result = "positive";
else
  result = "negative";
```

CFScript does not include an elseif statement. However, you can use an `if` statement immediately after an `else` statement to create the equivalent of a `cfelseif` tag, as the following example shows:

```
if(score GT 1)
  result = "positive";
else if(score EQ 0)
  result = "zero";
else
  result = "negative";
```

As with all conditional processing statements, you can use curly braces to enclose multiple statements for each condition, as follows:

```
if(score GT 1) {
  result = "positive";
  message = "The result was positive.";
  }
else {
  result = "negative";
  message = "The result was negative.";
  }
```

Note: Often, you can make your code clearer by using braces even where they are not required.

Using switch and case statements

The `switch` statement and its dependent `case` and `default` statements have the following syntax:

```
switch (expression) {
  case constant: [case constant:]... statement(s) break;
  [case constant: [case constant:]... statement(s) break;]...
  [default: statement(s)] }
```

Use the following rules and recommendations for `switch` statements:

- You cannot mix Boolean and numeric constant values in a `switch` statement.
- Each constant value must be a constant (that is, not a variable, a function, or other expression).
- Multiple `case` *constant:* statements can precede the statement or statements to execute if any of the cases are true. This lets you specify several matches for one code block.
- No two constant values can be the same.
- The statements following the colon in a `case` statement block do not have to be in braces. If a constant value equals the `switch` expression, ColdFusion executes all statements through the `break` statement.
- The `break` statement at the end of the `case` statement tells ColdFusion to exit the `switch` statement. ColdFusion does not generate an error message if you omit a `break` statement. However, if you omit it, ColdFusion executes all the statements in the following case statement, *even if that case is false*. In nearly all circumstances, this is not what you want to do.
- You can have only one `default` statement in a `switch` statement block. ColdFusion executes the statements in the `default` block if none of the `case` statement constants equals the expression value.
- The `default` statement does not have to follow all `switch` statements, but it is good programming practice to do so. If any `switch` statements follow the `default` statement you must end the `default` block code with a `break` statement.
- The `default` statement is not required. However, you should use one if the `case` constants do not include all possible values of the expression.
- The `default` statement does not have to follow all the `case` statements; however, it is good programming practice to put it there.

The following `switch` statement takes the value of a name variable:

1. If the name is John or Robert, it sets both the male variable and the found variable to True.
2. If the name is Mary, it sets the male variable to False and the found variable to True.
3. Otherwise, it sets the found variable to False.

```
switch(name) {
  case "John": case "Robert":
    male=True;
    found=True;
    break;
  case "Mary":
    male=False;
    found=True;
    break;
  default:
    found=False;
} //end switch
```

Using looping statements

CFScript provides a richer selection of looping constructs than those supplied by CFML tags. It enables you to create efficient looping constructs similar to those in most programming and scripting languages. CFScript provides the following looping constructs:

- For
- While
- Do-while
- For-in

CFScript also includes the `continue` and `break` statements that control loop processing.

The following sections describe these types of loops and their uses.

Using for loops

The for loop has the following format:

```
for (initial-expression; test-expression; final-expression) statement
```

The *initial-expression* and *final-expression* can be one of the following:

- A single assignment expression; for example, x=5 or loop=loop+1
- Any ColdFusion expression; for example, SetVariable("a",a+1)
- Empty

The *test-expression* can be one of the following:

- Any ColdFusion expression; for example:

```
A LT 5
index LE x
status EQ "not found" AND index LT end
```

- Empty

Note: The test expression is re-evaluated before each repeat of the loop. If code inside the loop changes any part of the test expression, it can affect the number of iterations in the loop.

The *statement* can be a single semicolon terminated statement or a statement block in curly braces.

When ColdFusion executes a for loop, it does the following:

1. Evaluates the *initial expression.*
2. Evaluates the *test-expression.*
3. If the *test-expression* is False, exits the loop and processing continues following the *statement.*

   If the *test-expression* is True:

   a. Executes the *statement* (or statement block).

   b. Evaluates the *final-expression.*

   c. Returns to Step 2.

For loops are most commonly used for processing in which an index variable is incremented each time through the loop, but it is not limited to this use.

The following simple for loop sets each element in a 10-element array with its index number.

```
for(index=1;
  index LT 10;
  index = index + 1)
  a[index]=index;
```

The following, more complex, example demonstrates two features:

- The use of curly braces to group multiple statements into a single block.
- An empty condition statement. All loop control logic is in the statement block.

```
<cfscript>
strings=ArrayNew(1);
ArraySet(strings, 1, 10, "lock");
strings[5]="key";
indx=0;
for( ; ; ) {
  indx=indx+1;
  if(Find("key",strings[indx],1)) {
    WriteOutput("Found key at " & indx & ".<br>");
    break;
    }
  else if (indx IS ArrayLen(strings)) {
    WriteOutput("Exited at " & indx & ".<br>");
    break;
    }
}
</cfscript>
```

This example shows one important issue that you must remember when creating loops: you must always ensure that the loop ends. If this example lacked the `else if` statement, and there was no "key" in the array, ColdFusion would loop forever or until a system error occurred; you would have to stop the server to end the loop.

The example also shows two issues with index arithmetic: in this form of loop you must make sure to initialize the index, and you must keep track of where the index is incremented. In this case, because the index is incremented at the top of the loop, you must initialize it to 0 so it becomes 1 in the first loop.

Using while loops

The while loop has the following format:

```
while (expression) statement
```

The `while` statement does the following:

1. Evaluates the *expression.*
2. If the *expression* is True, it does the following:
   a. Executes the *statement*, which can be a single semicolon-terminated statement or a statement block in curly braces.
   b. Returns to step 1.

If the *expression* is False, processing continues with the next statement.

The following example uses a `while` loop to populate a 10-element array with multiples of five.

```
a = ArrayNew(1);
loop = 1;
while (loop LE 10) {
  a[loop] = loop * 5;
  loop = loop + 1;
  }
```

As with other loops, you must make sure that at some point the `while` *expression* is False and you must be careful to check your index arithmetic.

Using do-while loops

The do-while loop is like a while loop, except that it tests the loop condition after executing the loop statement block. The do-while loop has the following format:

```
do statement while (expression);
```

The `do while` statement does the following:

1. Executes the *statement*, which can be a single semicolon-terminated statement or a statement block in curly braces.
2. Evaluates the *expression.*
3. If the *expression* is true, it returns to step 1.

If the *expression* is False, processing continues with the next statement.

The following example, like the while loop example, populates a 10-element array with multiples of 5:

```
a = ArrayNew(1);
loop = 1;
do {
  a[loop] = loop * 5;
  loop = loop + 1;
}
while (loop LE 10);
```

Because the loop index increment follows the array value assignment, the example initializes the loop variable to 1 and tests to make sure that it is less than or equal to 10.

The following example generates the same results as the previous two examples, but it increments the index before assigning the array value. As a result, it initializes the index to 0, and the end condition tests that the index is less than 10.

```
a = ArrayNew(1);
loop = 0;
do {
  loop = loop + 1;
  a[loop] = loop * 5;
}
while (loop LT 10);
```

The following example loops through a query:

```
<cfquery ... name="myQuery">
... sql goes here...
</cfquery>
<cfscript>
if (myQuery.RecordCount gt 0) {
  currRow=1;
  do {
    theValue=myQuery.myField[CurrRow];
    currRow=currRow+1;
  } while (currRow LTE myQuery.RecordCount);
}
</cfscript>
```

Using for-in loops

The for-in loop loops over the elements in a ColdFusion structure. It has the following format:

```
for (variable in structure) statement
```

The *variable* can be any ColdFusion identifier; it holds each structure key name as ColdFusion loops through the structure. The *structure* must be the name of an existing ColdFusion structure. The *statement* can be a single semicolon terminated statement or a statement block in curly braces.

The following example creates a structure with three elements. It then loops through the structure and displays the name and value of each key. Although the curly braces are not required here, they make it easier to determine the contents of the relatively long `WriteOutput` function. In general, you can make structured control flow, especially loops, clearer by using curly braces.

```
myStruct=StructNew();
myStruct.productName="kumquat";
mystruct.quality="fine";
myStruct.quantity=25;
for (keyName in myStruct) {
  WriteOutput("myStruct." & Keyname & " has the value: " &
    myStruct[keyName] &"<br>");
}
```

Note: Unlike the `cfloop` tag, CFScript for-in loops do not provide built-in support for looping over queries and lists.

Using continue and break statements

The continue and break statements enable you to control the processing inside loops:

- The `continue` statement tells ColdFusion to skip to the beginning of the next loop iteration.
- The `break` statement exits the current loop or `case` statement.

Using continue

The `continue` statement ends the current loop iteration, skips any code following it in the loop, and jumps to the beginning of the next loop iteration. For example, the following code loops through an array and display's each value that is not an empty string:

```
for ( loop=1; loop LE 10; loop = loop+1) {
  if(a[loop] EQ "") continue;
  WriteOutput(loop);
}
```

(To test this code snippet, you must first create an array, a, with 10 or more elements, some of which are not empty strings.)

In general, the `continue` statement is particularly useful if you loop over arrays or structures and you want to skip processing for array elements or structure members with specific values, such as the empty string.

Using break

The `break` statement exits the current loop or `case` statement. Processing continues at the next CFScript statement. You end `case` statement processing blocks with a `break` statement. You can also use a test case with a `break` statement to prevent infinite loops, as shown in the following example. This script loops through an array and prints out the array indexes that contain the value key. It uses a conditional test and a `break` statement to make sure that the loop ends when at the end of the array.

```
strings=ArrayNew(1);
ArraySet(strings, 1, 10, "lock");
strings[5]="key";
strings[9]="key";
indx=0;
for( ; ; ) {
  indx=indx+1;
  if(Find("key",strings[indx],1)) {
    WriteOutput("Found a key at " & indx & ".<br>");
```

```
    }
  else if (indx IS ArrayLen(strings)) {
    WriteOutput("Array ends at index " & indx & ".<br>");
    break;
  }
}
```

Handling exceptions

ColdFusion provides two statements for exception handling in CFScript: `try` and `catch`. These statements are equivalent to the CFML `cftry` and `cfcatch` tags.

Note: This section does not explain exception-handling concepts. For a discussion of exception handling in ColdFusion, see Chapter 14, "Handling Errors," on page 307.

Exception handling syntax and rules

Exception-handling code in CFScript has the following format:

```
try {
  Code where exceptions will be caught
}
catch(exceptionType exceptionVariable) {
  Code to handle exceptions of type exceptionType
  that occur in the try block
}
...
catch(exceptionTypeN exceptionVariableN) {
  Code to handle exceptions of type
  exceptionTypeN that occur in the try block
}
```

Note: In CFScript, `catch` statements follow the `try` block; you do not put them inside the `try` block. This structure differs from that of the `cftry` tag, which must include the `cfcatch` tags in its body.

When you have a `try` statement, you must have a `catch` statement. In the `catch` block, the *exceptionVariable* variable contains the exception type. This variable is the equivalent of the `cfcatch` tag `cfcatch.Type` built-in variable.

Exception handling example

The following code shows exception handling in CFScript. It uses a `CreateObject` function to create a Java object. The catch statement executes only if the `CreateObject` function generates an exception. The displayed information includes the exception message; the except.Message variable is the equivalent of calling the Java `getMessage` method on the returned Java exception object.

```
<cfscript>
  try {
    emp = CreateObject("Java", "Employees");
  }
  catch(Any excpt) {
    WriteOutput("The application was unable to perform a required
  operation.<br>
  Please try again later.<br>If this problem persists, contact
  Customer Service and include the following information:<br>
      #excpt.Message#<br>");
  }
</cfscript>
```

CFScript example

The example in this section uses the following CFScript features:

- Variable assignment
- Function calls
- For loops
- If-else statements
- `WriteOutput` functions
- Switch statements

The example uses CFScript without any other ColdFusion tags. It creates a structure of course applicants. This structure contains two arrays; the first has accepted students, the second has rejected students. The script also creates a structure with rejection reasons for some (but not all) rejected students. It then displays the accepted applicants followed by the rejected students and their rejection reasons.

```
<html>
<head>
  <title>CFScript Example</title>
</head>
<body>
<cfscript>

  //Set the variables

  acceptedApplicants[1] = "Cora Cardozo";
  acceptedApplicants[2] = "Betty Bethone";
  acceptedApplicants[3] = "Albert Albertson";
  rejectedApplicants[1] = "Erma Erp";
  rejectedApplicants[2] = "David Dalhousie";
  rejectedApplicants[3] = "Franny Farkle";
```

```
  applicants.accepted=acceptedApplicants;
  applicants.rejected=rejectedApplicants;

  rejectCode=StructNew();
  rejectCode["David Dalhousie"] = "score";
  rejectCode["Franny Farkle"] = "too late";

  //Sort and display accepted applicants

  ArraySort(applicants.accepted,"text","asc");
  WriteOutput("The following applicants were accepted:<hr>");
  for (j=1;j lte ArrayLen(applicants.accepted);j=j+1) {
    WriteOutput(applicants.accepted[j] & "<br>");
  }
  WriteOutput("<br>");

  //sort and display rejected applicants with reasons information

  ArraySort(applicants.rejected,"text","asc");
  WriteOutput("The following applicants were rejected:<hr>");
  for (j=1;j lte ArrayLen(applicants.rejected);j=j+1) {
    applicant=applicants.rejected[j];
    WriteOutput(applicant & "<br>");
    if (StructKeyExists(rejectCode,applicant)) {
      switch(rejectCode[applicant]) {
        case "score":
          WriteOutput("Reject reason: Score was too low.<br>");
          break;
        case "late":
          WriteOutput("Reject reason: Application was late.<br>");
          break;
        default:
          WriteOutput("Rejected with invalid reason code.<br>");
      } //end switch
    } //end if
    else {
      WriteOutput("Reject reason was not defined.<br>");
    } //end else
    WriteOutput("<br>");
  } //end for
</cfscript>
```

Reviewing the code

The following table describes the code:

Code	Description

`<cfscript>`
`acceptedApplicants[1] = "Cora Cardozo";`
`acceptedApplicants[2] = "Betty Bethone";`
`acceptedApplicants[3] = "Albert Albertson";`
`rejectedApplicants[1] = "Erma Erp";`
`rejectedApplicants[2] = "David Dalhousie";`
`rejectedApplicants[3] = "Franny Farkle";`
`applicants.accepted=acceptedApplicants;`
`applicants.rejected=rejectedApplicants;`
 	
`rejectCode=StructNew();`
`rejectCode["David Dalhousie"] = "score";`
`rejectCode["Franny Farkle"] = "too late";`
	Creates two one-dimensional arrays, one with the accepted applicants and another with the rejected applicants. The entries in each array are in random order.
	Creates a structure and assign each array to an element of the structure.
	Creates a structure with rejection codes for rejected applicants. The `rejectedCode` structure does not have entries for all rejected applicants, and one of its values does not match a valid code. The structure element references use associative array notation in order to use key names that contain spaces.

`ArraySort(applicants.accepted,"text","asc");` `WriteOutput("The following applicants were accepted:<hr>");` `for (j=1;j lte ArrayLen(applicants.accepted);j=j+1) {` `WriteOutput(applicants.accepted[j] & " ");` `}` `WriteOutput(" ");`	Sorts the accepted applicants alphabetically. Displays a heading. Loops through the accepted applicants and writes their names. Braces enhance clarity, although they are not needed for a single statement loop. Writes an additional line break at the end of the list of accepted applicants.
`ArraySort(applicants.rejected,"text","asc");` `WriteOutput("The following applicants were rejected:<hr>");`	Sorts `rejectedApplicants` array alphabetically and writes a heading.
`for (j=1;j lte ArrayLen(applicants.rejected);j=j+1) {` `applicant=applicants.rejected[j];` `WriteOutput(applicant & " ");`	Loops through the rejected applicants. Sets the applicant variable to the applicant name. This makes the code clearer and enables you to easily reference the `rejectCode` array later in the block. Writes the applicant name.
`if (StructKeyExists(rejectCode,applicant)) {` `switch(rejectcode[applicant]) {` `case "score":` `WriteOutput("Reject reason: Score was too low. ");` `break;` `case "late":` `WriteOutput("Reject reason: Application was late. ");` `break;` `default:` `WriteOutput("Rejected with invalid reason code. ");` `} //end switch` `} //end if`	Checks the `rejectCode` structure for a rejection code for the applicant. If a code exists, enters a switch statement that examines the rejection code value. If the rejection code value matches one of the known codes, displays an expanded explanation of the meaning. Otherwise (the default case), displays an indication that the rejection code is not valid. Comments at the end of blocks help clarify the control flow.

Code	Description
`else {` `  WriteOutput("Reject reason was not defined.` `   ");` `{`	If there is no entry for the applicant in the `rejectCode` structure, displays a message indicating that the reason was not defined.
`  WriteOutput(" ");` `} //end for` `</cfscript>`	Displays a blank line after each rejected applicant. Ends the for loop that handles each rejected applicant. Ends the CFScript.

CHAPTER 7
Using Regular Expressions in Functions

Regular expressions let you perform string matching operations using ColdFusion functions. This chapter describes how regular expressions work with the following functions:

- `REFind`
- `REFindNoCase`
- `REReplace`
- `REReplaceNoCase`

This chapter does *not* apply to regular expressions used in the `cfinput` and `cftextinput` tags. These tags use JavaScript regular expressions, which have a slightly different syntax than ColdFusion regular expressions. For information on JavaScript regular expressions, see Chapter 27, "Building Dynamic Forms with cfform Tags," on page 631.

Contents

About regular expressions 141
Regular expression syntax 143
Using backreferences 150
Returning matched subexpressions 153
Regular expression examples 157
Types of regular expression technologies 158

About regular expressions

In traditional string matching, as used by the ColdFusion `Find` and `Replace` functions, you provide the string pattern to search for and the string to search. The following example searches a string for the pattern " BIG " and returns a string index if found. The *string index* is the location in the search string where the string pattern begins.

```
<cfset IndexOfOccurrence=Find(" BIG ", "Some BIG string")>
<!--- The value of IndexOfOccurrence is 5 --->
```

You must provide the exact string pattern to match. If the exact pattern is not found, `Find` returns an index of 0. Because you must specify the exact string pattern to match, matches for dynamic data can be very difficult, if not impossible, to construct.

The next example uses a regular expression to perform the same search. This example searches for the first occurrence in the search string of any string pattern that consists entirely of uppercase letters enclosed by spaces:

```
<cfset IndexOfOccurrence=REFind(" [A-Z]+ ", "Some BIG string")>
<!--- The value of IndexOfOccurrence is 5 --->
```

The regular expression " [A-Z]+ " matches any string pattern consisting of a leading space, followed by any number of uppercase letters, followed by a trailing space. Therefore, this regular expression matches the string " BIG " and any string of uppercase letters enclosed in spaces.

By default, the matching of regular expressions is case-sensitive. You can use the case-insensitive functions, `REFindNoCase` and `REReplaceNoCase`, for case-insensitive matching.

Because you often process large amounts of dynamic textual data, regular expressions are invaluable in writing complex ColdFusion applications.

Using ColdFusion regular expression functions

ColdFusion supplies four functions that work with regular expressions:

- `REFind`
- `REFindNoCase`
- `REReplace`
- `REReplaceNoCase`

`REFind` and `REFindNoCase` use a regular expression to search a string for a pattern and return the string index where it finds the pattern. For example, the following function returns the index of the first instance of the string " BIG ":

```
<cfset IndexOfOccurrence=REFind(" BIG ", "Some BIG BIG string")>
<!--- The value of IndexOfOccurrence is 5 --->
```

To find the next occurrence of the string " BIG ", you must call the `REFind` function a second time. For an example of iterating over a search string to find all occurrences of the regular expression, see “Returning matched subexpressions” on page 153.

`REReplace` and `REReplaceNoCase` use regular expressions to search through a string and replace the string pattern that matches the regular expression with another string. You can use these functions to replace the first match, or to replace all matches.

For detailed descriptions of the ColdFusion functions that use regular expressions, see *CFML Reference*.

Basic regular expression syntax

The simplest regular expression contains only a literal characters. The literal characters must match exactly the text being searched. For example, you can use the regular expression function `REFind` to find the string pattern " BIG ", just as you can with the `Find` function:

```
<cfset IndexOfOccurrence=REFind(" BIG ", "Some BIG string")>
<!--- The value of IndexOfOccurrence is 5 --->
```

In this example, `REFind` must match the exact string pattern " BIG ".

To use the full power of regular expressions, combine literal characters with character sets and special characters, as in the following example:

```
<cfset IndexOfOccurrence=REFind(" [A-Z]+ ", "Some BIG string")>
<!--- The value of IndexOfOccurrence is 5 --->
```

The literal characters of the regular expression consists of the space characters at the beginning and end of the regular expression. The character set consists of that part of the regular expression in square brackets. This character set specifies to find a single uppercase letter from A to Z, inclusive. The plus sign (+) after the square brackets is a special character specifying to find one or more occurrences of the character set.

If you removed the + from the regular expression in the previous example, " [A-Z] " matches a literal space, followed by any single uppercase letter, followed by a single space. This regular expression matches " B " but not " BIG ". The `REFind` function returns 0 for the regular expression, meaning that it did not find a match.

You can construct very complicated regular expressions containing literal characters, character sets, and special characters. Like any programming language, the more you work with regular expressions, the more you can accomplish with them. The examples in this section are fairly basic. For more examples, see "Regular expression examples" on page 157.

Regular expression syntax

This section describes the basic rules for creating regular expressions.

Using character sets

The pattern within the square brackets of a regular expression defines a character set that is used to match a single character. For example, the regular expression " [A-Za-z] " specifies to match any single uppercase or lowercase letter enclosed by spaces. In the character set, a hyphen indicates a range of characters.

The regular expression " B[IAU]G " matches the strings " BIG ", " BAG ", and " BUG ", but does not match the string " BOG ".

If you specified the regular expression as " B[IA][GN] ", the concatenation of character sets creates a regular expression that matches the corresponding concatenation of characters in the search string. This regular expression matches a space, followed by "B", followed by an "I" or "A", followed by a "G" or "N", followed by a trailing space. The regular expression matches " BIG ", " BAG ", "BIN ", and "BAN ".

The regular expression [A-Z][a-z]* matches any word that starts with an uppercase letter and is followed by zero or more lowercase letters. The special character * after the closing square bracket specifies to match zero or more occurrences of the character set.

Note: The * only applies to the character set that immediately precedes it, not to the entire regular expression.

A + after the closing square bracket specifies to find one or more occurrences of the character set. You interpret the regular expression " **[A-Z]+** " as matching one or more uppercase letters enclosed by spaces. Therefore, this regular expression matches " BIG " and also matches " LARGE ", " HUGE ", " ENORMOUS ", and any other string of uppercase letters surrounded by spaces.

Considerations when using special characters

Since a regular expression followed by an * can match zero instances of the regular expression, it can also match the empty string. For example,

```
<cfoutput>REReplace("Hello","[T]*","7","ALL") -
  #REReplace("Hello","[T]*","7","ALL")#<BR>
</cfoutput>
```

results in the following output:

```
REReplace("Hello","[T]*","7","ALL") - 7H7e7l7l7o
```

The regular expression [T]* can match empty strings. It first matches the empty string before "H" in "Hello". The "ALL" argument tells `REReplace` to replace all instances of an expression. The empty string before "e" is matched and so on until the empty string before "o" is matched.

This result might be unexpected. The workarounds for these types of problems are specific to each case. In some cases you can use [T]+, which requires at least one "T", instead of [T]*. Alternatively, you can specify an additional pattern after [T]*.

In the following examples the regular expression has a "W" at the end:

```
<cfoutput>REReplace("Hello World","[T]*W","7","ALL") -
  #REReplace("Hello World","[T]*W","7","ALL")#<BR></cfoutput>
```

This expression results in the following more predictable output:

```
REReplace("Hello World","[T]*W","7","ALL") - Hello 7orld
```

Finding repeating characters

In some cases, you might want to find a repeating pattern of characters in a search string. For example, the regular expression "a{2,4}" specifies to match two to four occurrences of "a". Therefore, it would match: "aa", "aaa", "aaaa", but not "a" or "aaaaa". In the following example, the `REFind` function returns an index of 6:

```
<cfset IndexOfOccurrence=REFind("a{2,4}", "hahahaaahaaaahaaaaahhh")>
<!--- The value of IndexOfOccurrence is 6--->
```

The regular expression "[0-9]{3,}" specifies to match any integer number containing three or more digits: "123", "45678", etc. However, this regular expression does not match a one-digit or two-digit number.

You use the following syntax to find repeating characters:

- `{m,n}`

  Where *m* is 0 or greater and *n* is greater than or equal to *m*. Match *m* through *n* (inclusive) occurrences.

  The expression {0,1} is equivalent to the special character ?.

- `{m,}`

  Where *m* is 0 or greater. Match at least *m* occurrences. The syntax `{,n}` is not allowed.

  The expression {1,} is equivalent to the special character +, and {0,} is equivalent to *.

- `{m}`

  Where *m* is 0 or greater. Match exactly *m* occurrences.

Case sensitivity in regular expressions

ColdFusion supplies case-sensitive and case-insensitive functions for working with regular expressions. `REFind` and `REReplace` perform case-sensitive matching and `REFindNoCase` and `REReplaceNoCase` perform case-insensitive matching.

You can build a regular expression that models case-insensitive behavior, even when used with a case-sensitive function. To make a regular expression case insensitive, substitute individual characters with character sets. For example, the regular expression [Jj][Aa][Vv][Aa], when used with the case-sensitive functions `REFind` or `REReplace`, matches all of the following string patterns:

- JAVA
- java
- Java
- jAva
- All other combinations of case

Using subexpressions

Parentheses group parts of regular expressions together into grouped *subexpressions* that you can treat as a single unit. For example, the regular expression "ha" specifies to match a single occurrence of the string. The regular expression "(ha)+" matches one or more instances of “ha”.

In the following example, you use the regular expression "B(ha)+" to match the letter "B" followed by one or more occurrences of the string "ha":

```
<cfset IndexOfOccurrence=REFind("B(ha)+", "hahaBhahahaha")>
<!--- The value of IndexOfOccurrence is 5 --->
```

You can use the special character | in a subexpression to create a logical "OR". You can use the following regular expression to search for the word "jelly" or "jellies":

```
<cfset IndexOfOccurrence=REFind("jell(y|ies)", "I like peanut butter and
  jelly">
<!--- The value of IndexOfOccurrence is 26--->
```

Using special characters

Regular expressions define the following list of special characters:

```
+ * ? . [ ^ $ ( ) { | \
```

In some cases, you use a special character as a literal character. For example, if you want to search for the plus sign in a string, you have to escape the plus sign by preceding it with a backslash:

```
"\+"
```

The following table describes the special characters for regular expressions:

Special Character	Description
\	A backslash followed by any special character matches the literal character itself, that is, the backslash escapes the special character. For example, "\+" matches the plus sign, and "\\" matches a backslash.
.	A period matches any character, including newline. To match any character except a newline, use [^#chr(13)##chr(10)#], which excludes the ASCII carriage return and line feed codes. The corresponding escape codes are \r and \n.
[ ]	A one-character character set that matches any of the characters in that set. For example, "[akm]" matches an “a”, “k”, or “m”. A hyphen in a character set indicates a range of characters; for example, [a-z] matches any single lowercase letter. If the first character of a character set is the caret (^), the regular expression matches any character *except* those in the set. It does not match the empty string. For example, [^akm] matches any character except “a”, “k”, or “m”. The caret loses its special meaning if it is not the first character of the set.
^	If the caret is at the beginning of a regular expression, the matched string must be at the beginning of the string being searched. For example, the regular expression "^ColdFusion" matches the string "ColdFusion lets you use regular expressions" but not the string "In ColdFusion, you can use regular expressions."
$	If the dollar sign is at the end of a regular expression, the matched string must be at the end of the string being searched. For example, the regular expression "ColdFusion$" matches the string "I like ColdFusion" but not the string "ColdFusion is fun."
?	A character set or subexpression followed by a question mark matches zero or one occurrences of the character set or subexpression. For example, xy?z matches either “xyz” or “xz”.
|	The OR character allows a choice between two regular expressions. For example, jell(y|ies) matches either “jelly” or “jellies”.
+	A character set or subexpression followed by a plus sign matches one or more occurrences of the character set or subexpression. For example, [a-z]+ matches one or more lowercase characters.
*	A character set or subexpression followed by an asterisk matches zero or more occurrences of the character set or subexpression. For example, [a-z]* matches zero or more lowercase characters.

Special Character	Description
()	Parentheses group parts of a regular expression into subexpressions that you can treat as a single unit. For example, (ha)+ matches one or more instances of "ha".

(?x)	If at the beginning of a regular expression, it specifies to ignore whitespace in the regular expression and lets you use ## for end-of-line comments. You can match a space by escaping it with a backslash.
	For example, the following regular expression includes comments, preceded by ##, that are ignored by ColdFusion:
	`reFind("(?x)`
	`one ##first option`
	`|two ##second option`
	`|three\ point\ five ## note escaped spaces`
	`", "three point five")`

(?m)	If at the beginning of a regular expression, it specifies the multiline mode for the special characters ^ and $. When used with ^, the matched string can be at the start of the of entire search string or at the start of new lines, denoted by a linefeed character or chr(10), within the search string. For $, the matched string can be at the end the search string or at the end of new lines. Multiline mode does not recognize a carriage return, or chr(13), as a new line character. The following example searches for the string "two" across multiple lines: `#reFind("(?m)^two", "one#chr(10)#two")#` This example returns 4 to indicate that it matched "two" after the chr(10) linefeed. Without (?m), the regular expression would not match anything, because ^ only matches the start of the string. The character (?m) does not affect \A or \Z, which always match the start or end of the string, respectively. For information on \A and \Z, see "Using escape sequences" on page 148.
(?i)	If at the beginning of a regular expression for `REFind()`, it specifies to perform a case-insensitive compare. For example, the following line would return an index of 1: `#reFind("(?i)hi", "HI")#` If you omit the (?i), the line would return an index of zero to signify that it did not find the regular expression.

Special Character	Description

(?=...)	If at the beginning of a regular expression, it specifies to use positive lookahead when searching for the regular expression.
	Positive lookahead tests for the parenthesized subexpression like regular parenthesis, but does not include the contents in the match - it merely tests to see if it is there in proximity to the rest of the expression.
	For example, consider the expression to extract the protocol from a URL:
	`<cfset regex = "http(?=://)">`
	`<cfset string = "http://">`
	`<cfset result = reFind(regex, string, 1, "yes")>`
	`mid(string, result.pos[1], result.len[1])`
	This example results in the string "http". The lookahead parentheses ensure that the "://" is there, but does not include it in the result. If you did not use lookahead, the result would include the extraneous "://".
	Lookahead parentheses do not capture text, so backreference numbering will skip over these groups. For more information on backreferencing, see "Using backreferences" on page 150.

(?!...)	If at the beginning of a regular expression, it specifies to use negative lookahead. Negative is just like positive lookahead, as specified by (?=...), except that it tests for the absence of a match. Lookahead parentheses do not capture text, so backreference numbering will skip over these groups. For more information on backreferencing, see "Using backreferences" on page 150.
(?:...)	If you prefix a subexpression with "?:", ColdFusion performs all operations on the subexpression except that it will not capture the corresponding text for use with a back reference.

You must be aware of the following considerations when using special characters in character sets, such as [a-z]:

- To include a hyphen (-) in the square brackets of a character set as a literal character, you cannot escape it as you can other special characters because ColdFusion always interprets a hyphen as a range indicator. Therefore, if you use a literal hyphen in a character set, make it the last character in the set.
- To include a closing square bracket (]) in the character set, escape it with a backslash, as in [1-3\]A-z]. You do not have to escape the] character outside of the character set designator.

Using escape sequences

Escape sequences are special characters in regular expressions preceded by a backslash (\). You typically use escape sequences to represent special characters within a regular expression. For example, the escape sequence \t represents a tab character within the regular expression, and the \d escape sequence specifies any digit, similar to [0-9]. In ColdFusion the escape sequences are case-sensitive.

The following table lists the escape sequences that ColdFusion supports:

Escape Sequence	Description
\b	Specifies a boundary defined by a transition from an alphanumeric character to a nonalphanumeric character, or from a nonalphanumeric character to an alphanumeric character. For example, the string " Big" contains boundary defined by the space (nonalphanumeric character) and the "B" (alphanumeric character). The following example uses the \b escape sequence in a regular expression to locate the string "Big" at the end of the search string and not the fragment "big" inside the word "ambiguous". reFindNoCase("\bBig\b", "Don't be ambiguous about Big.") <!--- The value of IndexOfOccurrence is 26 ---> When used inside of a character set (e.g. [\b]), it specifies a backspace
\B	Specifies a boundary defined by no transition of character type. For example, two alphanumeric character in a row or two nonalphanumeric character in a row; opposite of \b.
\A	Specifies a beginning of string anchor, much like the ^ special character. However, unlike ^, you cannot combine \A with (?m) to specify the start of newlines in the search string.
\Z	Specifies an end of string anchor, much like the $ special character. However, unlike $, you cannot combine \Z with (?m) to specify the end of newlines in the search string.
\n	Newline character
\r	Carriage return
\t	Tab
\f	Form feed
\d	Any digit, similar to [0-9]
\D	Any nondigit character, similar to [^0-9]
\w	Any alphanumeric character, similar to [[:alnum:]]
\W	Any nonalphanumeric character, similar to [^[:alnum:]]
\s	Any whitespace character including tab, space, newline, carriage return, and form feed. Similar to [\t\n\r\f].
\S	Any nonwhitespace character, similar to [^ \t\n\r\f]
\xdd	A hexadecimal representation of character, where d is a hexadecimal digit
\ddd	An octal representation of a character, where d is an octal digit, in the form \000 to \377

Using character classes

In character sets within regular expressions, you can include a character class. You enclose the character class inside square brackets, as the following example shows:

```
REReplace ("Macromedia Web Site","[[:space:]]","*","ALL")
```

This code replaces all the spaces with *, producing this string:

```
Macromedia*Web*Site
```

You can combine character classes with other expressions within a character set. For example, the regular expression [[:space:]123] searches for a space, 1, 2, or 3. The following example also uses a character class in a regular expression:

```
<cfset IndexOfOccurrence=REFind("[[:space:]][A-Z]+[[:space:]]",
  "Some BIG string")>
<!--- The value of IndexOfOccurrence is 5 --->
```

The following table shows the character classes that ColdFusion supports. Regular expressions using these classes match any Unicode character in the class, not just ASCII or ISO-8859 characters.

Character class	Matches
:alpha:	Any alphabetic character.
:upper:	Any uppercase alphabetic character.
:lower:	Any lowercase alphabetic character
:digit:	Any digit. Same as \d.
:alnum:	Any alphanumeric character. Same as \w.
:xdigit:	Any hexadecimal digit. Same as [0-9A-Fa-f].
:blank:	Space or a tab.
:space:	Any whitespace character. Same as \s.
:print:	Any alphanumeric, punctuation, or space character.
:punct:	Any punctuation character
:graph:	Any alphanumeric or punctuation character.
:cntrl:	Any character not part of the character classes [:upper:], [:lower:], [:alpha:], [:digit:], [:punct:], [:graph:], [:print:], or [:xdigit:].
:word:	Any alphanumeric character, plus the underscore (_)
:ascii:	The ASCII characters, in the Hexadecimal range 0 - 7F

Using backreferences

You use parenthesis to group components of a regular expression into subexpressions. For example, the regular expression “(ha)+” matches one or more occurrences of the string “ha”.

ColdFusion performs an additional operation when using subexpressions; it automatically saves the characters in the search string matched by a subexpression for later use within the regular expression. Referencing the saved subexpression text is called *backreferencing*.

You can use backreferencing when searching for repeated words in a string, such as “the the” or “is is”. The following example uses backreferencing to find all repeated words in the search string and replace them with an asterisk:

```
REReplace("There is is coffee in the the kitchen",
  "[ ]+([A-Za-z]+)[ ]+\1"," * ","ALL")
```

Using this regular expression, ColdFusion detects the two occurrences of “is” as well as the two occurrences of “the”, replaces them with an asterisk enclosed in spaces, and returns the following string:

```
There * coffee in * kitchen
```

You interpret the regular expression [ ]+([A-Za-z]+)[ ]+\1 as follows:

Use the subexpression ([A-Za-z]+) to search for character strings consisting of one or more letters, enclosed by one or more spaces, [ ]+, followed by the same character string that matched the first subexpression, \1.

You reference the matched characters of a subexpression using a slash followed by a digit *n* (*n*) where the first subexpression in a regular expression is referenced as \1, the second as \2, etc. The next section includes an example using multiple backreferences.

Using backreferences in replacement strings

You can use backreferences in the replacement string of both the `REReplace` and `REReplaceNoCase` functions. For example, to replace the first repeated word in a text string with a single word, use the following syntax:

```
REReplace("There is is a cat in in the kitchen",
  "([A-Za-z ]+)\1","\1")
```

This results in the sentence:

```
"There is a cat in in the kitchen"
```

You can use the optional fourth parameter to `REReplace`, *scope*, to replace all repeated words, as in the following code:

```
REReplace("There is is a cat in in the kitchen",
  "([A-Za-z ]+)\1","\1","ALL")
```

This results in the following string:

```
“There is a cat in the kitchen”
```

The next example uses two backreferences to reverse the order of the words "apples" and "pears" in a sentence:

```
<cfset astring = "apples and pears, apples and pears, apples and pears">
<cfset newString = REReplace("#astring#", "(apples) and (pears)",
  "\2 and \1","ALL")>
```

In this example, you reference the subexpression (apples) as \1 and the subexpression (pears) as \2. The `REReplace` function returns the string:

```
"pears and apples, pears and apples, pears and apples"
```

Note: To use backreferences in either the search string or the replace string, you must use parentheses within the regular expression to create the corresponding subexpression. Otherwise, ColdFusion throws an exception.

Using backreferences to perform case conversions in replacement strings

The `REReplace` and `REReplaceNoCase` functions support special characters in replacement strings to convert replacement characters to uppercase or lowercase. The following table describes these special characters:

Special character	Description
\u	Converts the next character to uppercase.
\l	Converts the next character to lowercase.
\U	Converts all characters to uppercase until encountering \E.
\L	Converts all characters to lowercase until encountering \E.
\E	End \U or \L.

To include a literal \u, or other code, in a replacement string, escape it with another backslash; for example \\u .

For example, the following statement replaces the uppercase string "HELLO" with a lowercase "hello". This example uses backreferences to perform the replacement. For more information on using backreferences, see "Using backreferences in replacement strings" on page 151.

```
reReplace("HELLO", "([[:upper:]]*)", "Don't shout\scream \L\1")
```

The result of this example is the string "Don't shout\scream hello".

Escaping special characters in replacement strings

You use the backslash character, \, to escape backreference and case-conversion characters in replacement strings. For example, to include a literal "\u" in a replacement string, escape it, as in "\\u".

Omitting subexpressions from backreferences

By default, a set of parentheses will both group the subexpression and capture its matched text for later referral by backreferences. However, if you insert "?:" as the first characters of the subexpression, ColdFusion performs all operations on the subexpression except that it will not capture the corresponding text for use with a back reference.

This is useful when alternating over subexpressions containing differing numbers of groups would complicate backreference numbering. For example, consider an expression to insert a "Mr." in between Bonjour|Hi|Hello and Bond, using a nested group for alternating between Hi & Hello:

```
<cfset regex = "(Bonjour|H(?:i|ello))( Bond)">
<cfset replaceString = "\1 Mr.\2">
<cfset string = "Hello Bond">
#reReplace(string, regex, replaceString)#
```

This example returns "Hello Mr. Bond". If you did not prohibit the capturing of the Hi/Hello group, the \2 backreference would end up referring to that group instead of " Bond", and the result would be "Hello Mr.ello".

Returning matched subexpressions

The `REFind` and `REFindNoCase` functions return the location in the search string of the first match of the regular expression. Even though the search string in the next example contains two matches of the regular expression, the function only returns the index of the first:

```
<cfset IndexOfOccurrence=REFind(" BIG ", "Some BIG BIG string")>
<!--- The value of IndexOfOccurrence is 5 --->
```

To find all instances of the regular expression, you must call the `REFind` and `REFindNoCase` functions multiple times.

Both the `REFind` and `REFindNoCase` functions take an optional third parameter that specifies the starting index in the search string for the search. By default, the starting location is index 1, the beginning of the string.

To find the second instance of the regular expression in this example, you call `REFind` with a starting index of 8:

```
<cfset IndexOfOccurrence=REFind(" BIG ", "Some BIG BIG string", 8)>
<!--- The value of IndexOfOccurrence is 9 --->
```

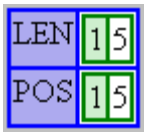
In this case, the function returns an index of 9, the starting index of the second string " BIG ".

To find the second occurrence of the string, you must know that the first string occurred at index 5 and that the string's length was 5. However, `REFind` only returns starting index of the string, not its length. So, you either must know the length of the matched string to call `REFind` the second time, or you must use subexpressions in the regular expression.

The `REFind` and `REFindNoCase` functions let you get information about matched subexpressions. If you set these functions' fourth parameter, `ReturnSubExpression`, to True, the functions return a CFML structure with two arrays, `pos` and `len`, containing the positions and lengths of text strings that match the subexpressions of a regular expression, as the following example shows:

```
<cfset sLenPos=REFind(" BIG ", "Some BIG BIG string", 1, "True")>
<cfoutput>
  <cfdump var="#sLenPos#">
</cfoutput><br>
```

The following figure shows the output of the `cfdump` tag:



Element one of the `pos` array contains the starting index in the search string of the string that matched the regular expression. Element one of the `len` array contains length of the matched string. For this example, the index of the first " BIG " string is 5 and its length is also 5. If there are no occurrences of the regular expression, the `pos` and `len` arrays each contain one element with a value of 0.

You can use the returned information with other string functions, such as `mid`. The following example returns that part of the search string matching the regular expression:

```
<cfset myString="Some BIG BIG string">
<cfset sLenPos=REFind(" BIG ", myString, 1, "True")>
<cfoutput>
  #mid(myString, sLenPos.pos[1], sLenPos.len[1])#
</cfoutput>
```

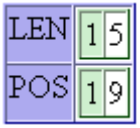
Each additional element in the `pos` array contains the position of the first match of each subexpression in the search string. Each additional element in `len` contains the length of the subexpression's match.

In the previous example, the regular expression " BIG " contained no subexpressions. Therefore, each array in the structure returned by `REFind` contains a single element.

After executing the previous example, you can call `REFind` a second time to find the second occurrence of the regular expression. This time, you use the information returned by the first call to make the second:

```
<cfset newstart = sLenPos.pos[1] + sLenPos.len[1] - 1>
<!--- subtract 1 because you need to start at the first space --->
<cfset sLenPos2=REFind(" BIG ", "Some BIG BIG string", newstart, "True")>
<cfoutput>
  <cfdump var="#sLenPos2#">
</cfoutput><br>
```

The following figure shows the output of the `cfdump` tag:



If you include subexpressions in your regular expression, each element of `pos` and `len` after element one contains the position and length of the first occurrence of each subexpression in the search string.

In the following example, the expression [A-Za-z]+ is a subexpression of a regular expression. The first match for the expression ([A-Za-z]+)[ ]+, is "is is".

```
<cfset sLenPos=REFind("([A-Za-z]+)[ ]+\1",
  "There is is a cat in in the kitchen", 1, "True")>
<cfoutput>
  <cfdump var="#sLenPos#">
</cfoutput><br>
```

The following figure shows the output of the `cfdump` tag:

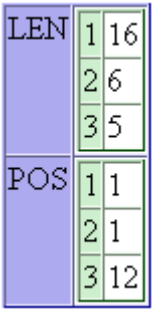


The entries sLenPos.pos[1] and sLenPos.len[1] contain information about the match of the entire regular expression. The array elements sLenPos.pos[2] and sLenPos.len[2] contain information about the first subexpression (“is”). Because `REFind` returns information on the first regular expression match only, the `sLenPos` structure does not contain information about the second match to the regular expression, "in in".

The regular expression in the following example uses two subexpressions. Therefore, each array in the output structure contains the position and length of the first match of the entire regular expression, the first match of the first subexpression, and the first match of the second subexpression.

```
<cfset sString = "apples and pears, apples and pears, apples and pears">
<cfset regex = "(apples) and (pears)">
<cfset sLenPos = REFind(regex, sString, 1, "True")>
<cfoutput>
  <cfdump var="#sLenPos#">
</cfoutput><br><br>
```

The following figure shows the output of the `cfdump` tag:



For a full discussion of subexpression usage, see the sections on `REFind` and `REFindNoCase` in the ColdFusion functions chapter in *CFML Reference*.

Specifying minimal matching

The regular expression quantifiers ?, *, +, {min,} and {min,max} specify a minimum and/or maximum number of instances of a given expression to match. By default, ColdFusion locates the greatest number characters in the search string that match the regular expression. This behavior is called *maximal matching*.

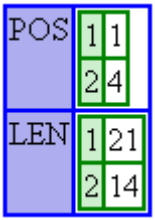
For example, you use the regular expression "(.*)" to search the string "one two". The regular expression "(.*)", matches both of the following:

- one
- one two

By default, ColdFusion always tries to match the regular expression to the largest string in the search string. The following code shows the results of this example:

```
<cfset sLenPos=REFind("<b>(.*)</b>", "<b>one</b> <b>two</b>", 1, "True")>
<cfoutput>
  <cfdump var="#sLenPos#">
</cfoutput><br>
```

The following figure shows the output of the `cfdump` tag:



Thus, the starting position of the string is 1 and its length is 21, which corresponds to the largest of the two possible matches.

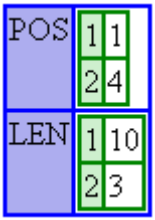
However, sometimes you might want to override this default behavior to find the shortest string that matches the regular expression. ColdFusion includes minimal-matching quantifiers that let you specify to match on the smallest string. The following table describes these expressions:

Expression	Description
*?	minimal-matching version of *
+?	minimal-matching version of +
??	minimal-matching version of ?
{min,}?	minimal-matching version of {min,}
{min,max}?	minimal-matching version of {min,max}
{n}?	(no different from {n}, supported for notational consistency)

If you modify the previous example to use the minimal-matching syntax, the code is as follows:

```
<cfset sLenPos=REFind("<b>(.*?)</b>", "<b>one</b> <b>two</b>", 1, "True")>
<cfoutput>
  <cfdump var="#sLenPos#">
</cfoutput><br>
```

The following figure shows the output of the `cfdump` tag:



Thus, the length of the string found by the regular expression is 10, corresponding to the string "one".

Regular expression examples

The following examples show some regular expressions and describe what they match:

Expression	Description
`[\?&]value=`	A URL parameter value in a URL.
`[A-Z]:(\\[A-Z0-9_]+)+`	An uppercase DOS/Windows path in which (a) is not the root of a drive, and (b) has only letters, numbers, and underscores in its text.
`^[A-Za-z][A-Za-z0-9_]*`	A ColdFusion variable with no qualifier.
`([A-Za-z][A-Za-z0-9_]*)(\.[A-Za-z][A-Za-z0-9_]*)?`	A ColdFusion variable with no more than one qualifier; for example, Form.VarName, but not Form.Image.VarName.
`(\+|-)?[1-9][0-9]*`	An integer that does not begin with a zero and has an optional sign.
`(\+|-)?[0-9]+(\.[0-9]*)?`	A real number.
`(\+|-)?[1-9]\.[0-9]*E(\+|-)?[0-9]+`	A real number in engineering notation.
a{2,4}	Two to four occurrences of “a”: aa, aaa, aaaa.
(ba){3,}	At least three “ba” pairs: bababa, babababa, and so on.

Regular expressions in CFML

The following examples of CFML show some common uses of regular expression functions:

Expression	Returns
`REReplace (CGI.Query_String, "CFID=[0-9]+[&]*", "")`	The query string with parameter CFID and its numeric value stripped out.
`REReplace(“I Love Jellies”, ”[[:lower:]]”,”x”,”ALL”`	I Lxxx Jxxxxxx
`REReplaceNoCase(“cabaret”,”[A-Z]”, ”G”,”ALL”)`	GGGGGGG
`REReplace (Report,"\$[0-9,]*\.[0-9]*", "$***.**")", "")`	The string value of the variable Report with all positive numbers in the dollar format changed to "$***.**".

Expression	Returns
`REFind ("[Uu]\.?[Ss]\.?[Aa]\.?", Report )`	The position in the variable Report of the first occurrence of the abbreviation USA. The letters can be in either case and the abbreviation can have a period after any letter.
`REFindNoCase("a+c","ABCAACCDD")`	4
`REReplace("There is is coffee in the the kitchen", "([A-Za-z]+)[ ]+\1","*","ALL")`	There * coffee in * kitchen
`REReplace(report, "<[^>]*>", "", "All")`	Removes all HTML tags from a string value of the report variable.

Types of regular expression technologies

Many types of regular expression technologies are available to programmers. JavaScript, Perl, and POSIX are all examples of different regular expression technologies. Each technology has its own syntax specifications and is not necessarily compatible with other technologies.

ColdFusion supports regular expressions that are Perl compliant with a few exceptions:

- A period, ., always matches newlines.
- In replacement strings, use \n instead of $n for backreference variables. ColdFusion escapes all $ in the replacement string.
- You do not have to escape backslashes in replacement strings. ColdFusion escapes them, with the exception of case conversion sequences or escaped versions (e.g. \u or \\u).
- Embedded modifiers ((?i), etc.) always affect the entire expression, even if they are inside a group.
- \Q and the combinations \u\L and \l\U are not supported in replacement strings.

The following Perl statements are not supported:

- Lookbehind (?<=) (<?!)
- \x{hhhh}
- \N
- \p
- \C

An excellent reference on regular expressions is *Mastering Regular Expressions*, by Jeffrey E. F. Friedl, O'Reilly & Associates, Inc., 1997, ISBN: 1-56592-257-3, available at www.oreilly.com.

PART II
Building Blocks of ColdFusion Applications

This part describes building blocks of Coldfusion applications. These building blocks let you organize and reuse code, and include the `cfinclude` tag, user-defined functions, ColdFusion components, custom tags, and ColdFusion Extension (CFX) tags.

The following chapters are included:

Chapter 8: Creating ColdFusion Elements 161
Chapter 9: Writing and Calling User-Defined Functions 171
Chapter 10: Building and Using ColdFusion Components 201
Chapter 11: Creating and Using Custom CFML Tags 241
Chapter 12: Building Custom CFXAPI Tags 259

CHAPTER 8
Creating ColdFusion Elements

This chapter describes ColdFusion elements that you create. These elements let you to organize your code. When you create any of these elements, you write your code once and use it, without copying it, in many places. This chapter describes the elements that you can create and their features, and provides advice on selecting among the elements.

Contents

About CFML elements you create 161

Including pages with the cfinclude tag 162

About user-defined functions 164

Using ColdFusion components 165

Using custom CFML tags 166

Using CFX tags 167

Selecting among ColdFusion code reuse methods 169

About CFML elements you create

ColdFusion provides you with several techniques and elements to create sections of code that you can use multiple times in an application. Many of the elements also let you extend the built-in capabilities of ColdFusion. ColdFusion provides the following techniques and elements:

- ColdFusion pages you include using the `cfinclude` tag
- User-defined functions (UDFs)
- ColdFusion components
- Custom CFML tags
- CFX (ColdFusion Extension) tags

The following sections describe the features of each of these elements and provide guidelines for determining which to use in your application. Other chapters describe the elements in detail. The last section in this chapter includes a table to help you choose among these techniques and elements for different purposes.

ColdFusion can also use elements developed using other technologies, including the following:

- JSP tags from JSP tag libraries

  For information on using JSP tags, see Chapter 37, "Integrating J2EE and Java Elements in CFML Applications," on page 917.
- Java objects, including objects in the Java runtime environment and JavaBeans

  For information on using Java objects, see Chapter 37, "Integrating J2EE and Java Elements in CFML Applications," on page 917.
- Microsoft COM (Component Object Model) objects

  For information on using COM objects, see Chapter 38, "Integrating COM and CORBA Objects in CFML Applications," on page 945.
- CORBA (Common Object Request Broker Architecture) objects

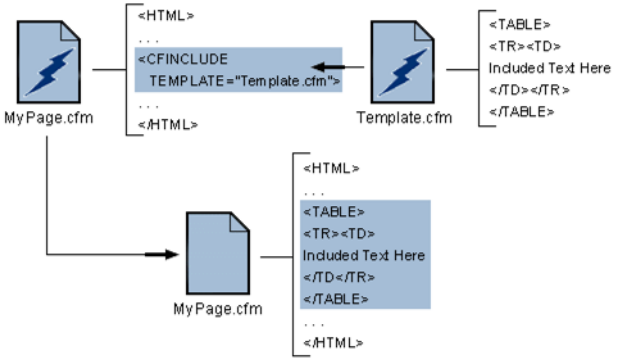
  For information on using CORBA objects, see Chapter 38, "Integrating COM and CORBA Objects in CFML Applications," on page 945.
- Web services

  For information on using web services, see Chapter 36, "Using Web Services," on page 885

Including pages with the cfinclude tag

The `cfinclude` tag adds the contents of a ColdFusion page to another ColdFusion page, as if the code on the included page were part of the page that uses the `cfinclude` tag. It lets you pursue a "write once use multiple times" strategy for ColdFusion elements that you incorporate in multiple pages. Instead of copying and maintaining the same code on multiple pages, you can store the code in one page and then refer to it in many pages. For example, the `cfinclude` tag is commonly used to put a header and footer on multiple pages. This way, if you change the header or footer design, you only change the contents of a single file.

The model of an included page is that it is part of your page; it just resides in a separate file. The `cfinclude` tag cannot pass parameters to the included page, but the included page has access to all the variables on the page that includes it. The following figure shows this model:



Using the cfinclude tag

When you use the `cfinclude` tag to include one ColdFusion page in another ColdFusion page, the page that includes another page is referred to as the *calling* page. When ColdFusion encounters a `cfinclude` tag it replaces the tag on the calling page with the output from processing the included page. The included page can also set variables in the calling page.

The following line shows a sample `cfinclude` tag:

```
<cfinclude template = "header.cfm">
```

Note: You cannot break CFML code blocks across pages. For example, if you open a `cfoutput` block in a ColdFusion page, you must close the block on the same page; you cannot include the closing portion of the block in an included page.

ColdFusion searches for included files as follows:

- The `template` attribute specifies a path relative to the directory of the calling page.
- If the template value is prefixed with a forward slash (/), ColdFusion MX searches for the included file in directories that you specify on the Mappings page of the ColdFusion MX Administrator.

Caution: A page must not include itself. Doing so causes an infinite processing loop, and you must stop the ColdFusion server to resolve the problem.

To include code in a calling page:

1. Create a ColdFusion page named header.cfm that displays your company's logo. Your page can consist of just the following lines, or it can include many lines to define an entire header:

   ```
   <img src="mylogo.gif">
   <br>
   ```

   (For this example to work, you must also put your company's logo as a GIF file in the same directory as the header.cfm file.)

2. Create a ColdFusion page with the following content:

   ```
   <html>
   <head>
     <title>Test for Include</title>
   </head>
   <body>
     <cfinclude template="header.cfm">
   </body>
   </html>
   ```

3. Save the file as includeheader.cfm and view it in a browser.

The header should appear along with the logo.

Recommended uses

Consider using the `cfinclude` tag in the following cases:

- For page headers and footers
- To divide a large page into multiple logical chunks that are easier to understand and manage
- For large "snippets" of code that are used in many places but do not require parameters or fit into the model of a function or tag

About user-defined functions

User-defined functions (UDFs) let you create application elements in a format in which you pass in arguments and get a return a value. You can define UDFs using CFScript or the `cffunction` tag. The two techniques have several differences, of which the following are the most important:

- If you use the `cffunction` tag, your function can include CFML tags.
- If you write your function using CFScript, you cannot include CFML tags.

You can use UDFs in your application pages just as you use standard ColdFusion functions. When you create a function for an algorithm or procedure that you use frequently, you can then use the function wherever you need the procedure, just as you would use a ColdFusion built-in function. For example, the following line calls the function MyFunct and passes it two arguments:

```
<cfset returnValue=MyFunct(Arg1, Arg2)>
```

You can group related functions in a ColdFusion component. For more information, see "Using ColdFusion components" on page 165.

As with custom tags, you can easily distribute UDFs to others. For example, the Common Function Library Project at www.cflib.org is an open-source collection of CFML user-defined functions.

Recommended uses

Typical uses of UDFs include, but are not limited to, the following:

- Data manipulation routines, such as a function to reverse an array
- String and date and time routines, such as a function to determine whether a string is a valid IP address
- Mathematical calculation routines, including standard trigonometric and statistical operations or calculating loan amortization
- Routines that call functions externally, for example using COM or CORBA, such as routines to determine the space available on a Windows file system drive

Consider using UDFs in the following circumstances:

- You must pass in a number of arguments, process the results, and return a value. UDFs can return complex values, including structures that contain multiple simple values.
- You want to provide logical units, such as data manipulation functions.
- Your code must be recursive.
- You distribute your code to others.

If you can create either a UDF or a custom CFML tag for a particular purpose, first consider creating a UDF because invoking it requires less system overhead than using a custom tag.

For more information

For more information on user-defined functions, see Chapter 9, "Writing and Calling User-Defined Functions," on page 171.

Using ColdFusion components

ColdFusion components (CFCs) are ColdFusion templates that contain related functions and arguments that each function accepts. The CFC contains the CFML tags necessary to define its functions and arguments and return a value. ColdFusion components are saved with a .cfc extension.

CFCs combine the power of objects with the simplicity of CFML. By packaging related functionality into a single unit, they provide an object or class shell from which functions can be called.

ColdFusion components can make their data private, so that it is available to all functions (also called methods) in the component, but not to any application that uses the component.

ColdFusion components have the following features:

- They are designed to provide related services in a single unit.
- They can provide web services and make them available over the Internet.
- They can provide ColdFusion services that Macromedia Flash clients can call directly.
- They have several features that are familiar to object-oriented programmers, including data hiding, inheritance, packages, and introspection.

Recommended uses

Consider using ColdFusion components when doing the following:

- Creating web services. (To create web services in ColdFusion, you must use components.)
- Creating services that are callable by Flash clients.
- Creating libraries of related functions, particularly if they must share data.
- Using integrated application security mechanisms based on roles and the requestor location.
- Developing code in an object-oriented manner, in which you use methods on objects and can create objects that extend the features of existing objects.

For more information

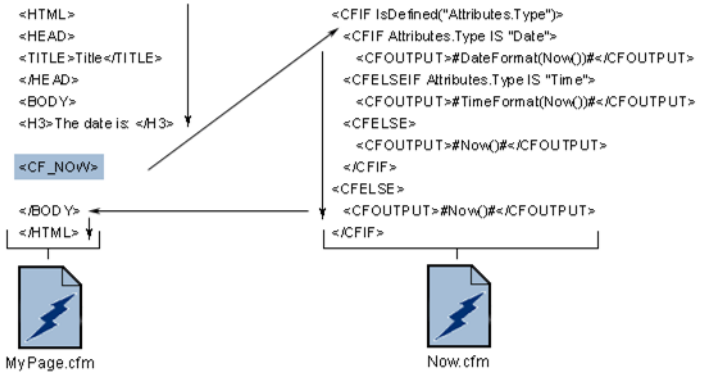
For more information on using ColdFusion components, see Chapter 10, "Building and Using ColdFusion Components," on page 201.

Using custom CFML tags

Custom tags written in CFML behave like ColdFusion tags. They can do all of the following:

- Take arguments.
- Have tag bodies with beginning and ending tags.
- Do specific processing when ColdFusion encounters the beginning tag.
- Do processing that is different from the beginning tag processing when ColdFusion encounters the ending tag.
- Have any valid ColdFusion page content in their bodies, including both ColdFusion built-in tags and custom tags (referred to as nested tags), or even JSP tags or JavaScript.
- Be called recursively; that is, a custom tag can, if designed properly, call itself in the tag body.
- Return values to the calling page in a common scope or the calling page's Variables scope, but custom tags do not return values directly, the way functions do.

Although a custom tag and a ColdFusion page that you include using the `cfinclude` tag are both ColdFusion pages, they differ in how they are processed. When a page calls a custom tag, it hands processing off to the custom tag page and waits until the custom tag page completes. When the custom tag finishes, it returns processing (and possibly data) to the calling page; the calling page can then complete its processing. The following figure shows how this works. The arrows indicate the flow of ColdFusion processing the pages.



Calling custom CFML tags

Unlike built-in tags, you can invoke custom CFML tags in the following three ways:

- Call a tag directly.
- Call a tag using the `cfmodule` tag.
- Use the `cfimport` tag to import a custom tag library directory.

To call a CFML custom tag directly, precede the file name with `cf_`, omit the .cfm extension, and put the name in angle brackets (<>). For example, use the following line to call the custom tag defined by the file mytag.cfm:

```
<cf_myTag>
```

If your tag takes a body, end it with the same tag name preceded with a forward slash (/), as follows:

```
</cf_myTag>
```

For information on using the `cfmodule` and `cfimport` tags to call custom CFML tags, see Chapter 11, "Creating and Using Custom CFML Tags," on page 241.

Recommended uses

ColdFusion custom tags let you abstract complex code and programming logic into simple units. These tags let you maintain a CFML-like design scheme for your code. You can easily distribute your custom tags and share tags with others. For example, the Macromedia ColdFusion Developer's Exchange includes a library of custom tags that perform a wide variety of often-complex jobs; see http://devex.macromedia.com/developer/gallery/index.cfm.

Consider using CFML custom tags in the following circumstances:

- You need a tag-like structure, which has a body and an end tag, with the body contents changing from invocation to invocation.
- You want to associate specific processing with the beginning tag, the ending tag, or both tags.
- To use a logical structure in which the tag body uses "child" tags or subtags. This structure is similar to the `cfform` tag, which uses subtags for the individual form fields.
- You do not need a function format in which the calling code uses a direct return value.
- Your code must be recursive.
- Your functionality is complex.
- To distribute your code in a convenient form to others.

If you can create either a UDF or a custom CFML tag for a purpose, first consider creating a UDF because invoking it requires less system overhead than using a custom tag.

For more information

For more information on custom CFML tags, see Chapter 11, "Creating and Using Custom CFML Tags," on page 241.

Using CFX tags

ColdFusion Extension (CFX) tags are custom tags that you write in Java or C++. Generally, you create a CFX tag to do something that is not possible in CFML. CFX tags also let you use existing Java or C++ code in your ColdFusion application. Unlike CFML custom tags, CFX tags cannot have bodies or ending tags.

CFX tags can return information to the calling page in a page variable or by writing text to the calling page.

CFX tags can do the following:

- Have any number of custom attributes.
- Create and manipulate ColdFusion queries.

- Dynamically generate HTML to be returned to the client.
- Set variables within the ColdFusion page from which they are called.
- Throw exceptions that result in standard ColdFusion error messages.

Calling CFX tags

To use a CFX tag, precede the class name with cfx_ and put the name in angle brackets. For example, use the following line to call the CFX tag defined by the MyCFXClass class and pass it one attribute.

```
<cfx_MyCFXClass myArgument="arg1">
```

Recommended uses

CFX tags provide one way of using C++ or Java code. However, you can also create Java classes and COM objects and access them using the `cfobject` tag. CFX tags, however, provide some built-in features that the `cfobject` tag does not have:

- CFX tags are easier to call in CFML code. You use CFX tags directly in CFML code as you would any other tag, and you can pass arguments using a standard tag format.
- ColdFusion provides predefined classes for use in your Java or C++ code that facilitate CFX tag development. These classes include support for request handling, error reporting, and query management.

You should consider using CFX tags in the following circumstances:

- You already have existing application functionality written in C++ or Java that you want to incorporate into your ColdFusion application.
- You cannot build the functionality you need using ColdFusion elements.
- You want to provide the new functionality in a tag format, as opposed to using the `cfobject` tag to import native Java or COM objects.
- You want use the Java and C++ classes provided by ColdFusion for developing your CFX code.

For more information

For more information on CFX tags, see Chapter 12, “Building Custom CFXAPI Tags,” on page 259.

Selecting among ColdFusion code reuse methods

The following table lists common reasons to employ code reuse methods and indicates the techniques to consider for each purpose. The letter **P** indicates that the method is preferred. (There can be more than one preferred method.) The letter **A** means that the method provides an alternative that might be useful in some circumstances.

This table does not include CFX tags. You use CFX tags only when you should code your functionality in C++ or Java. For more information about using CFX tags, see "Using CFX tags" on page 167.

Purpose	cfinclude tag	Custom tag	UDF	Component
Provide code, including CFML, HTML, and static text, that must be used in multiple pages.	P			
Deploy headers and footers.	P			
Include one page in another page.	P			
Divide pages into smaller units.	P			
Use variables from a calling page.	A	P	P	
Implement code that uses recursion.		P	P	P
Distribute your code to others.		P	P	P
Operate on a body of HTML or CFML text.		P		
Use subtags.		P		
Provide a computation, data manipulation, or other procedure.		A	P	
Provide a single functional element that takes any number of input values and returns a (possibly complex) result.		A	P	
Use variables, whose variable names might change from use to use.		A	P	P
Provide accessibility from Flash clients.		A	A	P
Use built-in user security features.			A	P
Encapsulate multiple related functions and properties.				P
Create web services.				P
Implement object-oriented coding methodologies.				P

CHAPTER 9
Writing and Calling User-Defined Functions

This chapter describes how to create and call user-defined functions (UDFs).

Contents

About user-defined functions 171
Creating user-defined functions 172
Calling user-defined functions 178
Working with arguments and variables in functions 178
Handling errors in UDFs 186
A user-defined function example 192
Using UDFs effectively 194

About user-defined functions

You can create your own custom functions, known as *user-defined functions*, or UDFs. You then use them in your application pages the same way you use standard ColdFusion functions. You can also organize functions you create by grouping related functions into ColdFusion components. For more information, see Chapter 10, "Building and Using ColdFusion Components," on page 201.

When you create a function for an algorithm or procedure that you use frequently, you can then use the function wherever you need the procedure. If you must change the procedure, you change only one piece of code. You can use your function anywhere that you can use a ColdFusion expression: in tag attributes, between number (#) signs in output, and in CFScript code. Typical uses of UDFs include, but are not limited to the following:

- Data manipulation routines, such as a function to reverse an array
- String and date/time routines, such as a function to determine whether a string is a valid IP address

- Mathematical calculation routines, including standard trigonometric and statistical operations or calculating loan amortization
- Routines that call functions externally, for example using COM or CORBA, including routines to determine the space available on a Windows file system drive

For information about selecting among User-defined functions, ColdFusion components, and custom tags, see Chapter 8, "Creating ColdFusion Elements," on page 161.

Note: The Common Function Library Project at www.cflib.org is an open source collection of CFML user-defined functions.

To use a user-defined function, you define the function and then call it.

Creating user-defined functions

Before you create a UDF, you must determine where you want to define it, and whether you want to use CFML or CFScript to create it.

Determining where to create a user-defined function

You can define a function in the following places:

- In a ColdFusion component. If you organize your functions in ColdFusion components, you use the functions as described in "Using ColdFusion components" on page 216.
- On the page where it is called. You can even define it below the place on the page where it is called, but this poor coding practice can result in confusing code.
- On a page that you include using a `cfinclude` tag. The `cfinclude` tag must be executed before the function gets called. For example, you can define all your application's functions on a single page and place a `cfinclude` tag at the top of pages that use the functions.
- On any page that puts the function name in a scope common with the page on which you call the function. For more information on UDF scoping, see "Specifying the scope of a function" on page 194.
- On the Application.cfc or Application.cfm page. For more information, see Chapter 13, "Designing and Optimizing a ColdFusion Application," on page 275

For recommendations on selecting where you define functions, see the sections "Using Application.cfm and function include files" on page 194 and "Specifying the scope of a function" on page 194.

About creating functions using CFScript

You use the `function` statement to define the function in CFScript. CFScript function definitions have the following features and limitations:

- The function definition syntax is familiar to anyone who uses JavaScript or most programming languages.
- CFScript is efficient for writing business logic, such as expressions and conditional operations.
- CFScript function definitions cannot include CFML tags.

The following is a CFScript definition for a function that returns a power of 2:

```
<cfscript>
function twoPower(exponent) {
    return 2^exponent;
}
</cfscript>
```

For more information on how to use CFScript to define a function, see "Defining functions in CFScript" on page 173.

Defining functions in CFScript

You define functions using CFScript in a manner similar to defining JavaScript functions. You can define multiple functions in a single CFScript block.

Note: For more information on using CFScript, see Chapter 6, "Extending ColdFusion Pages with CFML Scripting," on page 123.

CFScript function definition syntax

A CFScript function definition has the following syntax:

```
function functionName( [argName1[, argName2...]] )
  {
    CFScript Statements
  }
```

The following table describes the function variables:

Function variable	Description
functionName	The name of the function. You cannot use the name of a standard ColdFusion function or any name that starts with "cf". You cannot use the same name for two different function definitions. Function names cannot include periods.
argName1...	Names of the arguments required by the function. The number of arguments passed into the function must equal or exceed the number of arguments in the parentheses at the start of the function definition. If the calling page omits any of the required arguments, ColdFusion generates a mismatched argument count error.

The body of the function definition must be in curly braces, even if it is a empty.

The following two statements are allowed only in function definitions:

Statement	Description
`var` *variableName = expression*;	Creates and initializes a variable that is local to the function (function variable). This variable has meaning only inside the function and is not saved between calls to the function. It has precedence in the function body over any variables with the same name that exist in any other scopes. You never prefix a function variable with a scope identifier, and the name cannot include periods. The initial value of the variable is the result of evaluating the expression. The expression can be any valid ColdFusion expression, including a constant or even another UDF. All `var` statements must be at the top of the function declaration, before any other statements. You must initialize all variables when you declare them. You cannot use the same name for a function variable and an argument. Each var statement can initialize only one variable. You should use the var statement to initialize all function-only variables, including loop counters and temporary variables.
`return` *expression*;	Evaluates expression (which can be a variable), returns its value to the page that called the function, and exits the function. You can return any ColdFusion variable type.

A simple CFScript example

The following example function adds the two arguments and returns the result:

```
<cfscript>
function Sum(a,b) {
     var sum = a + b;
     return sum;
}
</cfscript>
```

In this example, a single line declares the function variable and uses an expression to set it to the value to be returned. This function can be simplified so that it does not use a function variable, as follows:

```
function MySum(a,b) {Return a + b;}
```

You must always use curly braces around the function definition body, even if it is a single statement.

About creating functions using tags

You use the `cffunction` tag to define a UDF in CFML. The `cffunction` tag syntax has the following features and limitations:

- Developers who have a background in CFML or HTML, but no scripting or programming experience will be more familiar with the syntax.
- You can include any ColdFusion tag in your function definition. Therefore, you can create a function, for example, that accesses a database.

- You can embed CFScript code inside the function definition.
- The `cffunction` tag provides attributes that enable you to easily limit the execution of the tag to authorized users or specify how the function can be accessed.

The following code uses the `cffunction` tag to define the exponentiation function:

```
<cffunction name="twoPower" output=True>
  <cfargument name="exponent">
  <cfreturn 2^exponent>
</cffunction>
```

For more information on how to use the `cffunction` tag to define a function, see "Defining functions using the cffunction tag" on page 175.

Defining functions using the cffunction tag

The `cffunction` and `cfargument` tags let you define functions in CFML without using CFScript.

Note: This chapter describes how to use the `cffunction` tag to define a function that is `not` part of a ColdFusion component. For information on ColdFusion components, see Chapter 10, "Building and Using ColdFusion Components," on page 201. For more information on the `cffunction` tag, see *CFML Reference*.

The cffunction tag function definition format

A `cffunction` tag function definition has the following format:

```
<cffunction name="functionName" [returnType="type" roles="roleList"
    access="accessType" output="Boolean"]>
  <cfargument name="argumentName" [Type="type" required="Boolean"
    default="defaultValue">]
  <!--- Function body code goes here. --->
  <cfreturn expression>
</cffunction>
```

where square brackets ([]) indicate optional arguments. You can have any number of `cfargument` tags.

The `cffunction` tag specifies the name you use when you call the function. You can optionally specify other function characteristics, as described in the following table:

Attribute	Description
name	The function name.
returnType	(Optional) The type of data that the function returns. The valid standard type names are: any, array, binary, boolean, date, guid, numeric, query, string, struct, uuid, variableName, xml, and void. If you specify any other name, ColdFusion requires the argument to be a ColdFusion component with that name. ColdFusion throws an error if you specify this attribute and the function tries to return data with a type that ColdFusion cannot automatically convert to the one you specified. For example, if the function returns the result of a numeric calculation, a returnType attribute of string or numeric is valid, but array is not.

Attribute	Description
roles	(Optional) A comma-delimited list of security roles that can invoke this method. If you omit this attribute, ColdFusion does not restrict user access to the function. If you use this attribute, the function executes only if the current user is logged in using the `cfloginuser` tag and is a member of one or more of the roles specified in the attribute. Otherwise, ColdFusion throws an unauthorized access exception. For more information on user security, see Chapter 16, "Securing Applications," on page 373.
output	(Optional) Determines how ColdFusion processes displayable output in the function body. If you do not specify this option, ColdFusion treats the body of the function as normal CFML. As a result, text and the result of any `cfoutput` tags in the function definition body are displayed each time the function executes. If you specify True or "yes", the body of the function is processed as if it were in a `cfoutput` tag. ColdFusion displays variable values and expression results if you surround the variables and expressions with number signs (#). If you specify False or "no", the function is processed as if it were in a `cfsilent` tag. The function does not display any output. The code that calls the function is responsible for displaying any function results.

You must use `cfargument` tags for required function arguments. All `cfargument` tags must precede any other CFML code in a `cffunction` tag body. Therefore, put the `cfargument` tags immediately following the `cffunction` opening tag. The `cfargument` tag takes the following attributes:

Attribute	Description
name	The argument name.
type	(Optional) The data type of the argument. The type of data that is passed to the function. The valid standard type names are any, array, binary, boolean, date, guid, numeric, query, string, struct, uuid, and variableName. If you specify any other name, ColdFusion requires the argument to be a ColdFusion component with that name. ColdFusion throws an error if you specify this attribute and the function is called with data of a type that ColdFusion cannot automatically convert to the one you specified. For example, if the argument `type` attribute is numeric, you cannot call the function with an array.
required	(Optional) A Boolean value specifying whether the argument is required, If set to True and the argument is omitted from the function call, ColdFusion throws an error. The default is False. The required attribute is not required if you specify a `default` attribute. Because you do not identify arguments when you call a function, all `cfargument` tags that specify required arguments must precede any `cfargument` tags that specify optional arguments in the `cffunction` definition.
default	(Optional) The default value for an optional argument if no argument value is passed. If you specify this attribute, ColdFusion ignores the `required` attribute.

Note: The `cfargument` tag is not required for optional arguments. This feature is useful if a function can take an indeterminate number of arguments. If you do not use the `cfargument` tag for an optional argument, reference it using its position in the Arguments scope array. For more information see "Using the Arguments scope as an array" on page 182.

Using a CFML tag in a user-defined function

The most important advantage of using the `cffunction` tag over defining a function in CFScript is that you can include CFML tags in the function. Thus, UDFs can encapsulate activities, such as database lookups, that require ColdFusion tags. Also, you can use the `cfoutput` tag to display output on the calling page with minimal coding.

Tip: To improve performance, avoid using the `cfparam` tag in ColdFusion functions. Instead, use the `cfset` tag.

The following example function looks up and returns an employee's department ID. It takes one argument, the employee ID, and looks up the corresponding department ID in the cfdocexamples Employee table:

```
<cffunction name="getDeptID" >
  <cfargument name="empID" required="true" type="numeric">
  <cfset var cfdocexamples="">
  <cfquery dataSource="cfdocexamples" name="deptID">
    SELECT Dept_ID
    FROM Employee
    WHERE Emp_ID = #empID#
  </cfquery>
  <cfreturn deptID.Dept_ID>
</cffunction>
```

Rules for function definitions

The following rules apply to functions that you define using CFScript or the `cffunction` tag:

- The function name must be unique. It must be different from any existing variable, UDF, or built-in function name, except you can use the ColdFusion advanced security function names.
- The function name must not start with the letters **cf** in any form. (For example, CF_MyFunction, cfmyFunction, and cfxMyFunction are not valid UDF names.)
- You cannot redefine or overload a function. If a function definition is active, ColdFusion generates an error if you define a second function with the same name.
- You cannot nest function definitions; that is, you cannot define one function inside another function definition.
- The function can be recursive, that is, the function definition body can call the function.
- The function does not have to return a value.

You can use tags or CFScript to create a UDF. Each technique has advantages and disadvantages.

Calling user-defined functions

You can call a function anywhere that you can use an expression, including in number signs (#) in a `cfoutput` tag, in a CFScript, or in a tag attribute value. One function can call another function, and you can use a function as an argument to another function.

You can call a UDF in two ways:

- With unnamed, positional arguments, as you would call a built-in function
- With named arguments, as you would use attributes in a tag

You can use either technique for any function. However, if you use named arguments, you must use the same argument names to call the function as you use to define the function. You cannot call a function with a mixture of named and unnamed arguments.

One example of a user-defined function is a TotalInterest function that calculates loan payments based on a principal amount, annual percentage, and loan duration in months. (For this function's definition, see "A user-defined function example" on page 192). You might call the function without argument names on a form's action page, as follows:

```
<cfoutput>
Interest: #TotalInterest(Form.Principal, Form.Percent, Form.Months)#
</cfoutput>
```

You might call the function with argument names, as follows:

```
<cfoutput>
Interest: #TotalInterest(principal=Form.Principal, annualPercent=Form.Percent,
     months=Form.Months)#
</cfoutput>
```

Working with arguments and variables in functions

Good argument naming practice

An argument's name should represent its use. For example, the following code is unlikely to result in confusion:

```
<cfscript>
  function SumN(Addend1,Addend2)
  { return Addend1 + Addend2; }
</cfscript>
<cfset x = 10>
<cfset y = 12>
<cfoutput>#SumN(x,y)#</cfoutput>
```

The following, similar code is more likely to result in programming errors:

```
<cfscript>
  function SumN(x,y)
  { return x + y; }
</cfscript>
<cfset x = 10>
<cfset y = 12>
<cfoutput>#SumN(x,y)#<cfoutput>
```

Passing arguments

ColdFusion passes the following data types to the function by value:

- Integers
- Real numbers
- Strings (including lists)
- Date-time objects
- Arrays

As a result, any changes that you make in the function to these arguments do not affect the variable that was used to call the function, even if the calling code is on the same ColdFusion page as the function definition.

ColdFusion passes queries, structures, and external objects such as COM objects into the function by reference. As a result, any changes to these arguments in the function also change the value of the variable in the calling code.

For an example of the effects of passing arguments, see "Passing complex data" on page 179.

Passing complex data

Structures, queries, and complex objects such as COM objects are passed to UDFs by reference, so the function uses the same copy of the data as the caller. Arrays are passed to user-defined functions by value, so the function gets a new copy of the array data and the array in the calling page is unchanged by the function. As a result, you must handle arrays differently from all other complex data types.

Passing structures, queries, and objects

For your function to modify the caller's copy of a structure, query, or object, you must pass the variable as an argument. Because the function gets a reference to the caller's structure, the caller variable reflects all changes in the function. You do not have to return the structure to the caller. After the function returns, the calling page accesses the changed data by using the structure variable that it passed to the function.

If you do not want a function to modify the caller's copy of a structure, query, or object, use the `Duplicate` function to make a copy and pass the copy to the function.

Passing arrays

If you want your function to modify the caller's copy of the array, the simplest solution is to pass the array to the function and return the changed array to the caller in the function `return` statement. In the caller, use the same variable name in the function argument and return variable.

The following example shows how to directly pass and return arrays. In this example, the `doubleOneDArray` function doubles the value of each element in a one-dimensional array.

```
<cfscript>
//Initialize some variables
//This creates a simple array.
a=ArrayNew(1);
```

```
a[1]=2;
a[2]=22;
//Define the function.
function doubleOneDArray(OneDArray) {
  var i = 0;
  for ( i = 1; i LE arrayLen(OneDArray); i = i + 1)
    { OneDArray[i] = OneDArray[i] * 2; }
  return OneDArray;
}
//Call the function.
a = doubleOneDArray(a);
</cfscript>
<cfdump var="#a#">
```

This solution is simple, but it is not always optimal:

- This technique requires ColdFusion to copy the entire array twice, once when you call the function and once when the function returns. This is inefficient for large arrays and can reduce performance, particularly if the function is called frequently.
- You can use the return value for other purposes, such as a status variable.

If you do not use the `return` statement to return the array to the caller, you can pass the array as an element in a structure and change the array values inside the structure. Then the calling page can access the changed data by using the structure variable it passed to the UDF.

The following code shows how to rewrite the previous example using an array in a structure. It returns True as a status indicator to the calling page and uses the structure to pass the array data back to the calling page.

```
<cfscript>
//Initialize some variables.
//This creates an simple array as an element in a structure.
arrayStruct=StructNew();
arrayStruct.Array=ArrayNew(1);
arrayStruct.Array[1]=2;
arrayStruct.Array[2]=22;
//Define the function.
function doubleOneDArrayS(OneDArrayStruct) {
  var i = 0;
  for ( i = 1; i LE arrayLen(OneDArrayStruct.Array); i = i + 1)
    { OneDArrayStruct.Array[i] = OneDArrayStruct.Array[i] * 2; }
  return True;
}
//Call the function.
Status = doubleOneDArrayS(arrayStruct);
WriteOutput("Status: " & Status);
</cfscript>
</br>
<cfdump var="#arrayStruct#">
```

You must use the same structure element name for the array (in this case Array) in the calling page and the function.

About the Arguments scope

All function arguments exist in their own scope, the Arguments scope.

The Arguments scope exists for the life of a function call. When the function returns, the scope and its variables are destroyed.

However, destroying the Argument scope does not destroy variables, such as structures or query objects, that ColdFusion passes to the function by reference. The variables on the calling page that you use as function arguments continue to exist; if the function changes the argument value, the variable in the calling page reflects the changed value.

The Arguments scope is special, in that you can treat the scope as either an array *or* a structure. This dual nature of the Arguments scope is useful because it makes it easy to use arguments in any of the following circumstances:

- You define the function using CFScript.
- You define the function using the `cffunction` tag.
- You pass arguments using argument name=value format.
- You pass arguments as values only.
- The function takes optional, undeclared arguments.

The following sections describe the general rules for using the Arguments scope as an array and a structure. For more information on using the Arguments scope in functions defined using CFScript, see "Using the Arguments scope in CFScript" on page 184. For more information on using the Arguments scope in functions defined using the `cffunction` tag, see "Using the Arguments scope in cffunction definitions" on page 184.

The contents of the Arguments scope

The following rules apply to the Arguments scope and its contents:

- The scope contains all the arguments passed into a function.
- If you use `cffunction` to define the function, the scope always contains an entry "slot" for each declared argument, even if you do not pass the argument to the function when you call it. If you do not pass a declared (optional) argument, the scope entry for that argument is empty.

  When you call a function that you defined using CFScript, you must pass the function a value for each argument declared in the function definition. Therefore, the Arguments scope for a CFScript call does not have empty slots.

The following example shows these rules. Assume that you have a function declared, as follows:

```
<cffunction name="TestFunction">
  <cfargument name="Arg1" >
  <cfargument name="Arg2">
</cffunction>
```

You can call this function with a single argument, as in the following line:

```
<cfset TestFunction(1)>
```

The resulting Arguments scope looks like the following:

As an array		As a structure	
Entry	**Value**	**Entry**	**Value**
1	1	Arg1	1
2	undefined	Arg2	undefined

In this example, the following functions return the value 2 because there are two defined arguments:

```
ArrayLen(Arguments)
StructCount(Arguments)
```

However, the following tests return the value False, because the contents of the second element in the Arguments scope is undefined.

```
Isdefined("Arguments.Arg2")
testArg2 = Arguments[2]>
Isdefined("testArg2")
```

Note: The `IsDefined` function does not test the existence of array elements. Instead, put any code that might try to access an undefined array element in a try block and use a catch block to handle exceptions that arise if elements do not exist.

Using the Arguments scope as an array

The following rules apply to referencing Arguments scope as an array:

- If you call the function using unnamed arguments, the array index is the position of the argument in the function call.
- If you use names to pass the arguments, the array indexes correspond to the order in which the arguments are declared in the function definition.
- If you use names to pass arguments, and do not pass all the arguments defined in the function, the Arguments array has an empty entry at the index corresponding to the argument that was not passed. This rule applies only to functions created using the `cffunction` tag.
- If you use a name to pass an optional argument that is not declared in the function definition, the array index of the argument is the sum of the following:
  a The number of arguments defined with names in the function.
  b The position of the optional argument among the arguments passed in that do not have names defined in the function.

  However, using argument names in this manner is not good programming practice because you cannot ensure that you always use the same optional argument names when calling the function.

To demonstrate these rules, define a simple function that displays the contents of its Arguments array and call the function with various argument combinations, as shown in the following example:

```
<cffunction name="TestFunction" >
  <cfargument name="Arg1">
```

```
  <cfargument name="Arg2">
  <cfloop index="i" from="1" to="#ArrayLen(Arguments)#">
    <cfoutput>Argument #i#: #Arguments[i]#<br></cfoutput>
  </cfloop>
</cffunction>

<strong>One Unnamed argument</strong><br>
<cfset TestFunction(1)>
<strong>Two Unnamed arguments</strong><br>
<cfset TestFunction(1, 2)>
<strong>Three Unnamed arguments</strong><br>
<cfset TestFunction(1, 2, 3)>
<strong>Arg1:</strong><br>
<cfset TestFunction(Arg1=8)>
<strong>Arg2:</strong><br>
<cfset TestFunction(Arg2=9)>
<strong>Arg1=8, Arg2=9:</strong><br>
<cfset TestFunction(Arg1=8, Arg2=9)>
<strong>Arg2=6, Arg1=7</strong><br>
<cfset TestFunction(Arg2=6, Arg1=7)>
<strong>Arg1=8, Arg2=9, Arg3=10:</strong><br>
<cfset TestFunction(Arg1=8, Arg2=9, Arg3=10)>
<strong>Arg2=6, Arg3=99, Arg1=7</strong><br>
<cfset TestFunction(Arg2=6, Arg3=99, Arg1=7)>
```

Note: Although you can use the Arguments scope as an array, the `IsArray(Arguments)` function always returns false and the `cfdump` tag displays the scope as a structure.

Using the Arguments scope as a structure

The following rule applies when referencing Arguments scope as a structure:

- Use the argument names as structure keys. For example, if your function definition includes a Principal argument, refer to the argument as Arguments.Principal.

The following rules are also true, but *avoid writing code that uses them*. To ensure program clarity, only use the Arguments structure for arguments that you name in the function definition. Use the Arguments scope as an array for optional arguments that you do not declare in the function definition.

- If the function can take unnamed optional arguments, use array notation to reference the unnamed arguments. For example, if the function declaration includes two named arguments and you call the function with three arguments, refer to the third argument as Arguments[3]. To determine if an unnamed optional argument exists, use the `StructKeyExists` function; for example, `structKeyExists(Arguments,"3")`.
- If you do not name an optional argument in the function definition, but do use a name for it in the function call, use the name specified in the function call For example, if you have an unnamed optional argument and call the function using the name myOptArg for the argument, you can refer to the argument as Arguments.myOptArg in the function body. This usage, however, is poor programming practice, as it makes the function definition contents depend on variable names in the code that calls the function.

Using the Arguments scope in CFScript

A function can have optional arguments that you do not have to specify when you call the function. To determine the number of arguments passed to the function, use the following function:

```
ArrayLen(Arguments)
```

When you define a function using CFScript, the function must use the Arguments scope to retrieve the optional arguments. For example, the following SumN function adds two or more numbers together. It requires two arguments and supports any number of additional optional arguments. You can refer to the first two, required, arguments as `Arg1` and `Arg2` or as `Arguments[1]` and `Arguments[2]`. You must refer to the third, fourth, and any additional optional arguments as `Arguments[3]`, `Arguments[4]`, and so on.

```
function SumN(Arg1,Arg2) {
  var arg_count = ArrayLen(Arguments);
  var sum = 0;
  var i = 0;
  for( i = 1 ; i LTE arg_count; i = i + 1 )
  {
    sum = sum + Arguments[i];
  }
  return sum;
}
```

With this function, any of the following function calls are valid:

```
SumN(Value1, Value2)
SumN(Value1, Value2, Value3)
SumN(Value1, Value2, Value3, Value4)
```

and so on.

The code never uses the Arg1 and Arg2 argument variables directly, because their values are always the first two elements in the Arguments array and it is simpler to step through the array. Specifying Arg1 and Arg2 in the function definition ensures that ColdFusion generates an error if you pass the function one or no arguments.

Note: Avoid referring to a required argument in the body of a function by both the argument name and its place in the Arguments scope array or structure, as this can be confusing and makes it easier to introduce errors.

For more information on the Arguments scope, see "About the Arguments scope" on page 181.

Using the Arguments scope in cffunction definitions

When you define a function using the `cffunction` tag, you generally refer to the arguments directly by name if all arguments are named in the `cfargument` tags. If you do use the Arguments scope identifier, follow the rules listed in "About the Arguments scope" on page 181.

Function-only variables

In addition to the Arguments scope, each function can have a number of variables that exist only inside the function, and are not saved between times the function gets called. As soon as the function exits, all the variables in this scope are removed.

In CFScript, you create function-only variables with the `var` statement. Unlike other variables, you *never* prefix function-only variables with a scope name.

Using function-only variables

Make sure to use the `var` statement in CFScript UDFs to declare all function-specific variables, such as loop indexes and temporary variables that are required only for the duration of the function call. Doing this ensures that these variables are available inside the function only, and makes sure that the variable names do not conflict with the names of variables in other scopes. If the calling page has variables of the same name, the two variables are independent and do not affect each other.

For example, if a ColdFusion page has a `cfloop` tag with an index variable i, and the tag body calls a CFScript UDF that also has a loop with a function-only index variable i, the UDF does not change the value of the calling page loop index, and the calling page does not change the UDF index. So you can safely call the function inside the `cfloop` tag body.

In general, use the `var` statement to declare all UDF variables, other than the function arguments or shared-scope variables, that you use only inside CFScript functions. Use another scope, however, if the value of the variable must persist between function calls; for example, for a counter that the function increments each time it is called.

Referencing caller variables

A function can use and change any variable that is available in the calling page, including variables in the caller's Variables (local) scope, as if the function was part of the calling page. For example, if you know that the calling page has a local variable called Customer_name (and there is no function scope variable named Customer_name) the function can read and change the variable by referring to it as Customer_name or (using better coding practice) Variables.Customer_name. Similarly, you can create a local variable inside a function and then refer to it anywhere in the calling page *after* the function call. You cannot refer to the variable before you call the function.

However, you should generally avoid using the caller's variables directly inside a function. Using the caller's variables creates a dependency on the caller. You must always ensure that the code outside the function uses the same variable names as the function. This can become difficult if you call the function from many pages.

You can avoid these problems by using only the function arguments and the return value to pass data between the caller and the function. Do not reference calling page variables directly in the function. As a result, you can use the function anywhere in an application (or even in multiple applications), without concern for the calling code's variables.

As with many programming practice, there are valid exceptions to this recommendation. For example you might do any of the following:

- Use a shared scope variable, such as an Application or Session scope counter variable.
- Use the Request scope to store variables used in the function, as shown in "Using the Request scope for static variables and constants" on page 196.).
- Create context-specific functions that work directly with caller data if you *always* synchronize variable names.

Note: If your function must directly change a simple variable in the caller (one that is not passed to the function by reference), you can place the variable inside a structure argument.

Using arguments

Function arguments can have the same names, but different values, as variables in the caller. Avoid such uses for clarity, however.

The following rules apply to argument persistence:

- Because simple variable and array arguments are passed by value, their names and values exist only while the function executes.
- Because structures, queries, and objects such as COM objects are passed by reference, the argument *name* exists only while the function executes, but the underlying *data* persists after the function returns and can be accessed by using the caller's variable name. The caller's variable name and the argument name can, and should, be different.

Note: If a function must use a variable from another scope that has the same name as a function-only variable, prefix the external variable with its scope identifier, such as Variables or Form. (However, remember that using variables from other scopes directly in your code is often poor practice.)

Handling errors in UDFs

This section discusses the following topics:

- Displaying error messages directly in the function
- Returning function status information to the calling page
- Using `try`/`catch` or `cftry`/`cfcatch` blocks and the `cfthrow` and `cfrethrow` tags to handle and generate exceptions

The technique you use depends on the circumstances of your function and application and on your preferred programming style. However, most functions should use the second or third technique, or a combination of the two. The following sections discuss the uses, advantages, and disadvantages of each technique, and provide examples of their use.

Displaying error messages

Your function can test for errors and use the `WriteOutput` function to display an error message directly to the user. This method is particularly useful for providing immediate feedback to users for simple input errors. You can use it independently or in conjunction with either of the other two error-handling methods.

For example, the following variation on a "Hello world" function displays an error message if you do not enter a name in the form:

```
<cfform method="POST" action="#CGI.script_name#">
  <p>Enter your Name: 
  <input name="name" type="text" hspace="30" maxlength="30">
  <input type="Submit" name="submit" value="OK">
</cfform>
<cfscript>
  function HelloFriend(Name) {
    if (Name is "") WriteOutput("You forgot your name!");
    else WriteOutput("Hello " & name &"!");
    return "";
  }
  if (IsDefined("Form.submit")) HelloFriend(Form.name);
</cfscript>
```

Reviewing the code

The following table describes the code:

Code	Description

`<cfform method="POST" action="#CGI.script_name#">`
`<p>Enter your Name: `
`<input name="name" type="text" hspace="30" maxlength="30">`
`<input type="Submit" name="submit" value="OK">`
`</cfform>`
	Creates a simple form requesting you to enter your name.
	Uses the script_name CGI variable to post to this page without specifying a URL.
	If you do not enter a name, the form posts an empty string as the name field.

`<cfscript>`
`function HelloFriend(Name) {`
`if (Name is "") WriteOutput("You forgot your name!");`
`else WriteOutput("Hello " & name &"!");`
`return "";`
`}`
`if (IsDefined("Form.submit")) HelloFriend(Form.name);`
`</cfscript>`
	Defines a function to display "Hello *name*!" First, checks whether the argument is an empty string. If so, displays an error message.
	Otherwise displays the hello message.
	Returns the empty string. (The caller does not use the return value). It is not necessary to use curly braces around the if or else statement bodies because they are single statements.
	If this page has been called by submitting the form, calls the HelloFriend function. Otherwise, the page just displays the form.

Providing status information

In some cases, such as those where the function cannot provide a corrective action, the function cannot, or should not, handle the error directly. In these cases, your function can return information to the calling page. The calling page must handle the error information and act appropriately.

Consider the following mechanisms for providing status information:

- Use the return value to indicate the function status only. The return value can be a Boolean success/failure indicator. The return value can also be a status code, for example where 1 indicates success, and various failure types are assigned known numbers. With this method, the function must set a variable in the caller to the value of a successful result.
- Set a status variable that is available to the caller (not the return variable) to indicate success or failure and any information about the failure. With this method, the function can return the result directly to the caller. In this method, the function should use only the return value and structure arguments to pass the status back to the caller.

Each of these methods can have variants, and each has advantages and disadvantages. The technique that you use should depend on the type of function, the application in which you use it, and your coding style.

The following example, which modifies the function used in "A user-defined function example" on page 192, uses one version of the status variable method. It provides two forms of error information:

- It returns -1, instead of an interest value, if it encounters an error. This value can serve as an error indicator because you never pay negative interest on a loan.
- It also writes an error message to a structure that contains an error description variable. Because the message is in a structure, it is available to both the calling page and the function.

The TotalInterest function

After changes to handle errors, the `TotalInterest` function looks like the following. Code that is changed from the example in "A user-defined function example" on page 192 is in bold.

```
<cfscript>
function TotalInterest(principal, annualPercent, months, status) {
  Var years = 0;
  Var interestRate = 0;
  Var totalInterest = 0;
  principal = trim(principal);
  principal = REReplace(principal,"[\$,]","","ALL");
  annualPercent = Replace(annualPercent,"%","","ALL");
  if ((principal LE 0) OR (annualPercent LE 0) OR (months LE 0)) {
    Status.errorMsg = "All values must be greater than 0";
    Return -1;
  }
  interestRate = annualPercent / 100;
  years = months / 12;
  totalInterest = principal*(((1+ interestRate)^years)-1);
  Return DollarFormat(totalInterest);
}
</cfscript>
```

Reviewing the code

The following table describes the code that has been changed or added to the previous version of this example. For a description of the initial code, see "A user-defined function example" on page 192.

Code	Description
`function TotalInterest(principal, annualPercent, months, status)`	The function now takes an additional argument, a status structure. Uses a structure for the status variable so that changes that the function makes affect the status structure in the caller.
`if ((principal LE 0) OR (annualPercent LE 0) OR (months LE 0)) { Status.errorMsg = "All values must be greater than 0"; Return -1; }`	Checks to make sure the principal, percent rate, and duration are all greater than zero. If any is not, sets the errorMsg key (the only key) in the Status structure to a descriptive string. Also, returns -1 to the caller and exits the function without processing further.

Calling the function

The code that calls the function now looks like the following. Code that is changed from the example in "A user-defined function example" on page 192 is in bold.

```
<cfset status = StructNew()>
<cfset myInterest = TotalInterest(Form.Principal,
  Form.AnnualPercent,Form.Months, status)>
<cfif myInterest EQ -1>
  <cfoutput>
    ERROR: #status.errorMsg#<br>
  </cfoutput>
<cfelse>
  <cfoutput>
    Loan amount: #Form.Principal#<br>
    Annual percentage rate:
      #Form.AnnualPercent#<br>
    Loan duration: #Form.Months# months<br>
    TOTAL INTEREST: #myInterest#<br>
  </cfoutput>
</cfif>
```

Reviewing the code

The following table describes the code that has been changed or added:

Code	Description
`<cfset status = StructNew()>`	Creates a structure to hold the function status.
`<cfset myInterest = TotalInterest (Form.Principal, Form.AnnualPercent, Form.Months, status)>`	Calls the function. This time, the function requires four arguments, including the status variable.

Code	Description
<cfif myInterest EQ -1> <cfoutput> ERROR: #status.errorMsg# </cfoutput>	If the function returns -1, there must be an error. Displays the message that the function placed in the status.errorMsg structure key.

<cfelse>
  <cfoutput>
  Loan amount: #Form.Principal#
 	
  Annual percentage rate:
  #Form.AnnualPercent#
 	
  Loan duration: #Form.Months#
  months
 	
  TOTAL INTEREST: #myInterst#
 	
  </cfoutput>
</cfif>
	If the function does not return -1, it returns an interest value. Displays the input values and the function return value.

Using exceptions

UDFs written in CFScript can handle exceptions using the `try` and `catch` statements. UDFs written using the `cffunction` tag can use the `cftry`, `cfcatch`, `cfthrow`, and `cfrethrow` tags. Using exceptions corresponds to the way many functions in other programming languages handle errors, and can be an effective way to handle errors. In particular, it separates the functional code from the error-handling code, and it can be more efficient than other methods at runtime, because it does not require testing and branching.

Exceptions in UDFs have the following two dimensions:

- Handling exceptions generated by running the UDF code
- Generating exceptions when the UDF identifies invalid data or other conditions that would cause errors if processing continued

Handling exceptions in UDFs

A UDF should use try/catch blocks to handle exceptions in the same conditions that any other ColdFusion application uses try/catch blocks. These are typically circumstances where the function uses an external resource, such as a Java, COM, or CORBA object, a database, or a file. When possible, your application should prevent, rather than catch, exceptions caused by invalid application data. For example, the application should prevent users from entering a zero value for a form field that is used to divide another number, rather than handling exceptions generated by dividing by zero.

When ColdFusion catches an exception, the function can use any of the following methods to handle the exception:

- If the error is recoverable (for example, if the problem is a database timeout where a retry might resolve the issue), try to recover from the problem.
- Display a message, as described in “Displaying error messages” on page 186.
- Return an error status, as described in “Providing status information” on page 187.

- If the UDF is defined using the `cffunction` tag, throw a custom exception, or rethrow the exception so that it will be caught by the calling ColdFusion page. For more information on throwing and rethrowing exceptions, see "Handling runtime exceptions with ColdFusion tags" on page 322.

Generating exceptions in UDFs

If you define your function using the `cffunction` tag, you can use the `cfthrow` and `cfrethrow` tags to throw errors to the page that called the function. You can use this technique whenever your UDF identifies an error, instead of displaying a message or returning an error status. For example, the following code rewrites the example from "Providing status information" on page 187 to use the `cffunction` tag and CFML, and to throw and handle an exception if any of the form values are not positive numbers.

The lines that identify invalid data and throw the exception are in bold. The remaining lines are equivalent to the CFScript code in the previous example. However, the code that removes unwanted characters must precede the error checking code.

```
<cffunction name="TotalInterest">
  <cfargument name="principal" required="Yes">
  <cfargument name="annualPercent" required="Yes">
  <cfargument name="months" required="Yes">
  <cfset var years = 0>
  <cfset var interestRate = 0>
  <cfset var totalInterest = 0>

  <cfset principal = trim(principal)>
  <cfset principal = REReplace(principal,"[\$,]","","ALL")>
  <cfset annualPercent = Replace(annualPercent,"%","","ALL")>

  <cfif ((principal LE 0) OR (annualPercent LE 0) OR (months LE 0))>
    <cfthrow type="InvalidData" message="All values must be greater than 0.">
  </cfif>

  <cfset interestRate = annualPercent / 100>
  <cfset years = months / 12>
  <cfset totalInterest = principal*
      (((1+ interestRate)^years)-1)>
  <cfreturn DollarFormat(totalInterest)>
</cffunction>
```

The code that calls the function and handles the exception looks like the following. The changed lines are in bold.

```
<cftry>
  <cfset status = StructNew()>
  <cfset myInterest = TotalInterest(Form.Principal, Form.AnnualPercent,
    Form.Months, status)>
  <cfoutput>
    Loan amount: #Form.Principal#<br>
    Annual percentage rate: #Form.AnnualPercent#<br>
    Loan duration: #Form.Months# months<br>
    TOTAL INTEREST: #myInterest#<br>
  </cfoutput>
```

```
<cfcatch type="InvalidData">
  <cfoutput>
    #cfcatch.message#<br>
  </cfoutput>
</cfcatch>
</cftry>
```

A user-defined function example

The following simple function takes a principal amount, an annual percentage rate, and a loan duration in months and returns the total amount of interest to be paid over the period. You can optionally use the percent sign for the percentage rate, and include the dollar sign and comma separators for the principal amount.

You could use the `TotalInterest` function in a `cfoutput` tag of a form's action page, as follows:

```
<cfoutput>
  Loan amount: #Form.Principal#<br>
  Annual percentage rate: #Form.AnnualPercent#<br>
  Loan duration: #Form.Months# months<br>
  TOTAL INTEREST: #TotalInterest(Form.Principal, Form.AnnualPercent,
    Form.Months)#<br>
</cfoutput>
```

Defining the function using CFScript

```
<cfscript>
function TotalInterest(principal, annualPercent, months) {
  Var years = 0;
  Var interestRate = 0;
  Var totalInterest = 0;
  principal = trim(principal);
  principal = REReplace(principal,"[\$,]","","ALL");
  annualPercent = Replace(annualPercent,"%","","ALL");
  interestRate = annualPercent / 100;
  years = months / 12;
  totalInterest = principal*(((1+ interestRate)^years)-1);
  Return DollarFormat(totalInterest);
}
</cfscript>
```

Reviewing the code

The following table describes the code:

Code	Description

`function TotalInterest(principal,`
`  annualPercent,`
`  months) {`
	Starts the `TotalInterest` function definition. Requires three variables: the principal amount, the annual percentage rate, and the loan duration in months.

`Var years = 0;`
`Var interestRate = 0;`
`Var totalInterest = 0;`
	Declares intermediate variables used in the function and initializes them to 0. All `var` statements must precede the rest of the function code.

`principal = trim(principal);`
`principal =`
`  REReplace(principal,"[\$,]","","ALL");`
`annualPercent =`
`  Replace(annualPercent,"%","","ALL");`
`interestRate = annualPercent / 100;`
`years = months / 12;`
	Removes any leading or trailing spaces from the principal argument. Removes any dollar sign ($) and comma (,) characters from the principal argument to get a numeric value.
	Removes any percent (%) character from the `annualPercent` argument to get a numeric value, then divides the percentage value by 100 to get the interest rate.
	Converts the loan from months to years.

`  totalInterest = principal*(((1+`
`  interestRate)^years)-1);`
`  Return DollarFormat(totalInterest);`
`}`
	Calculates the total amount of interest due. It is possible to calculate the value in the `Return` statement, but this example uses an intermediate totalInterest variable to make the code easier to read. Returns the result formatted as a US currency string.
	Ends the function definition.

Defining the function using the cffunction tag

The following code replaces CFScript statements with their equivalent CFML tags.

```
<cffunction name="TotalInterest">
  <cfargument name="principal" required="Yes">
  <cfargument name="annualPercent" required="Yes">
  <cfargument name="months" required="Yes">
  <cfset var years = 0>
  <cfset var interestRate = 0>
  <cfset var totalInterest = 0>
  <cfset principal = trim(principal)>
  <cfset principal = REReplace(principal,"[\$,]","","ALL")>
  <cfset annualPercent = Replace(annualPercent,"%","","ALL")>
  <cfset interestRate = annualPercent / 100>
  <cfset years = months / 12>
  <cfset totalInterest = principal*
       (((1+ interestRate)^years)-1)>
  <cfreturn DollarFormat(totalInterest)>
</cffunction>
```

Using UDFs effectively

This section provides information that will help you use user-defined functions more effectively.

Using functions in ColdFusion component

In many cases, the most effective use of UDFs is within a CFC. For more information on CFCs, see Chapter 10, "Building and Using ColdFusion Components," on page 201.

Using Application.cfm and function include files

Consider the following techniques for making your functions available to your ColdFusion pages:

- If you consistently call a small number of UDFs, consider putting their definitions on the Application.cfm page.
- If you call UDFs in only a few of your application pages, do not include their definitions in Application.cfm.
- If you use many UDFs, put their definitions on one or more ColdFusion pages that contain only UDFs. You can include the UDF definition page in any page that calls the UDFs.

The next section describes other techniques for making UDFs available to your ColdFusion pages.

Specifying the scope of a function

User-defined function names are essentially ColdFusion variables. ColdFusion variables are names for data. Function names are names (references) for segments of CFML code. Therefore, like variables, functions belong to scopes.

About functions and scopes

Like ColdFusion variables, UDFs exist in a scope:

- When you define a UDF, ColdFusion puts it in the Variables scope.
- You can assign a UDF to a scope the same way you assign a variable to a scope, by assigning the function to a name in the new scope. For example, the following line assigns the MyFunc UDF to the Request scope:

  ```
  <cfset Request.MyFunc = Variables.MyFunc>
  ```

  You can now use the function from any page in the Request scope by calling Request.MyFunc.

Selecting a function scope

The following table describes the advantages and disadvantages of scopes that you might considering using for your functions:

Scope	Considerations
Application	Makes the function available across all invocations of the application. Access to UDFs in Application scope is multithreaded and you can execute multiple copies of the UDF at one time.
Request	Makes the function available for the life of the current HTTP request, including in all custom tags and nested custom tags. This scope is useful if a function is used in a page and in the custom tags it calls, or in nested custom tags.
Server	Makes the function available to all pages on a single server. In most cases, this scope is not a good choice because in clustered systems, it only makes the function available on a single server, and all code that uses the function must be inside a `cflock` block.
Session	Makes the function available to all pages during the current user session. This scope has no significant advantages over the Application scope.

Using the Request scope

You can effectively manage functions that are used in application pages and custom tags by doing the following:

1. Define the functions on a function definitions page.
2. On the functions page, assign the functions to the request scope.
3. Use a `cfinclude` tag to include the function definition page on the application page, but do not include it on any custom tag pages.
4. Always call the functions using the request scope.

This way you only need to include the functions once per request and they are available throughout the life of the request. For example, create a myFuncs.cfm page that defines your functions and assigns them to the Request scope using syntax such as the following:

```
function MyFunc1(Argument1, Argument2)
{ Function definition goes here }
Request.MyFunc1 = MyFunc1
```

The application page includes the myFuncs.cfm page:

```
<cfinclude template="myfuncs.cfm">
```

The application page and all custom tags (and nested custom tags) call the functions as follows:

```
Request.MyFunc1(Value1, Value2)
```

Using the Request scope for static variables and constants

This section describes how to partially break the rule described in the section "Referencing caller variables" on page 185. Here, the function defines variables in the Request scope. However, it is a specific solution to a specific issue, where the following circumstances exist:

- Your function initializes a large number of variables.
- The variables have either of the following characteristics:
  - They must be *static*: they are used only in the function, the function can change their values, and their values must persist from one invocation of the function to the next.
  - They are *named constants*; that is the variable value never changes.
- Your application page (and any custom tags) calls the function multiple times.
- You can assure that the variable names are used only by the function.

In these circumstances, you can improve efficiency and save processing time by defining your function's variables in the Request scope, rather than the Function scope. The function tests for the Request scope variables and initializes them if they do not exist. In subsequent calls, the variables exist and the function does not reset them.

The `NumberAsString` function, written by Ben Forta and available from www.cflib.org, takes advantage of this technique.

Using function names as function arguments

Because function names are ColdFusion variables, you can pass a function's name as an argument to another function. This technique allows a function to use another function as a component. For example, a calling page can call a calculation function, and pass it the name of a function that does some subroutine of the overall function.

This way, the calling page could use a single function for different specific calculations, such as calculating different forms of interest. The initial function provides the framework, while the function whose name is passed to it can implement a specific algorithm that is required by the calling page.

The following simple example shows this use. The `binop` function is a generalized function that takes the name of a function that performs a specific binary operation and two operands. The `binop` function simply calls the specified function and passes it the operands. This code defines a single operation function, the sum function. A more complete implementation would define multiple binary operations.

```
<cfscript>
function binop(operation, operand1, operand2)
{ return (operation(operand1, operand2)); }
function sum(addend1, addend2)
{ return addend1 + addend2;}
x = binop(sum, 3, 5);
writeoutput(x);
</cfscript>
```

Handling query results using UDFs

When you call a UDF in the body of a tag that has a `query` attribute, such as a `cfloop query=...` tag, any function argument that is a query column name passes a single element of the column, not the entire column. Therefore, the function must manipulate a single query element.

For example, the following code defines a function to combine a single first name and last name to make a full name. It queries the cfdocexamples database to get the first and last names of all employees, and then it uses a `cfoutput` tag to loop through the query and call the function on each row in the query.

```
<cfscript>
function FullName(aFirstName, aLastName)
 { return aFirstName & " " & aLastName; }
</cfscript>

<cfquery name="GetEmployees" datasource="cfdocexamples">
  SELECT FirstName, LastName
  FROM Employee
</cfquery>

<cfoutput query="GetEmployees">
#FullName(FirstName, LastName)#<br>
</cfoutput>
```

You generally use functions that manipulate many rows of a query *outside* tags that loop over queries. Pass the query to the function and loop over it inside the function. For example, the following function changes text in a query column to uppercase. It takes a query name as an argument.

```
function UCaseColumn(myquery, colName) {
  var currentRow = 1;
  for (; currentRow lte myquery.RecordCount;
    currentRow = currentRow + 1)
  {
    myquery[colName][currentRow] =
      UCase(myquery[colName][currentRow]);
  }
  Return "";
}
```

The following code uses a script that calls the `UCaseColumn` function to convert all the last names in the GetEmployees query to uppercase. It then uses `cfoutput` to loop over the query and display the contents of the column.

```
<cfscript>
  UCaseColumn(GetEmployees, "LastName");
</cfscript>
<cfoutput query="GetEmployees">
  #LastName#<br>
</cfoutput>
```

Identifying and checking for UDFs

You can use the `IsCustomFunction` function to determine whether a name represents a UDF. The `IsCustomFunction` function generates an error if its argument does not exist. As a result, you must ensure that the name exists before calling the function, for example, by calling the `IsDefined` function. The following code shows this use:

```
<cfscript>
if(IsDefined("MyFunc"))
  if(IsCustomFunction(MyFunc))
    WriteOutput("MyFunc is a user-defined function");
  else
    WriteOutput("Myfunc is defined but is NOT a user-defined function");
else
  WriteOutput("MyFunc is not defined");
</cfscript>
```

You do *not* surround the argument to `IsCustomFunction` in quotation marks, so you can use this function to determine if function arguments are themselves functions.

Using the Evaluate function

If your user-defined function uses the `Evaluate` function on arguments that contain strings, you must make sure that all variable names you use as arguments include the scope identifier. Doing so avoids conflicts with function-only variables.

The following example returns the result of evaluating its argument. It produces the expected results, the value of the argument, if you pass the argument using its fully scoped name, Variables.myname. However, the function returns the value of the function local variable if you pass the argument as myname, without the Variables scope identifier.

```
<cfscript>
  myname = "globalName";
  function readname(name) {
    var myname = "localName";
    return (Evaluate(name));
  }
</cfscript>

<cfoutput>
<!--- This one collides with local variable name. --->
  The result of calling readname with myname is:
      #readname("myname")# <br>
<!--- This one finds the name passed in. --->
  The result of calling readname with Variables.myname is:
    #readname("Variables.myname")#
</cfoutput>
```

Using recursion

A *recursive* function is a function that calls itself. Recursive functions are useful when a problem can be solved by an algorithm that repeats the same operation multiple times using the results of the preceding repetition. Factorial calculation, used in the following example, is one case where recursion is useful. The Towers of Hanoi game is also solved using a recursive algorithm.

A recursive function, like looping code, must have an end condition that always stops the function. Otherwise, the function will continue until a system error occurs or you stop the ColdFusion server.

The following example calculates the factorial of a number, that is, the product of all the integers from 1 through the number; for example, 4 factorial is 4 X 3 X 2 X 1 = 24.

```
function Factorial(factor) {
  If (factor LTE 1)
    return 1;
  else
    return factor * Factorial(factor -1);
}
```

If the function is called with a number greater than 1, it calls itself using an argument one less than it received. It multiplies that result by the original argument, and returns the result. Therefore, the function keeps calling itself until the factor is reduced to 1. The final recursive call returns 1, and the preceding call returns 2 * 1, and so on until all the initial call returns the end result.

Caution: If a recursive function calls itself too many times, it causes a stack overflow. Always test any recursive functions under conditions that are likely to cause the maximum number of recursions to ensure that they do not cause a stack overflow.

CHAPTER 10
Building and Using ColdFusion Components

This chapter describes Macromedia ColdFusion component (CFC) concepts and elements and tells you how to create and use CFCs. It provides you with all the tools to create and use CFCs; however, it does not discuss CFC methodologies or provide information on designing applications for CFC use. For more information on creating applications that use CFCs, see the Macromedia website: www.macromedia.com.

Contents

About ColdFusion components 201

Creating ColdFusion components 204

Using ColdFusion components 216

Passing parameters to methods 225

Using CFCs effectively 230

ColdFusion component example 239

About ColdFusion components

A ColdFusion component (CFC) is a file saved with the extension .cfc. A CFC can contain data and functions. Within a CFC, data is referred to as *properties*. Although you use the `cffunction` tag to define functions within a CFC, they are typically referred to as *methods* instead of functions.

The page on which you define a CFC is also known as a *component page*. Component pages use the same tags and functions that regular CFML pages do, plus a small number of special tags (in particular, the `cfcomponent` tag) and tag attributes.

You define related methods in a CFC. Unlike ColdFusion custom tags, a single CFC can perform many related actions, defined in multiple methods. The methods may share a data context, such as metadata and scoping, or manage a particular database or set of tables. For example, you can define the methods to insert, update, delete, and retrieve records from a particular database or table in one CFC.

This section describes the following topics:

- CFCs and object-oriented programming
- When to use CFCs

CFCs and object-oriented programming

CFCs are building blocks that let you develop ColdFusion code in an object-oriented manner, although CFCs do not require you to do object-oriented programming. Some of the object-oriented features of CFCs include encapsulation, inheritance, and introspection. CFC object-oriented features are similar to the object-oriented elements in other languages, like JavaScript.

The technique of incorporating both code and data into one object such as a CFC is known as *encapsulation*. Encapsulation lets users pass data to and get a result from your CFC without having to understand the underlying code. When you use encapsulation, you can validate data that is passed to the CFC. CFCs can also enforce data types, check for required parameters, and optionally assign default values.

One CFC can *inherit* the methods and properties of another CFC. Inheritance lets you build multiple specific components without rewriting the code for the basic building blocks of the components. For more information, see "Using inheritance and the Super keyword" on page 230.

CFCs support *introspection*; that is, they can provide information about themselves. If you display a component page directly in an HTML browser, inspect it in the ColdFusion and Macromedia Dreamweaver MX component browsers, or use the CFML `GetMetadata` function, you see information about the component. This information includes its path, property, methods, and additional information that you can specify using special documentation attributes and tags. For more information, see "Using introspection to get information about components" on page 236

When you use a ColdFusion component, you can simply *invoke* a method in the CFC. However, typically, you create an *instance* of the CFC, and then invoke methods and refer to properties of the CFC.

When to use CFCs

This section describes the ways in which you can use CFCs:

- Developing structured, reusable code
- Creating web services
- Creating Macromedia Flash Remoting elements
- Using asynchronous CFCs

Developing structured, reusable code

CFCs provide an excellent method for developing structured applications that separate display elements from logical elements and encapsulate database queries. You can use CFCs to create application functionality that you (and others) can reuse wherever needed, similar to user-defined functions (UDFs) and custom tags. If you want to modify, add, or remove component functionality, you make changes in only one component file.

CFCs have several advantages over UDFs and custom tags. These advantages, which CFCs automatically provide, include all of the following:

- The ability to group related methods into a single component, and to group related components into a package
- Properties that multiple methods can share
- The This scope, a component-specific scope
- Inheritance of component methods and properties from a base component, including the use of the Super keyword
- Access control
- Introspection for CFC methods, properties, and metadata

CFCs have one characteristic that prevents them from being the automatic choice for all code reuse. It takes relatively more processing time to instantiate a CFC than to process a custom tag. In turn, it takes substantially more time to process a custom tag than to execute a user-defined function (UDF). However, after a CFC is instantiated, calling a CFC method has about the same processing overhead as an equivalent UDF. As a result, you should not use CFCs in place of independent, single-purpose custom tags or UDFs. Instead, you should use CFCs to create bodies of related methods, particularly methods that share properties.

For more information about UDFs, custom tags, and other ColdFusion code reuse techniques, see Chapter 8, "Creating ColdFusion Elements," on page 161.

Creating web services

ColdFusion MX 7 can automatically publish CFC methods as web services. To publish a CFC method as a web service, you specify the `access="remote"` attribute in the method's `cffunction` tag. ColdFusion generates all the required Web Services Description Language (WSDL) code and exports the CFC methods. For more information on creating web services in ColdFusion, see Chapter 36, "Using Web Services," on page 885.

Creating Macromedia Flash Remoting elements

Flash applications that use Flash Remoting MX can easily take advantage of ColdFusion components for business logic. In a CFC, the `cffunction` tag names the function and contains the application logic, and the `cfreturn` tag returns the result to Flash.

Note: For ColdFusion component methods to communicate with Flash applications, you must set the `access` attribute of the `cffunction` tag to remote.

For more information on creating CFCs for Flash Remoting MX, see "Using Flash with CFCs" on page 817.

Using asynchronous CFCs

ColdFusion MX 7 provides an event gateway that lets you send a message to a CFC asynchronously. This gateway lets you initialize processing by a CFC without waiting for the CFC to complete or return a value. You can use asynchronous CFCs that use this gateway for the following:

- Reindexing a Verity collection
- Logging information
- Running batch processes

For more information on using asynchronous CFCs, see Chapter 42, "About event gateways," on page 1020.

Creating ColdFusion components

When you create CFCs, you create methods, which are ColdFusion user-defined functions, in the component page. You pass data to a method by using parameters. The method then performs the function and, if specified in the `cfreturn` tag, returns data.

You can also define variables in a CFC. Within a CFC, these variables are known as *properties*.

Tags for creating CFCs

The following table lists the tags that you use to create a CFC. You use these tags on the CFML page that defines the CFC.

Tag	Description
`cfcomponent`	Contains a component definition; includes attributes for introspection. For more information, see "Building ColdFusion components" on page 205.
`cffunction`	Defines a component method (function); includes attributes for introspection. For more information, see "Defining component methods" on page 205.
`cfargument`	Defines a parameter (argument) to a method; includes attributes for introspection. For more information, see "Defining and using method parameters" on page 209.
`cfproperty`	Defines variables for CFCs that provide web services; also use to document component properties. For more information, see "The cfproperty tag" on page 215.

Elements of a CFC

A CFC has the following characteristics:

- It is a single CFML page with a .cfc filename extension. The component name is the same as the filename. For example, if the file is myComponent.cfc, the component name is myComponent.
- The page is surrounded by a `cfcomponent` tag. No code can be outside this tag.

- The component page defines methods (functions), properties (data), or both. Most CFCs have methods, or methods and properties, but you can also have a CFC that contains only properties.
- You use the `cffunction` tag to define CFC methods. The CFScript `function` statement can create simple methods, but it does not provide options to control access to the method, provide metadata, specify a return type, or control generated output.
- You can write code on the component page that is outside of `cffunction` definitions. This code executes when the CFC is instantiated or whenever you invoke a method of the CFC.

Building ColdFusion components

You use the `cfcomponent` and `cffunction` tags to create ColdFusion components. By itself, the `cffunction` tag does not provide functionality. The `cfcomponent` tag provides an envelope that describes the functionality that you build in CFML and enclose in `cffunction` tags. The following example shows the skeleton of a component with two methods:

```
<cfcomponent>
  <cffunction name="firstMethod">
     <!--- CFML code for this method goes here. --->
  </cffunction>
  <cffunction name="secondMethod">
     <!--- CFML code for this method goes here. --->
  </cffunction>
</cfcomponent>
```

Defining component methods

You define component methods using `cffunction` tags. The following example defines a CFC that contains two methods, `getall` and `getsalary`:

```
<cfcomponent>
  <cffunction name="getall" output="false" returntype="query">
    <cfset var queryall="">
    <cfquery name="queryall" datasource="cfdocexamples">
      SELECT * FROM EMPLOYEE
    </cfquery>
    <cfreturn queryall>
  </cffunction>
  <cffunction name="getsalary" output="false">
    <cfset var getNamesandSalary="">
      <cfquery name="getNamesandSalary" datasource="cfdocexamples">
        SELECT FirstName, LastName, Salary FROM EMPLOYEE
      </cfquery>
    <cfreturn getNamesandSalary>
  </cffunction>
</cfcomponent>
```

Because component methods are ColdFusion functions, most of their features and coding techniques are identical to those of user-defined functions. For more information on using the `cffunction` tag to create functions, see Chapter 9, "Writing and Calling User-Defined Functions," on page 171. Like other ColdFusion functions, CFC methods can display information directly by generating output, or can return a value to the code or client that invoked the method.

You use the following `cffunction` tag attributes only for CFCs:

- The `displayname` and `hint` attributes, which document the CFC; for more information, see "Documenting CFCs" on page 214.
- The `access` attribute, which controls access to the CFC; for more information, see "Using access security" on page 235.

For detailed reference information on the `cffunction` tag, see *CFML Reference*.

Defining CFCs with related methods

When defining CFCs, it is good programming practice to organize related methods in one CFC. For example, you could put all methods that perform operations related to a user, such as `addUser`, `editUser`, and `storeUserPreferences`, in one CFC. You can group related mathematical functions into one CFC. A CFC can also contain all the methods and properties necessary for a shopping cart. The following CFC contains two `cffunction` tags that define two component methods, `getEmp` and `getDept`. When invoked, the component methods query the ExampleApps database. The `cfreturn` tag returns the query results to the client, or page, where the method was invoked.

```
<cfcomponent>
  <cffunction name="getEmp">
    <cfset var empQuery="">
     <cfquery name="empQuery" datasource="cfdocexamples" dbtype="ODBC" >
      SELECT FIRSTNAME, LASTNAME, EMAIL
      FROM tblEmployees
     </cfquery>
     <cfreturn empQuery>
  </cffunction>
  <cffunction name="getDept">
    <cfset var deptQuery="">
    <cfquery name="deptQuery" datasource="cfdocexamples" dbtype="ODBC" >
      SELECT *
      FROM tblDepartments
     </cfquery>
     <cfreturn deptQuery>
  </cffunction>
</cfcomponent>
```

Putting executable code in a separate file

You can put executable code in a separate file from the main component definition page. By placing the method execution code in a separate file, you can separate property initialization code, meta information, and the method definition shell from the executable method definition code. This technique lets you modularize your code and helps prevent CFML pages from getting too long and complex.

To separate the component method code, use a `cfinclude` tag on the component definition page to call the page that contains the component method code.

Note: If your method takes arguments or returns data to the page that invokes it, the `cfargument` tag and the `cfreturn` tag must be on the component definition page, not on the included page.

To create a component method using the cfinclude tag:

1. Create a tellTime.cfc file with the following code:

```
<cfcomponent>
  <cffunction name="getUTCTime">
    <cfinclude template="getUTCTime.cfm">
  <cfreturn utcStruct.Hour & ":" & utcStruct.Minute>
  </cffunction>
</cfcomponent>
```

2. Create a ColdFusion page with the following code, and save it as getUTCTime.cfm in the same directory as tellTime.cfc:

```
<cfscript>
  serverTime=now();
  utcTime=GetTimeZoneInfo();
  utcStruct=structNew();
  utcStruct.Hour=DatePart("h", serverTime);
  utcStruct.Minute=DatePart("n", serverTime);
  utcStruct.Hour=utcStruct.Hour + utcTime.utcHourOffSet;
  utcStruct.Minute=utcStruct.Minute + utcTime.utcMinuteOffSet;
  if (utcStruct.Minute LT 10)
    utcStruct.Minute = "0" & utcStruct.Minute;
</cfscript>
```

   In the example, the `getUTCTime` method definition calls the getUTCTime.cfm file with the `cfinclude` tag. The getUTCTime.cfm code calculates the UTC time representation of the current time and populates a structure with hour and minute values. The method in tellTime.cfc then uses the information in the structure to return the current UTC time as a string to the calling page. The included page must not include a `cfreturn` statement.

Initializing instance data

Some components have *instance data*, which is data that persists as long as the component instance exists. For example, a shopping cart component might have instance data that includes the IDs and quantities of items that the user puts in the shopping cart. Instance data is often shared by several methods that can create, delete, or modify the data.

You can refer to instance data of a CFC only if you create an instance of the CFC. From inside the CFC, you refer to instance data of the CFC using the `this` prefix, for example `this.firstvariable`. From the calling page, you refer to instance data using dot notation, including the name of the instance of the component and the name of the instance data, as in `objectname.ivarname`. Components whose methods you invoke without first instantiating the component do not typically have instance data.

You initialize instance data at the top of the component definition, before the method definitions. ColdFusion executes this code when it instantiates the component; for example, when a `cfobject` tag creates the component instance. Because this code executes only when the instance is created and it typically "constructs" properties of the component, instance data initialization code is sometimes called *constructor* code.

You can use any CFML tag or function in constructor code, and the code can perform any ColdFusion processing, such as querying a database or data validation and manipulation. If one component extends another, the parent component's constructor code executes before the child component's constructor code.

Note: ColdFusion does not require you to put the initialization code at the top of the component definition; however, it is good programming practice to do so.

The following example shows constructor code for a shopping cart CFC:

```
<cfcomponent>
  <!--- Initialize the array for the cart item IDs and quantities. --->
  <cfset This.CartData = ArrayNew(2)>
  <!--- The following variable has the ID of the "Special Deal" product for
      this session. --->
  <cfset This.Special_ID = RandRange(1, 999)>
```

For information on scopes, see "The This scope" on page 227 and "The Variables scope" on page 228.

A useful technique is to define a method named `init()`, which initializes an instance of a CFC, acting as a constructor. The `init()` method can initialize constants and return an instance of the component to the calling page. The following code illustrates an example of an `init()` method:

```
<cfcomponent displayname="shoppingCart">
  <cffunction name="init" access="public" output="no"
  returntype="shoppingCart">
    <cfargument name="shoppingCartID" type="UUID" required="yes">
    <cfset variables.shoppingCartID = arguments.shoppingCartID>
    <cfreturn this>
  </cffunction>

  <!--- Additional methods go here. --->
</cfcomponent>
```

In this example, the `init()` method uses the variables scope to make the shopping cart ID available anywhere in the CFC. For more information about scope, see "CFC variables and scope" on page 227.

Defining and using method parameters

You pass data to a method by using parameters. To define a component method parameter, use the `cfargument` tag in the `cffunction` tag body. To define multiple parameters, use multiple `cfargument` tags. The tag names a parameter and lets you specify the following:

- Whether the parameter is required
- The type of data that is required
- A default argument value
- Display name and hint metadata for CFC introspection

Note: You can create CFC methods that do not use `cfargument` tags, for example, if you use positional parameters in your methods. However, most CFC methods use the `cfargument` tag.

Example: convertTemp.cfc

The convertTemp.cfc file consists of the following:

```
<cfcomponent>
  <!--- Celsius to Fahrenheit conversion method. --->
  <cffunction name="ctof" output="false">
    <cfargument name="temp" required="yes" type="numeric">
    <cfreturn ((temp*9)/5)+32>
  </cffunction>

  <!--- Fahrenheit to Celsius conversion method. --->
  <cffunction name="ftoc" output="false">
    <cfargument name="temp" required="yes" type="numeric">
    <cfreturn ((temp-32)*5/9)>
  </cffunction>
</cfcomponent>
```

Reviewing the code

The convertTemp CFC contains two methods that convert temperature. The following table describes the code and its function:

Code	Description
`<cfcomponent>`	Defines the component.
`<cffunction name="ctof" output="false">`	Defines the `ctof` method. Indicates that this method does not display output.
`<cfargument name="temp" required="yes" type="numeric">`	Creates the `temp` parameter of the `ctof` method. Indicates that it is required and that the expected value is `numeric`.
`<cfreturn ((temp*9)/5)+32>`	Defines the value that the method returns.
`</cffunction>`	Ends the method definition.
`<cffunction name="ftoc" output="false">`	Defines the `ftoc` method. Indicates that this method does not display output.
`<cfargument name="temp" required="yes" type="numeric">`	Creates the `temp` parameter of the `ftoc` method. Indicates that it is required and that the expected value is `numeric`.

Code	Description
`<cfreturn ((temp-32)*5/9)>`	Defines the value that the method returns.
`</cffunction>`	Ends the method definition.
`</cfcomponent>`	Ends the component definition.

Example: tempConversion.cfm

The ColdFusion page tempConversion.cfm is an HTML form in which the user enters the temperature to convert, and selects the type of conversion to perform. When the user clicks the Submit button, ColdFusion performs the actions on the processForm.cfm page. The file tempConversion.cfm, which should be in the same directory as convertTemp.cfc, consists of the following:

```
<cfform action="processForm.cfm" method="POST">
  Enter the temperature:
    <input name="temperature" type="text"><br><br>
  Select the type of conversion:<br>
    <select name="conversionType">
      <option value="CtoF">Celsius to Farenheit</option>
      <option value="FtoC">Farenheit to Celsius</option>
    </select><br><br>
  <input name="submitform" type="submit" value="Submit">
</cfform>
```

Example: processForm.cfm

The ColdFusion page processForm.cfm calls the appropriate component method, based on what the user entered in the form on the tempConversion.cfm page. It should be in the same directory as convertTemp.cfc.

```
<cfif #form.conversionType# is "CtoF">
  <cfinvoke component="convertTemp" method="ctof" returnvariable="newtemp"
    temp=#form.temperature#>
  <cfoutput>#form.temperature# degrees Celsius is #newtemp# degrees
    Farenheit.</cfoutput>
  <cfelseif #form.conversionType# is "FtoC">
    <cfinvoke component="convertTemp" method="ftoc"
      returnvariable="newtemp" temp=#form.temperature#>
    <cfoutput>#form.temperature# degrees Fahrenheit is #newtemp# degrees
      Celsius.</cfoutput>
</cfif>
```

Reviewing the code

The file processForm.cfm invokes the appropriate component method. The following table describes the code and its function:

Code	Description
`<cfif form.conversionType is "CtoF">`	Executes the code in the `cfif` block if the user selected Celsius to Fahrenheit as the conversion type in the form on the tempConversion.cfm page.
`<cfinvoke component="convertTemp" method="ctof" returnvariable="newtemp" arguments.temp="#form.temperature#" >`	Invokes the `ctof` method of the `convertTemp` component, without creating an instance of the `convertTemp` component. Specifies `newtemp` as the result variable for the method. Assigns the temperature value that the user entered in the form to the variable `temp`, which is specified in the `cfargument` tag of the `ctof` method. When invoking the `ctof` method, the `temp` variable is assigned to the Arguments scope. For more information about variables and scope, see "CFC variables and scope" on page 227.
`<cfoutput>#form.temperature# degrees Celsius is #newtemp# degrees Fahrenheit.</cfoutput>`	Displays the temperature that the user entered in the form, the text "degrees Celsius is," the new temperature value that results from the `ctof` method, and the text "degrees Fahrenheit."
`<cfelseif #form.conversionType# is "FtoC">`	Executes the code in the `cfelseif` block if the user selected Fahrenheit to Celsius as the conversion type in the form on the tempConversion.cfm page.
`<cfinvoke component="converttemp" method="ftoc" returnvariable="newtemp" temp=#form.temperature#>`	Invokes the `ftoc` method of the `convertTemp` component, without creating an instance of the `convertTemp` component. Specifies `newtemp` as the result variable for the method. Assigns the temperature value that the user entered in the form to the variable `temp`, which is specified in the `cfargument` tag of the `ftoc` method. When invoking the `ftoc` method, the `temp` variable is assigned to the Arguments scope. For more information about variables and scope, see "CFC variables and scope" on page 227
`<cfoutput>#form.temperature# degrees Fahrenheit is #newtemp# degrees Celsius.</cfoutput>`	Displays the temperature that the user entered in the form, the text "degrees Fahrenheit is," the new temperature value that results from the `ftoc` method, and the text "degrees Celsius."
`</cfif>`	Closes the `cfif` block.

To run the example, display the tempConversion.cfm page in your browser. When you enter a value in the text box of the form, the value is stored in the `form.temperature` variable. Processing is then performed on the processForm.cfm page, which refers to the value as `form.temperature`. When you invoke either method, the `cfinvoke` tag assigns the value `form.temperature` to `temp`; `temp` is the argument specified in the `cfargument` tag of the appropriate method. The appropriate method in the `convertTemp` component performs the necessary calculations and returns the new value as `newtemp`.

For detailed reference information on the `cfargument` tag, see *CFML Reference*.

To access the parameter values in the component method definition, use structure- or array-like notation with the Arguments scope. The following example refers to the `lastName` argument as `Arguments.lastname`; it could also refer to it as `Arguments[1]`. In addition, you can access arguments directly using number (#) signs, such as `#lastname#`; however, it is better programming practice to identify the scope (for example, `#Arguments.lastname#`). Also, you can use Array- or structure-like notation, which lets you loop over multiple parameters.

For more information on the Arguments scope, see "The Arguments scope" on page 229.

To define parameters in the component method definition:

- Create a new component with the following contents, and save it as corpQuery.cfc in a directory under your web root directory:

```
<cfcomponent>
  <cffunction name="getEmp">
    <cfargument name="lastName" type="string" required="true"
        hint="Employee last name">
    <cfset var empQuery="">
     <cfquery name="empQuery" datasource="cfdocexamples">
      SELECT LASTNAME, FIRSTNAME, EMAIL
      FROM tblEmployees
      WHERE LASTNAME LIKE '#Arguments.lastName#'
     </cfquery>
      <!--- Use cfdump for debugging purposes. --->
     <cfdump var=#empQuery#>
  </cffunction>
  <cffunction name="getCat" hint="Get items below specified cost">
  <cfargument name="cost" type="numeric" required="true">
    <cfset var catQuery="">
    <cfquery name="catQuery" datasource="cfdocexamples">
      SELECT ItemName, ItemDescription, ItemCost
      FROM tblItems
      WHERE ItemCost <= #Arguments.cost#
     </cfquery>
      <!--- Use cfdump for debugging purposes. --->
     <cfdump var=#catQuery#>
  </cffunction>
</cfcomponent>
```

In the example, the `cfargument` attributes specify the following:

- The `name` attributes define the parameter names.
- The `type` attribute for the `lastName` argument specifies that the parameter must be a text string. The `type` attribute for the `cost` argument specifies that the parameter must be a numeric value. These attributes validate the data before it is submitted to the database.
- The `required` attributes indicate that the parameters are required or an exception will be thrown.
- The Arguments scope provides access to the parameter values.

Providing results

ColdFusion components can provide information in the following ways:

- They can generate output that is displayed on the calling page.
- They can return a variable.

You can use either technique, or a combination of both, in your applications. The technique that you use should depend on your application's needs and your coding methodologies. For example, many CFC methods that perform business logic return the results as a variable, and many CFC methods that display output directly are designed as modular units for generating output, and do not do business logic.

Displaying output

If you do not specifically suppress output, any text, HTML code, or output that CFML tags generate inside your method gets returned as generated output to the client that calls the component method. If the client is a web browser, it displays these results. For example, the following `getLocalTime1` component method shows the local time directly on the page that invokes the method:

```
<cfcomponent>
  <cffunction name="getLocalTime1">
    <cfoutput>#TimeFormat(now())#</cfoutput>
  </cffunction>
</cfcomponent>
```

Component methods that are called using Flash Remoting or as web services cannot use this method to provide results.

Returning a results variable

In the component method definition, you use the `cfreturn` tag to return the results to the client as variable data. For example, the following `getLocalTime2` component method returns the local time as a variable to the ColdFusion page or other client that invokes the method:

```
<cfcomponent>
  <cffunction name="getLocalTime">
    <cfreturn TimeFormat(now())>
  </cffunction>
</cfcomponent>
```

The ColdFusion page or other client, such as a Flash application, that receives the result then uses the variable data as appropriate.

Note: If a CFC is invoked using a URL or by submitting a form, ColdFusion MX returns the variable as a WDDX packet. A CFC that is invoked by Flash Remoting MX, or any other instance of a CFC, must not return the This scope.

You can return values of all data types, including strings, integers, arrays, structures, and instances of CFCs. The `cfreturn` tag returns a single variable, as does the `return` CFScript statement. Therefore, if you want to return more than one result value at a time, use a structure. If you do not want to display output in a method, use `output="false"` in the `cffunction` tag.

For more information on using the `cfreturn` tag, see *CFML Reference*.

Documenting CFCs

ColdFusion provides several ways to include documentation about your CFCs in your component definitions. The documentation is available when you use introspection to display information about the CFC or call the `GetMetadata` function to get the component's metadata. You can use the following tools for documenting CFCs:

- The `displayname` and `hint` attributes
- User-defined metadata attributes
- The `cfproperty` tag

The following sections describe these tools. For information on displaying the information, see "Using introspection to get information about components" on page 236.

The displayname and hint attributes

The `cfcomponent`, `cffunction`, `cfargument`, and `cfproperty` tags have `displayname` and `hint` attributes.

The `displayname` attribute lets you provide a more descriptive name for a component, attribute, method, or property. When you use introspection, this attribute appears in parentheses next to the component or method name, or on the parameter information line.

You use the `hint` attribute for longer descriptions of the component, method, or argument. In the introspection display, this attribute appears on a separate line or on several lines of the component or method description, and at the end of the argument description.

Metadata attributes

You can include arbitrary metadata information as attributes of the `cfcomponent`, `cffunction`, `cfargument`, and `cfproperty` tags. To create a metadata attribute, specify the metadata attribute name and its value. For example, in the following `cfcomponent` tag, the `Author` attribute is a metadata attribute. This attribute is not used as a function parameter; instead, it indicates who wrote this CFC.

```
<cfcomponent name="makeForm" Author="Bean Lapin">
```

Metadata attributes are not used by ColdFusion MX for processing; they also do not appear in standard ColdFusion introspection displays; however, you can access and display them by using the `GetMetaData` function to get the metadata. Each attribute name is a key in the metadata structure of the CFC element.

Metadata attributes are used for more than documentation. Your application can use the `GetMetadata` function to get the metadata attributes and act based on the values. For example, a `mathCFC` component might have the following `cfcomponent` tag:

```
<cfcomponent displayname="Math Functions" MetaType="Float">
```

In this case, a ColdFusion page with the following code sets the `MetaTypeInfo` variable to `Float`:

```
<cfobject component="mathCFC" name="MathFuncs">
<cfset MetaTypeInfo=GetMetadata(MathFuncs).MetaType>
```

Note: All metadata values are replaced by strings in the metadata structure returned from the `GetMetadata` function. Because of this, you should not use expressions in custom metadata attributes.

The cfproperty tag

The `cfproperty` tag is used to create complex data types with WSDL descriptors and for component property documentation, as follows:

- It can create complex data types with WSDL descriptions for ColdFusion web services. For more information, see "Using ColdFusion components to define data types for web services" on page 902.
- It can provide documentation of component properties in the ColdFusion introspection output. The introspection information includes the values of the standard `cfproperty` tag attributes.

Note: The `cfproperty` tag does *not* create a variable or assign it a value. It is used for information purposes only. You use a `cfset` tag, or CFScript assignment statement, to create the property and set its value.

Saving and naming ColdFusion components

The following table lists the locations in which you can save component files and how they can be accessed from each location:

	URL	Form	Flash Remoting	Web services	ColdFusion page
Current directory	N/A	Yes	N/A	N/A	Yes
Web root	Yes	Yes	Yes	Yes	Yes
ColdFusion mappings	No	No	No	No	Yes
Custom tag roots	No	No	No	No	Yes

Note: ColdFusion MX mappings and custom tag roots can exist within the web root. If so, they are accessible to remote requests, including URL, form, Flash Remoting, and web services invocation.

When you store components in the same directory, they are members of a component *package*. You can group related CFCs into packages. Your application can refer to any component in a directory specifically by using a qualified component name that starts with a subdirectory of one of the accessible directories and uses a period to delimit each directory in the path to the directory that contains the component. For example, the following example is a qualified name of a component named `price`:

```
catalog.product.price
```

In this example, the price.cfc file must be in the catalog\product subdirectory of a directory that ColdFusion searches for components, as listed in the preceding table. When you refer to a component using the qualified name, ColdFusion looks for the component in the order described in "Specifying the CFC location" on page 224.

Establishing a descriptive naming convention is a good practice, especially if you plan to install the components as part of a packaged application.

Using ColdFusion components

There are two ways to use a CFC:

- You can *instantiate* a CFC object, which creates a CFC instance. You then invoke the methods of the instance. You can access the CFC methods and data as instance elements. You can also use the instance in the `cfinvoke` tag to invoke the CFC methods. When you instantiate a CFC, data in the CFC is preserved as long as the CFC instance exists, and ColdFusion does not incur the overhead of creating the instance each time you call a method.

  Instantiate CFCs to preserve data in the CFC. To ensure processing efficiency if you use the CFC more than once on a page, instantiate the CFC before you invoke its methods.

  Methods that are executed remotely through Flash Remoting and web services always create a new instance of the CFC before executing the method.
- You can *invoke* (call) a method of the CFC without creating an instance of the CFC, which is referred to as *transiently* invoking a method. In this case, ColdFusion creates an instance of the CFC that exists only from the time you invoke the method until the method returns a result. No data is preserved between invocations and there is no instance of the CFC that you can reuse elsewhere in your CFML. It is considered a best practice to create an instance of a CFC before invoking any of its methods, unless your CFML request uses the CFC only once. If you transiently invoke a method frequently, consider creating a user-defined function to replace the CFC method.

You can create persistent CFCs by assigning the CFC instance to a persistent scope, such as the Session or Application scope. This way, you can create CFCs for objects, such as shopping carts or logged-in users, that must persist for sessions. You can also create CFCs that provide application-specific data and methods.

This section describes how to use CFCs, including the following topics:

- Tags for using CFCs
- CFC invocation techniques
- Instantiating CFCs

- Invoking CFC methods with the cfinvoke tag
- Using components directly in CFScript and CFML
- Invoking CFC methods using forms and URLs
- Accessing CFCs from outside ColdFusion and basic HTML
- Specifying the CFC location

Tags for using CFCs

The following table lists the tags that you use to instantiate or invoke a CFC. You use these tags on the CFML page on which you instantiate or invoke the CFC.

Tag	Description
`cfinvoke`	Invokes a method of a CFC.
`cfinvokeargument`	Passes the name and value of a parameter to a component method.
`cfobject`	Creates a CFC instance.
`CreateObject`	Creates a CFC instance.

CFC invocation techniques

ColdFusion provides many ways to instantiate CFCs and invoke CFC methods. The following table lists the techniques, including the ColdFusion tags and functions that you use:

Invocation	Description	For more information
`cfinvoke` tag	Invokes a component method. Can invoke methods of a CFC instance or invoke the methods transiently.	See "Invoking CFC methods with the cfinvoke tag" on page 218.
`cfset` tag and assignment statements	Invoke methods and access properties of a component instance.	See "Using components directly in CFScript and CFML" on page 220.
URL (HTTP GET)	Transiently invokes a component method by specifying the component and method names in the URL string.	See "Invoking component methods using a URL" on page 222.
Form control (HTTP POST)	Transiently invokes a component method using the HTML `form` and `input` tags and their attributes.	See "Invoking component methods using a form" on page 222.
Flash Remoting	ActionScript can transiently invoke component methods.	See Chapter 33, "Using the Flash Remoting Service," on page 807.
Web services	The `cfinvoke` tag and CFScript consume web services in ColdFusion MX 7. External applications can also consume CFC methods as web services.	See Chapter 36, "Using Web Services," on page 885.

Instantiating CFCs

If you use a CFC multiple times in a ColdFusion request, or if you use a CFC with persistent properties, use the `cfobject` tag or `CreateObject` function to instantiate the CFC before you call its methods.

The following example uses the `cfobject` tag to create an instance of the tellTime CFC.

```
<cfobject component="tellTime" name="tellTimeObj">
```

The following example uses the `CreateObject` function to instantiate the same component in CFScript:

```
tellTimeObj = CreateObject("component", "tellTime");
```

Invoking CFC methods with the cfinvoke tag

The `cfinvoke` tag can invoke methods on a CFC instance or invoke CFC methods transiently. You can also use the `cfinvoke` tag to invoke CFC methods from within a CFC.

Invoking methods of a CFC instance

To invoke a component method of a CFC instance, use the `cfinvoke` tag and specify the following:

- The CFC instance name, *enclosed in number signs* (#), in the `component` attribute.
- The method name, in the `method` attribute.
- Any parameters. For information on passing parameters, see "Passing parameters to methods using the cfinvoke tag" on page 225.
- If the component method returns a result, the name of the variable that will contain the result in the `returnVariable` attribute.

To invoke a method of a component instance using the cfinvoke tag:

1. Create a file named tellTime2.cfc with the following code:

```
<cfcomponent>
  <cffunction name="getLocalTime" access="remote">
    <cfreturn TimeFormat(now())>
  </cffunction>
  <cffunction name="getUTCTime" access="remote">
    <cfscript>
      serverTime=now();
      utcTime=GetTimeZoneInfo();
      utcStruct=structNew();
      utcStruct.Hour=DatePart("h", serverTime);
      utcStruct.Minute=DatePart("n", serverTime);
      utcStruct.Hour=utcStruct.Hour + utcTime.utcHourOffSet;
      utcStruct.Minute=utcStruct.Minute + utcTime.utcMinuteOffSet;
      if (utcStruct.Minute LT 10)
        utcStruct.Minute = "0" & utcStruct.Minute;
    </cfscript>
    <cfreturn utcStruct.Hour & ":" & utcStruct.Minute>
  </cffunction>
</cfcomponent>
```

The example defines two component methods: `getLocalTime` and `getUTCTime`.

2. Create a new ColdFusion page, with the following code and save it in the same directory as the tellTime component:

```
<!--- Create the component instance. --->
<cfobject component="tellTime2" name="tellTimeObj">
<!--- Invoke the methods. --->
<cfinvoke component="#tellTimeObj#" method="getLocalTime"
  returnvariable="localTime" >
<cfinvoke component="#tellTimeObj#" method="getUTCTime"
  returnvariable="UTCTime" >
<!--- Display the results. --->
<h3>Time Display Page</h3>
<cfoutput>
  Server's Local Time: #localTime#<br>
  Calculated UTC Time: #UTCTime#
</cfoutput>
```

This example uses the `cfobject` tag to create an instance of the tellTime component and the `cfinvoke` tag to invoke the instance's `getLocalTime` and `getUTCTime` methods. In this example, the CFC contains the functional logic in the methods, which return a result to the calling page, and the calling page displays the results. This structure separates the logic from the display functions, which usually results in more reusable code.

Invoking component methods transiently using the cfinvoke tag

In ColdFusion pages or components, the `cfinvoke` tag can invoke component methods without creating a persistent CFC instance.

To invoke a component method transiently, use the `cfinvoke` tag and specify the following:

- The name or path of the component, in the `component` attribute.
- The method name, in the `method` attribute.
- Any parameters. For information on passing parameters, see "Passing parameters to methods using the cfinvoke tag" on page 225.
- If the component method returns a result, the name of the variable that will contain the result, in the `returnVariable` attribute.

To invoke a component method using the cfinvoke tag:

1. Create the following component and save it as tellTime.cfc:

```
<cfcomponent>
  <cffunction name="getLocalTime">
    <cfoutput>#TimeFormat(now())#</cfoutput>
  </cffunction>
</cfcomponent>
```

The example defines a component with one method, `getLocalTime`, that displays the current time.

2. Create a ColdFusion page, with the following code, and save it in the same directory as the tellTime component:

```
<h3>Time Display Page</h3>
<b>Server's Local Time:</b>
<cfinvoke component="tellTime" method="getLocalTime">
```

Using the `cfinvoke` tag, the example invokes the `getLocalTime` component method without creating a persistent CFC instance.

Using the cfinvoke tag within the CFC definition

You can use the `cfinvoke` tag to invoke a component method within the component definition; for example, to call a utility method that provides a service to other methods in the component. To use the `cfinvoke` tag in this instance, do not create an instance or specify the component name in the `cfinvoke` tag, as the following example shows:

```
<cfcomponent>
  <cffunction name="servicemethod" access="public">
    <cfoutput>At your service...<br></cfoutput>
  </cffunction>
  <cffunction name="mymethod" access="public">
    <cfoutput>We're in mymethod.<br></cfoutput>
    <!--- Invoke a method in this CFC. --->
    <cfinvoke method="servicemethod">
  </cffunction>
</cfcomponent>
```

Note: When you invoke a method from within the component definition in which you define the method, do not use the This scope, because this resets the access privileges.

Invoking methods using dynamic method names

The `cfinvoke` tag is the only way to efficiently invoke different component methods based on variable data (for example, form input). In this case, you use a variable name, such as Form.method, as the value of the `method` attribute. In the following example, the user selects a report from a form:

```
<select name="whichreport">
  <option value="all">Complete Report</option>
  <option value="salary">Salary Information</option>
</select>
```

The `cfinvoke` tag then invokes the appropriate method, based on what the user selected:

```
<cfinvoke component="getdata" method="#form.whichreport#"
  returnvariable="queryall">
```

Using components directly in CFScript and CFML

You can invoke methods of a component instance directly using CFScript or in CFML tags. To invoke component methods directly, use the `CreateObject` function or `cfobject` tag to instantiate the component. Thereafter, use the instance name followed by a period and the method that you are calling to invoke an instance of the method. You must always use parentheses after the method name, even if the method does not take any parameters.

You can use this syntax anywhere that you can use a ColdFusion function, such as in `cfset` tags or surrounded by number signs in the body of a `cfoutput` tag.

Invoking component methods in CFScript

The following example shows how to invoke component methods in CFScript:

```
<!--- Instantiate once and reuse the instance.--->
<cfscript>
  tellTimeObj=CreateObject("component","tellTime");
  WriteOutput("Server's Local Time: " & tellTimeObj.getLocalTime());
  WriteOutput("<br> Calculated UTC Time: " & tellTimeObj.getUTCTime());
</cfscript>
```

In the example, the three CFScript statements do the following:

1. The `CreateObject` function instantiates the tellTime CFC as tellTimeObj.
2. The first `WriteOutput` function displays text followed by the results returned by the `getLocalTime` method of the tellTimeObj instance.
3. The second `WriteOutput` function displays text followed by the results returned by the `getUTCTime` method of the tellTimeObj instance.

In CFScript, you use the method name in standard function syntax, such as `methodName()`.

Invoking component methods in CFML

The following example uses CFML tags to produce the same results as the CFScript example:

```
<cfobject name="tellTimeObj" component="tellTime">
<cfoutput>
  Server's Local Time: #tellTimeObj.getLocalTime()#<br>
  Calculated UTC Time: #tellTimeObj.getUTCTime()#
</cfoutput>
```

Accessing component data directly

You can access data in the component's This scope directly in CFScript and `cfset` assignment statements. For example, if a user data CFC has a `This.lastUpdated` property, you could have code such as the following:

```
<cfobject name="userDataCFC" component="userData">
<cfif DateDiff("d", userDataCFC.lastUpdated, Now()) GT 30>
  <!--- Code to deal with older data goes here. --->
</cfif>
```

For more information, see "The This scope" on page 227.

Invoking CFC methods using forms and URLs

You can invoke CFC methods directly by specifying the CFC in a URL, or using HTML and CFML form tags. Because all HTTP requests are transient, these methods only let you transiently invoke methods. They do not let you create persistent CFC instances.

Invoking component methods using a URL

To invoke a component method using a URL, you must append the method name to the URL in standard URL query-string, name-value syntax. You can invoke only one component method per URL request; for example:

```
http://localhost:8500/tellTime.cfc?method=getLocalTime
```

Note: To use URL invocation, you must set the `access` attribute of the `cffunction` tag to `remote`.

To pass parameters to component methods using a URL, append the parameters to the URL in standard URL query-string, name-value pair syntax; for example:

```
http://localhost:8500/corpQuery.cfc?method=getEmp&lastName=camden
```

To pass multiple parameters within a URL, use the ampersand character (&) to delimit the name-value pairs; for example:

```
http://localhost:8500/
  corpQuerySecure.cfc?method=getAuth&store=women&dept=shoes
```

Note: To ensure data security, Macromedia strongly recommends that you not pass sensitive information over the web using URL strings. Potentially sensitive information includes all personal user information, including passwords, addresses, telephone numbers, and so on.

If a CFC method that you access using the URL displays output directly, the user's browser shows the output. You can suppress output by specifying `output="No"` in the `cffunction` tag. If the CFC returns a result using the `cfreturn` tag, ColdFusion converts the text to HTML edit format (with special characters replaced by their HTML escape sequences), puts the result in a WDDX packet, and includes the packet in the HTML that it returns to the client.

Invoking component methods using a form

To invoke a method using a ColdFusion or HTML form, the following must be true:

- The `form` or `cfform` tag `action` attribute must specify the CFC filename or path followed by `?method=`*`methodname`*, where methodname is the name of the method; for example, `<form action="myComponent.cfc?method=myMethod" method="POST">`.
- The form must have an input tag for each component method parameter. The `name` attribute of the tag must be the method parameter name and the field value is the parameter value.
- The `cffunction` tag that defines the CFC method being invoked must specify the `access="remote"` attribute.

If the CFC method that you invoke from the form displays output directly, the user's browser shows the output. (You can use the `cffunction` tag `output` attribute to disable displaying output.) If the CFC returns a result using the `cfreturn` tag, ColdFusion converts the text to HTML edit format, puts it in a WDDX packet, and includes the packet in the HTML that it returns to the client.

To invoke component methods using a form:

1. Create a corpFind.cfm file with the following contents:

```
<h2>Find People</h2>
<form action="components/corpQuery.cfc?method="getEmp" method="post">
  <p>Enter employee's last Name:</p>
  <input type="Text" name="lastName">
  <input type="Hidden" name="method" value="getEmp">
  <input type="Submit" title="Submit Query"><br>
</form>
```

   In the example, the `form` tag's `action` attribute points to the `corpQuery` component and invokes the `getEmp` method.

2. Create a corpQuery.cfc file, specifying `access="remote"` for each `cffunction` tag, as the following example shows:

```
<cfcomponent>
  <cffunction name="getEmp" access="remote">
    <cfargument name="lastName" required="true">
    <cfset var empQuery="">
     <cfquery name="empQuery" datasource="cfdocexamples">
      SELECT LASTNAME, FIRSTNAME, EMAIL
      FROM tblEmployees
      WHERE LASTNAME LIKE '#arguments.lastName#'
     </cfquery>
     <cfoutput>Results filtered by #arguments.lastName#:</cfoutput><br>
     <cfdump var=#empQuery#>
  </cffunction>
</cfcomponent>
```

3. Open a web browser and enter the following URL:

```
http://localhost/corpFind.cfm
```

   ColdFusion displays the search form. After you enter values and click the Submit Query button, the browser displays the results.

Accessing CFCs from outside ColdFusion and basic HTML

Flash applications that use Flash Remoting MX can easily take advantage of ColdFusion components for business logic. Similarly, you can export CFCs so that any application can access CFC methods as web services.

Using Flash Remoting MX

For ColdFusion component methods to communicate with Flash Remoting MX applications, you must set the `access` attribute of the `cffunction` tag to `remote`.

For more information on creating CFCs for Flash Remoting MX, see Chapter 33, "Using the Flash Remoting Service," on page 807.

Invoking components as web services

Any application, whether it is a ColdFusion application, a Java application, JSP page, or a .Net application, can access well-formed ColdFusion components as web services by referencing the WSDL file that ColdFusion automatically generates.

To see a component's WSDL definition, specify the component web address in a URL, followed by ?wsdl; for example:

```
http://localhost:8500/MyComponents/arithCFC.cfc?wsdl
```

For more information on using CFCs as web services, see Chapter 36, "Using Web Services," on page 885.

Specifying the CFC location

When you instantiate or invoke a component, you can specify the component name only, or you can specify a *qualified* path. To specify a qualified path, separate the directory names with periods, not backslashes. For example, myApp.cfcs.myComponent specifies the component defined in myApp\cfcs\myComponent.cfc. For additional information, see "Saving and naming ColdFusion components" on page 215.

ColdFusion uses the following rules to find the specified CFC:

- If you use a `cfinvoke` or `cfobject` tag, or the `CreateObject` function, to access the CFC from a CFML page, ColdFusion searches directories in the following order:
  a. Local directory of the calling CFML page
  b. Web root
  c. Directories specified on the Custom Tag Paths page of ColdFusion MX Administrator
- If you specify only a component name, ColdFusion searches each of these directories, in turn, for the component.
- If you specify a qualified path, such as myApp.cfcs.myComponent, ColdFusion MX looks for a directory matching the first element of the path in each of these directories (in this example, myApp). If ColdFusion MX finds a matching directory, it looks for a file in the specified path beneath that directory, such as myApp\cfcs\myComponent.cfc, relative to each of these directories.

  Note: If ColdFusion finds a directory that matches the first path element, but does not find a CFC under that directory, ColdFusion returns a *not found* error and does *not* search for another directory.

- If you invoke a CFC method remotely, using a specific URL, a form field, Flash Remoting MX, or a web service invocation, ColdFusion looks in the specified path relative to the web root. For form fields and URLs that are specified directly on local web pages, ColdFusion also searches relative to the page directory.

Note: On UNIX and Linux systems, ColdFusion MX attempts to match a CFC name or Custom tag name with a filename, as follows: First, it attempts to find a file with the name that is all lowercase. If it fails, it tries to find a file whose case matches the CFML case. For example, if you specify `<cfobject name="myObject" Component="myComponent">`, ColdFusion first looks for mycomponent.cfc and, if it doesn't find it, ColdFusion looks for myComponent.cfc.

Passing parameters to methods

This section describes how to pass parameters to a method in a CFC, including:

- Passing parameters to methods using the cfinvoke tag
- Passing parameters in direct method invocations
- Passing parameters in a URL

Passing parameters to methods using the cfinvoke tag

When you use the `cfinvoke` tag, ColdFusion MX provides several methods for passing parameters to CFC methods:

- As `cfinvoke` tag attributes, in `name="value"` format
- In the `cfinvoke` tag `argumentcollection` attribute
- In the `cfinvoke` tag body, using the `cfinvokeargument` tag

You can use any combination of these methods in a single invocation. If you use the same name in two or three of these methods, ColdFusion uses the value based on the following order of precedence:

1. `cfinvokeargument` tags
2. `cfinvoke` attribute name-value pairs
3. `argumentcollection` arguments

Passing parameters using attribute format

You can pass parameters in the `cfinvoke` tag as tag attribute name-value pairs, as the following example shows:

```
<cfinvoke component="authQuery" method="getAuthSecure"
  lastName="#session.username#" password="#url.password#">
```

In the example, the parameters are passed as the `lastName` and `password` attributes.

Note: The `cfinvoke` tag attribute names are reserved and cannot be used for parameter names. The reserved attribute names are: `component`, `method`, `argumentCollection`, and `returnVariable`. For more information, see *CFML Reference*.

Passing parameters in the argumentCollection attribute

If you save attributes to a structure, you can pass the structure directly using the `cfinvoke` tag's `argumentCollection` attribute. This technique is useful if an existing structure or scope (such as the Forms scope) contains values that you want to pass to a CFC as parameters, and for using conditional or looping code to create parameters.

When you pass an argumentCollection structure, each structure key is the name of a parameter inside the structure.

The following example passes the Form scope to the `addUser` method of the UserDataCFC component. In the method, each form field name is a parameter name; the method can use the contents of the form fields to add a user to a database.

```
<cfinvoke component="UserDataCFC" method="addUser"
  argumentCollection="#Form#">
```

Passing parameters using the cfinvokeargument tag

To pass parameters in the `cfinvoke` tag body, use the `cfinvokeargument` tag. Using the `cfinvokeargument` tag, for example, you can build conditional processing that passes a different parameter based on user input.

The following example invokes the `corpQuery` component:

```
<cfinvoke component="corpQuery" method="getEmp">
  <cfinvokeargument name="lastName" value="Wilder">
</cfinvoke>
```

The `cfinvokeargument` tag passes the `lastName` parameter to the component method.

In the following example, a form already let the user select the report to generate. After instantiating the `getdata` and `reports` components, the action page invokes the `doquery` component instance, which returns the query results in `queryall`. The action page then invokes the `doreport` component instance and uses the `cfinvokeargument` tag to pass the query results to the `doreport` instance, where the output is generated.

```
<cfobject component="getdata" name="doquery">
<cfobject component="reports" name="doreport">
  <cfinvoke component="#doquery#" method="#form.whichreport#"
  returnvariable="queryall">
  <cfinvoke component="#doreport#"method="#form.whichreport#">
    <cfinvokeargument name="queryall" value="#queryall#">
  </cfinvoke>
```

Passing parameters in direct method invocations

ColdFusion provides three methods for passing parameters to CFC methods in direct method invocations:

- You can pass the parameters the form of comma-separated `name="value"` entries, as in the following CFScript example:

  ```
  authorized = securityCFC.getAuth(name="Almonzo", Password="LauRa123");
  ```

- You can pass the parameters in an `argumentCollection` structure. The following code is equivalent to the previous example:

  ```
  argsColl = structNew();
  argsColl.username = "Amonzo";
  argsColl.password = "LauRa123;
  authorized = securityCFC.getAuth(argumentCollection = argsColl);
  ```

- You can pass positional parameters to a method by separating them with commas. The following example calls the `getAuth` method, and passes the name and password as positional parameters:

  ```
  authorized = securityCFC.getAuth("Almonzo", "LauRa123");
  ```

  Note: For more information on using positional parameters and component methods in ColdFusion functions, see "Creating user-defined functions" on page 172.

Passing parameters in a URL

ColdFusion lets you pass parameters to CFC methods in a URL. To do so, you append the URL in standard URL query-string, name-value pair syntax; for example:

```
http://localhost:8500/CompanyQuery.cfc?method=getEmp&lastName=Adams
```

CFC variables and scope

This section describes how CFCs interact with ColdFusion scopes and use local variables, including the following topics:

- The This scope
- The Variables scope
- The Arguments scope
- Other variable scopes
- Function local variables

Note: Components also have a Super keyword that is sometimes called a scope. For information on the Super keyword, see "Using the Super keyword" on page 232.

The This scope

The This scope is available within the CFC and is shared by all CFC methods. It is also available in the base component (if the CFC is a child component), on the page that instantiates the CFC, and all CFML pages included by the CFC.

Inside the CFC, you define and access This scope variables by using the prefix This, as in the following line:

```
<cfset This.color="green">
```

In the calling page, you can define and access CFC This scope variables by using the CFC instance name as the prefix. For example, if you create a CFC instance named `car` and, within the `car` CFC specify `<cfset This.color="green">`, a ColdFusion page that instantiates the CFC could refer to the component's color property as `#car.color#`.

Variable values in the This scope last as long as the CFC instance exists and, therefore, can persist between calls to methods of a CFC instance.

Note: The This scope identifier works like the This keyword of JavaScript and ActionScript. CFCs do not follow the Java class model, and the This keyword behaves differently in ColdFusion MX than in Java. In Java, This is a private scope, whereas in ColdFusion, it is a public scope.

The Variables scope

The Variables scope in a CFC is private to the CFC. It includes variables defined in the CFC body (initialization or constructor code) and in the CFC methods. When you set Variables scope variables in the CFC, they cannot be seen by pages that invoke the CFC.

The CFC Variables scope does not include any of the Variables scope variables that are declared or available in the page that instantiates or invokes the CFC. However, you can make the Variables scope of the page that invokes a CFC accessible to the CFC by passing Variables as an argument to the CFC method.

You set a Variables scope variable by assigning a value to a name that has the Variables prefix or no prefix.

Values in the Variables scope last as long as the CFC instance exists, and therefore can last between calls to CFC instance methods.

The Variables scope is available to included pages, and Variables scope variables that are declared in the included page are available in the component page.

Note: The Variables scope is not the same as the var keyword, which makes variables private within a function. You should always define function-local variables using the var keyword.

Example: sharing the Variables scope

The following example shows how to make the Variables scope of the page that invokes a CFC accessible to the CFC by passing Variables as an argument to the CFC method. It also illustrates that the Variables scope is private to the CFC.

The following code is for the callGreetMe.cfm page:

```
<cfset Variables.MyName="Wilson">
<cfobject component="greetMe" name="myGreetings">
<cfoutput>
  Before invoking the CFC, Variables.Myname is: #Variables.MyName#.<br>
  Passing Variables scope to hello method. It returns:
  #myGreetings.hello(Variables.MyName)#.<br>
    After invoking the CFC, Variables.Myname is: #Variables.MyName#.<br>
</cfoutput>
<cfinvoke component="greetMe" method="VarScopeInCfc">
```

The following code is for the greetMe CFC:

```
<cfcomponent>
<cfset Variables.MyName="Tuckerman">
  <cffunction name="hello">
    <cfargument name="Name" Required=true>
    <cfset Variables.MyName="Hello " & Arguments.Name>
    <cfreturn Variables.MyName>
  </cffunction>
  <cffunction name="VarScopeInCfc">
    <cfoutput>Within the VarScopeInCfc method, Variables.MyName is:
  #variables.MyName#<br></cfoutput>
  </cffunction>
</cfcomponent>
```

In this example, the callGreetMe.cfm page does the following:

1. Sets the MyName variable in its Variables scope to *Wilson*.
2. Displays the *Variables.MyName* value.
3. Calls the greetMe CFC and passes its Variables scope as a parameter.
4. Displays the value returned by the greetMe CFC.
5. Displays the *Variables.MyName* value.
6. Invokes the VarScopeInCfc method, which displays the value of *Variables.MyName* within the CFC.

When you browse the callGreetMe.cfm page, the following appears:

```
Before invoking the CFC, Variables.Myname is: Wilson.
Passing Variables scope to hello method. It returns: Hello Wilson.
After invoking the CFC, Variables.Myname is: Wilson.
Within the VarScopeInCfc method, Variables.MyName is: Tuckerman
```

The Arguments scope

The Arguments scope exists only in a method, and is not available outside the method. The scope contains the variables that you passed into the method, including variables that you passed in the following ways:

- As named attributes to the `cfinvoke` tag
- In the `cfargumentcollection` attribute of the `cfinvoke` tag
- In `cfinvokeargument` tags
- As attributes or parameters passed into the method when the method is invoked as a web service, by Flash Remoting, as a direct URL, or by submitting a form

You can access variables in the Arguments scope using structure notation (`Arguments.variablename`), or array notation (`Arguments[1]` or `Arguments["variablename"]`).

The Arguments scope does not persist between calls to CFC methods.

Variables in the Arguments scope are available to pages included by the method.

Other variable scopes

A CFC shares the Form, URL, Request, CGI, Cookie, Client, Session, Application, Server, and Flash scopes with the calling page. Variables in these scopes are also available to all pages that are included by a CFC. These variables do not have any behavior that is specific to CFCs.

Function local variables

Variables that you declare with the Var keyword inside a `cffunction` tag or CFScript `function` definition are available only in the method in which they are defined, and only last from the time the method is invoked until it returns the result. You cannot use the Var keyword outside of function definitions.

Tip: You should always use the Var keyword on variables that are only used inside of the function in which they are declared.

You must define all function local variables at the top of the function definition, before any other CFML code; for example:

```
<cffunction ...>
  <cfset Var testVariable = "this is a local variable">
    <!--- Function code goes here. --->
  <cfreturn myresult>
</cffunction>
```

Any arguments declared with the `cfargument` tag must appear before any variables defined with the `cfset` tag. You can also put any `cfscript` tag first and define variables that you declare with the Var keyword in the script.

Use function local variables if you put the CFC in a persistent scope such as the Session scope, and the function has data that must be freed when the function exits.

Local variables do not persist between calls to CFC methods.

Local variables are available to pages included by the method.

Using CFCs effectively

This section describes the following techniques:

- Structuring and reusing code
- Building secure ColdFusion components
- Using introspection to get information about components

Structuring and reusing code

The following sections provide information about the techniques that ColdFusion MX provides for structuring and reusing component code:

- Using inheritance and the Super keyword
- Using component packages
- Using CFCs in persistent scopes

Using inheritance and the Super keyword

Component inheritance and the Super keyword are two important tools for creating structured, object-oriented ColdFusion components.

Component inheritance lets you create a single base component and reuse this code in multiple subclasses that are derived from the base component. Typically a base component is more general, and subcomponents are typically more specific. Each subclass does not have to redefine the code in the base component, but can override it if necessary.

The Super keyword lets a component that overrides a base component method execute the original base component method. This technique lets your subclassed component override a method without losing the ability to call the original version of the method.

Using component inheritance

Component inheritance lets you import component methods and properties from one component to another component. Inherited components share any component methods or properties that they inherit from other components, and ColdFusion MX initializes instance data in the parent CFC when you instantiate the CFC that extends it.

When using component inheritance, inheritance should define an *is a* relationship between components. For example, a component named president.cfc inherits its methods and properties from manager.cfc, which inherits its methods and properties from employee.cfc. In other words, president.cfc *is a* manager.cfc; manager.cfc *is an* employee.cfc; and president.cfc *is an* employee.cfc.

In this example, employee.cfc is the *base* component; it's the component upon which the others are based. The manager component extends the employee component; it has all the methods and properties of the employee component, and some additional ones. The president component extends the manager component. The president component is called a subcomponent or child component of the manager component, which, in turn, is a child component of the employee component.

To use component inheritance:

1. Create the employee.cfc file with the following content:

```
<cfcomponent>
  <cfset This.basesalary=40*20>
</cfcomponent>
```

2. Create the manager.cfc file with the following content:

```
<cfcomponent extends="employee">
  <cfset This.mgrBonus=40*10>
</cfcomponent>
```

   In the example, the `cfcomponent` tag's `extends` attribute points to the `employee` component.

3. Create the president.cfc file with the following content:

```
<cfcomponent extends="manager">
  <cfset This.prezBonus=40*20>
</cfcomponent>
```

   In the example, the `cfcomponent` tag's `extends` attribute points to the `manager` component.

4. Create the inherit.cfm file with the following content, and save it in the same directory as the components you created in the previous steps:

```
<cfobject name="empObj" component="employee">
<cfobject name="mgrObj" component="manager">
<cfobject name="prezObj" component="president">
<cfoutput>
  An employee's salary is #empObj.basesalary# per week.<br>
  A manager's salary is #mgrObj.basesalary + mgrObj.mgrBonus# per week.<br>
  A president's salalry is #prezObj.basesalary + prezObj.mgrBonus +
    prezObj.PrezBonus# per week.
</cfoutput>
```

When you browse the inherit.cfm file, the `manager` component refers to the `basesalary` defined in employee.cfc, which is the base component; the `president` component refers to both the `basesalary` defined in the `employee` component, and the `mgrBonus` defined in the `manager` component. The `manager` component is the parent class of the `president` component.

Using the component.cfc file

All CFCs automatically extend the ColdFusion WEB-INF/cftags/component.cfc component. (The WEB-INF directory is in the *cf_root*/wwwroot directory on ColdFusion configured with an embedded J2EE server. It is in the *cf_root* directory when you deploy ColdFusion on a J2EE server.) This CFC is distributed as a zero-length file. You can use it for any core methods or properties that you want *all* CFCs in your ColdFusion application server instance to inherit.

Note: If you install a newer version of ColdFusion, the installation procedure replaces the existing component.cfc file with a new version. Therefore, before upgrading to a new version of ColdFusion, you should save any code that you have added to the component.cfc file, and then copy the code into the new component.cfc file.

Using the Super keyword

You use the Super keyword only on CFCs that use the `Extends` attribute to extend another CFC. Unlike ColdFusion scopes, the Super keyword is not used for variables; it is only used for CFC methods, and it is not available on ColdFusion pages that invoke CFCs.

The Super keyword lets you refer to versions of methods that are defined in the CFC that the current component extends. For example, the employee, manager, and president CFCs each contain a `getPaid` method. The manager CFC extends the employee CFC. Therefore, the manager CFC can use the original versions of the overridden `getPaid` method, as defined in the employee CFC, by prefixing the method name with Super.

To use the Super keyword:

1. Create the employee.cfc file with the following content:

```
<cfcomponent>
  <cffunction name="getPaid" returntype="numeric">
    <cfset var salary=40*20>
    <cfreturn salary>
  </cffunction>
</cfcomponent>
```

2. Create the manager.cfc file with the following content:

```
<cfcomponent extends="employee">
  <cffunction name="getPaid" returntype="numeric">
    <cfset var salary=1.5 * Super.getPaid()>
    <cfreturn salary>
  </cffunction>
</cfcomponent>
```

3. Create the president.cfc file with the following content:

```
<cfcomponent extends="manager">
  <cffunction name="getPaid" returntype="numeric">
    <cfset var salary=1.5 * Super.getPaid()>
    <cfreturn salary>
  </cffunction>
</cfcomponent>
```

4. Create the payday.cfm file with the following content, and save it in the same directory as the components that you created in the previous steps:

```
<cfobject name="empObj" component="employee">
<cfobject name="mgrObj" component="manager">
<cfobject name="prezObj" component="president">
<cfoutput>
  <cfoutput>
    An employee earns #empObj.getPaid()#.<br>
    A manager earns #mgrObj.getPaid()#.<br>
    The president earns #prezObj.getPaid()#.
  </cfoutput>
</cfoutput>
```

In this example, each `getPaid` method in a child component invoked the `getPaid` method of its parent component. The child's `getPaid` method then used the salary returned by the parent's `getPaid` method to calculate the appropriate amount.

Included pages can use the Super keyword.

Note: The Super keyword supports only one level of inheritance. If you use multiple levels of inheritance, you can only use the Super keyword to access the current component's immediate parent. The example in this section illustrates handling this limitation by invoking methods in a chain.

Using component packages

Components stored in the same directory are members of a component *package*. Component packages help prevent naming conflicts, and facilitate easy component deployment; for example:

- ColdFusion searches the current directory first for a CFC. If you put two components in a single directory as a package, and one component refers to the other with only the component name, not a qualified path, ColdFusion always searches the package directory first for the component. As a result, if you structure each application's components into a package, your applications can use the same component names without sharing the component code.
- If you use the `access="package"` attribute in a method's `cffunction` tag, access to the method is limited to components in the same package. Components in other packages cannot use this method, even if they specify it with a fully qualified component name. For more information on access security, see "Using access security" on page 235.

To invoke a packaged component method using the cfinvoke tag:

1. In your web root directory, create a directory named *appResources*.
2. In the appResources directory, create a directory named *components*.
3. Copy the tellTime2.cfc file you created in "Invoking methods of a CFC instance" on page 218 and the getUTCTime.cfm file that you created in "Putting executable code in a separate file" on page 207 to the components directory.
4. Create the timeDisplay.cfm file with the following content and save it in your web root directory:

```
<!--- Create the component instance. --->
<cfobject component="appResources.components.tellTime2" name="tellTimeObj">
<!--- Invoke the methods. --->
<cfinvoke component="#tellTimeObj#" method="getLocalTime"
  returnvariable="localTime" >
<cfinvoke component="#tellTimeObj#" method="getUTCTime"
  returnvariable="UTCTime" >
<!--- Display the results. --->
<h3>Time Display Page</h3>
<cfoutput>
  Server's Local Time: #localTime#<br>
  Calculated UTC Time: #UTCTime#
</cfoutput>
```

   You use dot syntax to navigate directory structures. Place the directory name before the component name.
5. Browse the timeDisplay.cfm file in your browser.

The following example shows a CFScript invocation:

```
<cfscript>
helloCFC = createObject("component", "appResources.components.catQuery");
helloCFC.getSaleItems();
</cfscript>
```

The following example shows a URL invocation:

```
http://localhost/appResources/components/catQuery.cfc?method=getSalesItems
```

Using CFCs in persistent scopes

You can put a CFC instance in the Session or Application scope. This way, the component properties continue to exist while the scope persists. For example, you might want to use a CFC for a shopping cart application, where the shopping cart contents must persist for the length of the user's session. If you put the shopping cart CFC in the Session scope, you can use component properties to store the cart contents. For example, the following line creates an instance of the `shoppingCart` component in the Session scope:

```
<cfobject name="Session.myShoppingCart" component="shoppingCart">
```

Code that manipulates persistent scope CFC properties must be locked, just as all other code that manipulates persistent scope properties must be locked. Therefore, you must lock both of the following types of application code:

- Code that directly manipulates properties of a persistent scope CFC instance
- Code that calls methods of a persistent scope CFC instance that manipulate properties of the instance

If you put multiple CFC instances in a single persistent scope, you can create a named lock for each CFC instance. For more information on locking, see Chapter 15, "Using Persistent Data and Locking," on page 339.

Note: Session scope CFCs cannot be serialized, so you cannot use them with clustered sessions; for example, if you want to support session failover among servers.

Building secure ColdFusion components

To restrict access to component methods, ColdFusion components use the following security features:

- Access security
- Role-based security
- Programmatic security

Using access security

CFC access security lets you limit the code that can access the components. You specify the access to a CFC method by specifying the `cffunction access` attribute, as follows:

Type	Description
private	Available only to the component that declares the method and any components that extend the component in which it is defined. This usage is similar to the Java protected keyword, *not* the Java private keyword.
package	Available only to the component that declares the method, components that extend the component, or any other components in the package. A package consists of all components defined in a single directory. For more information on packages, see "Using component packages" on page 233.
public	Available to any locally executing ColdFusion page or component method.
remote	Available to a locally or remotely executing ColdFusion page or component method, or to a local or remote client through a URL, form submission, Flash Remoting MX, or as a web service.

Using role-based security

If you specify a `roles` attribute in a `cffunction` tag, only users who are logged in with one of the specified roles can execute the method. When a user tries to invoke a method that he or she is not authorized to invoke, an exception is returned.

The following example creates a component method that deletes files:

```
<cfcomponent>
  <cffunction
      name="deleteFile" access="remote" roles="admin,manager" output=”no”>
    <cfargument name="filepath" required="yes">
    <cffile action="DELETE" file=#arguments.filepath#>
  </cffunction>
</cfcomponent>
```

In the example, the `cffunction` tag includes the `roles` attribute to specify the user roles allowed to access it. In this example, only users in the role `admin` and `manager` can access the function. Notice that multiple roles are delimited by a comma.

For information on ColdFusion security, including the `cflogin` tag and role-based security in ColdFusion MX, see Chapter 16, “Securing Applications,” on page 373.

Using programmatic security

You can implement your own security within a method to protect resources. For example you can use the ColdFusion function `IsUserInRole` to determine if a user is in particular role, as the following example shows:

```
<cffunction name="foo">
  <cfif IsUserInRole("admin")>
    … do stuff allowed for admin
  <cfelseif IsUserInRole("user")>
    … do stuff allowed for user
  <cfelse>
    <cfoutput>unauthorized access</cfoutput>
    <cfabort>
  </cfif>
</cffunction>
```

Using introspection to get information about components

ColdFusion provides several ways for you to get information about components:

- Requesting a component page from the browser
- Using the ColdFusion component browser
- Using the Dreamweaver MX 2004 component panel
- Using the GetMetaData function

Development teams can use the information about components as up-to-date API reference information.

Note: For information about how to include documentation in CFCs for display using introspection, see “Documenting CFCs” on page 214.

Requesting a component page from the browser

When you access a CFC directly with a web browser without specifying a component method, the following chain of events occurs:

1. The request is redirected to the cfcexplorer.cfc file, which is located in the *cf_root*/wwwroot/CFIDE/componentutils directory.
2. The cfcexplorer component prompts users for the ColdFusion RDS or Administrator password, if necessary.
3. The cfcexplorer component renders an HTML description and returns it to the browser.

The resulting display looks like the following example:

---

tests.cfcs.arithCFC

Component arithCFC (Arithmetic Functions)

Miscellaneous functions for doing arithmetic

hierarchy:	WEB-INF.cftags.component tests.cfcs.arithCFC
path:	C:\cfmx\wwwroot\tests\cfcs\arithCFC.cfc
properties:	something
methods:	add, multiply *

* - private method

Property	Hint	Type	Req.	Implemented In	Default Value
Prop1 (display name here)	I just need this for an example	any		arithCFC	-

`add (Addition Function)`

remote numeric **add** (*numeric* arg1, *numeric* arg2)

Adds integers or floating point numbers

Output: enabled
Parameters:
arg1: numeric, optional, argument 1 - must be numeric
arg2: numeric, optional, arg2

multiply*

private numeric **multiply** (*numeric* argA, *numeric* argB)

Output: enabled
Parameters:
argA: numeric, optional, argA
argB: numeric, optional, argB

Using the ColdFusion component browser

You can also browse the components available in ColdFusion using the component browser, which is located at *cf_root*/wwwroot/CFIDE/componentutils/componentdoc.cfm.

The browser has three panes:

- The upper-left pane lists all CFC packages that ColdFusion can access, and has all components and refresh links.
- The lower-left pane lists CFC component names. When the browser first appears, or when you click the all components link in the upper pane, the lower pane lists all available components. If you click a package name in the upper left pane, the lower pane lists only the components in the package.
- The right pane initially lists the paths of all components. When you click a component name in the lower-left pane, the right pane shows the ColdFusion introspection page, as described in "Requesting a component page from the browser" on page 237.

Using the Dreamweaver MX 2004 component panel

The Dreamweaver MX 2004 Components panel lists all available components, including their methods, method parameters, and properties. The panel's context menu includes options to create a new component, edit the selected component, insert code to invoke the component, or show detailed information on the component or component element. The Get description option shows the ColdFusion introspection page, as described in "Requesting a component page from the browser" on page 237. For more information on viewing and editing CFCs in Dreamweaver MX 2004, see the Dreamweaver MX 2004 online help.

Using the GetMetaData function

The CFML `GetMetaData` function returns a structure that contains all the metadata of a CFC instance. This structure contains substantially more data about the CFC than the `cfdump` tag shows, and includes the following information:

- All attributes to the component tag, including any metadata-only attributes, plus the component path.
- An array of structures that contains complete information on each method (function) in the component. This information describes all attributes, including metadata-only function and parameter attributes.

- Within each function structure, a Parameters element that contains an array of parameters specified by `cfargument` tags. Information on each parameter includes any metadata-only attributes.
- Information about any properties that are specified using the `cfproperty` tag.

To display metadata for a CFC:

1. Create the tellAboutCfcs.cfm file in the same directory as the telltime.cfc file, with the following code:

```
<!--- Create an instance of the component. --->
<cfobject component="tellTime" name="tellTimeObj">
<!--- Create a new structure. --->
<cfset aboutcfc=structNew()>
<!--- Populate the structure with the metadata for the
       tellTimeObj instance of the tellTime CFC. --->
<cfset aboutcfc=GetMetaData(tellTimeObj)>
<cfdump var="aboutcfc">
```

2. View the tellAboutCfcs.cfm file in a browser.

For information on how to specify CFC metadata, including how to use component tags and how to specify metadata-only attributes, see "Documenting CFCs" on page 214.

ColdFusion component example

A number of code examples in *ColdFusion MX Developer's Guide* reuse code, particularly queries. To illustrate the advantages of CFCs, these examples invoke the appropriate method in the CFC that appears in the following example. Although Macromedia recommends using CFCs to create structured, reusable code, some code examples in this manual contain queries within a CFML page, rather than invoking a CFC, in order to clearly illustrate a particular element of ColdFusion.

```
<cfcomponent>
  <cffunction name="allemployees" access="public" output="false"
    returntype="query">
    <cfset var getNames="">
    <cfquery name="getNames" datasource="cfdocexamples">
      SELECT * FROM Employee
    </cfquery>
  </cffunction>

  <cffunction name="namesalarycontract" access="public" output="false"
    returntype="query">
    <cfset var EmpList="">
    <cfquery name="EmpList" datasource="cfdocexamples">
      SELECT Firstname, Lastname, Salary, Contract
      FROM Employee
    </cfquery>
  </cffunction>

  <cffunction name="fullname" access="public" output="false"
    returntype="query">
    <cfset var engquery="">
```

```
    <cfquery name="engquery" datasource="cfdocexamples">
      SELECT FirstName + ' ' + LastName AS FullName
      FROM Employee
    </cfquery>
  </cffunction>

  <cffunction name="bydept" access="public" output="false" returntype="query">
    <cfset var deptquery="">
    <cfquery name="deptquery" datasource="cfdocexamples">
      SELECT Dept_ID, FirstName + ' ' + LastName
      AS FullName
      FROM Employee
      ORDER BY Dept_ID
    </cfquery>
  </cffunction>

  <cffunction name="employeebyURLID" access="public" output="false"
    returntype="query">
    <cfset var GetRecordtoUpdate="">
    <cfquery name="GetRecordtoUpdate" datasource="cfdocexamples">
      SELECT * FROM Employee
      WHERE Emp_ID = #URL.Emp_ID#
    </cfquery>
  </cffunction>

  <cffunction name="deleteemployee" access="public" output="false"
    returntype="void">
    <cfset var DeleteEmployee="">
    <cfquery name="DeleteEmployee" datasource="cfdocexamples">
      DELETE FROM Employee
      WHERE Emp_ID = #Form.Emp_ID#
    </cfquery>
  </cffunction>

  <cffunction name="distinctlocs"  access="public" output="false"
    returntype="void">
    <cfset var GetDepartments="">
    <cfquery name="GetDepartments" datasource="cfdocexamples">
      SELECT DISTINCT Location
      FROM Departmt
    </cfquery>
  </cffunction>
</cfcomponent>
```

CHAPTER 11
Creating and Using Custom CFML Tags

This chapter describes how to create and use custom CFML tags that encapsulate common code.

Contents

Creating custom tags 241
Passing data to custom tags 245
Managing custom tags 249
Executing custom tags 250
Nesting custom tags 253

Creating custom tags

Custom tags let you extend CFML by adding your own tags to the ones supplied with ColdFusion. After you define a custom tag, you can use it on a ColdFusion page just as you would any of the standard CFML tags, such as `cfquery` and `cfoutput`.

You use custom tags to encapsulate your application logic so that it can be referenced from any ColdFusion page. Custom tags allow for rapid application development and code reuse while offering off-the-shelf solutions for many programming chores.

For example, you might create a custom tag, named `cf_happybirthday`, to generate a birthday message. You could then use that tag in a ColdFusion page, as follows:

```
<cf_happybirthday name="Ted Cantor" birthDate="December 5, 1987">
```

When ColdFusion processes the page containing this tag, it could output the message:

```
December 5, 1987 is Ted Cantor's Birthday.
Please wish him well.
```

A custom tag can also have a body and end tag, for example:

```
<cf_happybirthdayMessge name="Ellen Smith" birthDate="June 8, 1993">
  <P> Happy Birthday Ellen!</P>
  <P> May you have many more!</P>
</cf_happybirthdayMessge>
```

This tag could output the message:

```
June 8, 1993 is Ellen Smith's Birthday.
Happy Birthday Ellen!
May you have many more!
```

For more information about using end tags, see "Handling end tags" on page 250.

Creating and calling custom tags

You implement a custom tag with a single ColdFusion page. You then call the custom tag from a ColdFusion page by inserting the prefix `cf_` before the page's file name. The page referencing the custom tag is referred to as the *calling* page.

To create and call a custom tag:

1. Create a ColdFusion page, the custom tag page, that shows the current date:

   ```
   <cfoutput>#DateFormat(Now())#</cfoutput>
   ```

2. Save the file as date.cfm.
3. Create a ColdFusion page, the calling page, with the following content:

   ```
   <html>
   <head>
     <title>Date Custom Tag</title>
   </head>
   <body>

     <!--- Call the custom tag defined in date.cfm --->
     <cf_date>

   </body>
   </html>
   ```

4. Save the file as callingdate.cfm.
5. View callingdate.cfm in your browser.

   This custom tag returns the current date in the format DD-MMM-YY.

As you can see from this example, creating a custom tag in CFML is no different from writing any ColdFusion page. You can use all CFML constructs, as well as HTML. You are free to use any naming convention that fits your development practice. Unique descriptive names make it easy for you and others to find the right tag.

Note: Although tag names in ColdFusion pages are case-insensitive, custom tag filenames must be lowercase on UNIX.

Storing custom tag pages

You must store custom tag pages in any one of the following:

- The same directory as the calling page
- The cfusion\CustomTags directory
- A subdirectory of the cfusion\CustomTags directory
- A directory that you specify in the ColdFusion MX Administrator

To share a custom tag among applications in multiple directories, place it in the cfusion\CustomTags directory. You can create subdirectories to organize custom tags. ColdFusion searches recursively for the Custom Tags directory, stepping down through any existing subdirectories until the custom tag is found.

You might have a situation where you have multiple custom tags with the same name. To guarantee which tag ColdFusion calls, copy it to the same directory as the calling page. Or, use the `cfmodule` tag with the `template` attribute to specify the absolute path to the custom tag. For more information on `cfmodule`, see the next section.

Calling custom tags using the cfmodule tag

You can also use the `cfmodule` tag to call custom tags if you want to specify the location of the custom tag page. The `cfmodule` tag is useful if you are concerned about possible name conflicts when invoking a custom tag, or if the application must use a variable to dynamically call a custom tag at runtime.

You must use either a `template` or `name` attribute in the tag, but you cannot use both. The following table describes the basic `cfmodule` attributes:

Attribute	Description
template	Required if the `name` attribute is not used. Same as the `template` attribute in `cfinclude`. This attribute: • Specifies a path relative to the directory of the calling page. • If the path value is prefixed with "/", ColdFusion searches directories explicitly mapped in the ColdFusion MX Administrator for the included file. Example: `<cfmodule template="../MyTag.cfm">` identifies a custom tag file in the parent directory.
name	Required if the `template` attribute is not used. Use period-separated names to uniquely identify a subdirectory under the CustomTags root directory. Example: `<cfmodule name="MyApp.GetUserOptions">` identifies the file GetUserOptions.cfm in the CustomTags\MyApp directory under the ColdFusion root directory.
attributes	The custom tag's attributes.

For example, the following code specifies to execute the custom tag defined by the mytag.cfm page in the parent directory of the calling page:

```
<cfmodule template="../mytag.cfm">
```

For more information on using the `cfmodule` tag, see *CFML Reference*.

Calling custom tags using the cfimport tag

You can use the `cfimport` tag to import custom tags from a directory as a tag library. The following example imports the tags from the directory myCustomTags:

```
<cfimport prefix="mytags" taglib="myCustomTags">
```

Once imported, you call the custom tags using the prefix that you set when importing, as the following example shows:

```
<mytags:customTagName>
```

where customTagName corresponds to a ColdFusion application page named customTagName.cfm. If the tag takes attributes, you include them in the call:

```
<mytags:custom_tag_name attribute1=val_1 attribute2=val_2>
```

You can also include end tags when calling your custom tags, as the following example shows:

```
<mytags:custom_tag_name attribute1=val_1 attribute2=val_2>
...
</mytags:custom_tag_name>
```

ColdFusion calls the custom tag page twice for a tag that includes an end tag: once for the start tag and once for the end tag. For more information on how ColdFusion handles end tags, and how to write your custom tags to handle them, see "Handling end tags" on page 250.

One of the advantages to using the `cfimport` tag is that you can define a directory structure for your custom tags to organize them by category. For example, you can put all security tags in one directory, and all interface tags in another. You then import the tags from each directory and give them a different prefix:

```
<cfimport prefix="security" taglib="securityTags">
<cfimport prefix="ui" taglib="uiTags">
...
<security:validateUser name="Bob">
...
<ui:greeting name="Bob">
...
```

Reading your code becomes easier because you can identify the location of your custom tags from the prefix.

Securing custom tags

The ColdFusion security framework enables you to selectively restrict access to individual tag files and tag directories. This can be an important safeguard in team development. For details, see *Configuring and Administering ColdFusion MX*.

Accessing existing custom tags

Before creating a custom tag in CFML, you should review the Custom Tag section of the ColdFusion Developer Exchange at http://devex.macromedia.com/developer/gallery/index.cfm. You might find a tag here that does what you want.

Tags are grouped in several broad categories and are downloadable as freeware, shareware, or commercial software. You can view each tag's syntax and usage information. The gallery contains a wealth of background information on custom tags and an online discussion forum for tag topics.

Tag names with the `cf_` preface are CFML custom tags; those with the `cfx_` preface are ColdFusion extensions written in C++. For more information about the CFX tags, see Chapter 12, "Building Custom CFXAPI Tags," on page 259.

If you do not find a tag that meets your specific needs, you can create your own custom tags in CFML.

Passing data to custom tags

To make your custom tags flexible, you will often want to pass data to them for processing. This section describes how to write custom tags that take tag attributes and other data as input from a calling page.

Passing values to and from custom tags

Because custom tags are individual ColdFusion pages, variables and other data are not automatically shared between a custom tag and the calling page. To pass data from the calling page to the custom tag, you can specify attribute name/value pairs in the custom tag, just as you do for normal HTML and CFML tags.

For example, to pass the value of the NameYouEntered variable to the cf_getmd tag, you can call the custom tag as follows:

```
<cf_getmd Name=#NameYouEntered#>
```

To pass multiple attributes to a custom tag, separate them with a space in the tag as follows:

```
<cf_mytag Firstname="Thadeus" Lastname="Jones">
```

In the custom tag, you use the Attributes scope to access attributes passed to the tag. Therefore, in the getmd.cfm page, you refer to the passed attribute as `Attributes.Name`. The mytag.cfm custom tag page refers to the passed attributes as `Attributes.Firstname` and `Attributes.Lastname`.

The custom tag page can also access variables set in the calling page by prefixing the calling page's local variable with Caller. However, this is not the best way to pass information to a custom tag, because each calling page would be required to create variables with the names required by the custom tag. You can create more flexible custom tags by passing parameters using attributes.

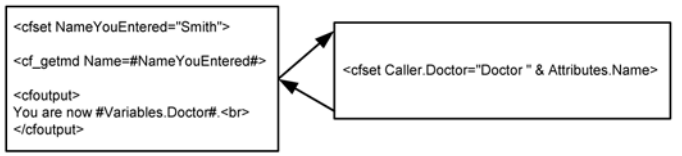
Variables created within a custom tag are deleted when the processing of the tag terminates. Therefore, if you want to pass information back to the calling page, you must write that information back to the Caller scope of the calling page. You cannot access the custom tag's variables outside the custom tag itself.

For example, use the following code in the getmd.cfm page to set the variable Doctor on the calling page:

```
<cfset Caller.Doctor="Doctor " & Attributes.Name>
```

If the variable Doctor does not exist in the calling page, this statement creates it. If the variable exists, the custom tag overwrites it.

The following figure shows the relationship between the variables on the calling page and the custom tag:



One common technique used by custom tags is for the custom tag to take as input an attribute containing the name of the variable to use to pass back results. For example, the calling page passes returnHere as the name of the variable to use to pass back results:

```
<cf_mytag resultName="returnHere">
```

In mytag.cfm, the custom tag passes back its results using the following code:

```
<cfset "Caller.#Attributes.resultName#" = result>
```

Tip: Be careful not to overwrite variables in the calling page from the custom tag. You should adopt a naming convention to minimize the chance of overwriting variables. For example, prefix the returned variable with customtagname_, where customtagname is the name of the custom tag.

Note: Data pertaining to the HTTP request or to the current application is visible in the custom tag page. This includes the variables in the Form, Url, Cgi, Request, Cookies, Server, Application, Session, and Client scopes.

Using tag attributes summary

Custom tag attribute values are passed from the calling page to the custom tag page as name-value pairs. CFML custom tags support required and optional attributes. Custom tag attributes conform to the following CFML coding standards:

- ColdFusion passes any attributes in the Attributes scope.
- Use the `Attributes.`*`attribute_name`* syntax when referring to passed attributes to distinguish them from custom tag page local variables.
- Attributes are case-insensitive.
- Attributes can be listed in any order within a tag.
- Attribute name-value pairs for a tag must be separated by a space in the tag invocation.
- Passed values that contain spaces must be enclosed in double-quotation marks.
- Use the `cfparam` tag with a `default` attribute at the top of a custom tag to test for and assign defaults for optional attributes that are passed from a calling page. For example:

```
<!--- The value of the variable Attributes.Name comes from the calling page.
  If
  the calling page does not set it, make it “Who". --->
<cfparam name="Attributes.Name" default="Who">
```

- Use the `cfparam` tag or a `cfif` tag with an `IsDefined` function at the top of a custom tag to test for required attributes that must be passed from a calling page; for example, the following code issues an abort if the user does not specify the Name attribute to the custom tag:

```
<cfif not IsDefined("Attributes.Name")>
  <cfabort showError="The Name attribute is required.">
</cfif>
```

Custom tag example with attributes

The example in this section creates a custom tag that uses an attribute that is passed to it to set the value of a variable called Doctor on the calling page.

To create a custom tag:

1. Create a new ColdFusion page (the calling page) with the following content:

```
<html>
<head>
  <title>Enter Name</title>
</head>
<body>
<!--- Enter a name, which could also be done in a form --->
<!--- This example simply uses a cfset --->
<cfset NameYouEntered="Smith">

<!--- Display the current name --->
<cfoutput>
Before you leave this page, you're #Variables.NameYouEntered#.<br>
</cfoutput>

<!--- go to the custom tag --->
<cf_getmd Name="#NameYouEntered#">
<!--- Come back from the Custom tag --->

<!--- display the results of the custom tag --->
<cfoutput>
You are now #Variables.Doctor#.<br>
</cfoutput>
</body>
</html>
```

2. Save the page as `callingpage.cfm`.
3. Create another new page (the custom tag) with the following content:

```
<!--- The value of the variable Attributes.Name comes from the calling page.
      If the calling page does not set it, make it “Who". --->
<cfparam name="Attributes.Name" default="Who">

<!--- Create a variable called Doctor, make its value "Doctor "
     followed by the value of the variable Attributes.Name.
     Make its scope Caller so it is passed back to the calling page
--->
<cfset Caller.Doctor="Doctor " & Attributes.Name>
```

4. Save the page as `getmd.cfm`.
5. Open the file `callingpage.cfm` in your browser.

The calling page uses the `getmd` custom tag and displays the results.

Reviewing the code

The following table describes the code and its function:

Code	Description
`<cfset NameYouEntered="Smith">`	In the calling page, create a variable NameYouEntered and assign it the value "Smith."
`<cfoutput>` `Before you leave this page, you're #Variables.NameYouEntered#. ` `</cfoutput>`	In the calling page, display the value of the NameYouEntered variable before calling the custom tag.
`<cf_getmd Name="#NameYouEntered#">`	In the calling page, call the getmd custom tag and pass it the `Name` attribute whose value is the value of the local variable NameYouEntered.
`<cfparam name="Attributes.Name" default="Who">`	The custom tag page normally gets the Name variable in the Attributes scope from the calling page. Assign it the value "Who" if the calling page did not pass an attribute.
`<cfset Caller.Doctor="Doctor " & Attributes.Name>`	In the custom tag page, create a variable called Doctor in the Caller scope so it will exist in the calling page as a local variable. Set its value to the concatenation of the string "Doctor" and the value of the Attributes.Name variable.
`<cfoutput>` `You are now #Variables.Doctor#. ` `</cfoutput>`	In the calling page, display the value of the Doctor variable returned by the custom tag page. (This example uses the Variables scope prefix to emphasize the fact that the variable is returned as a local variable.)

Passing custom tag attributes using CFML structures

You can use the reserved attribute `attributecollection` to pass attributes to custom tags using a structure. The `attributecollection` attribute must reference a structure containing the attribute names as the keys and the attribute values as the values. You can freely mix `attributecollection` with other attributes when you call a custom tag.

The key-value pairs in the structure specified by the `attributecollection` attribute get copied into the custom tag page's Attributes scope. This has the same effect as specifying the `attributecollection` entries as individual attributes when you call the custom tag. The custom tag page refers to the attributes passed using `attributecollection` the same way as it does other attributes; for example, as Attributes.CustomerName or Attributes.Department_number.

Note: You can use both tag attributes and `attributecollections`. If you pass an attribute with the same name using both methods, ColdFusion passes only the tag attribute to the custom tag and ignores the corresponding attribute from the attribute collection.

Custom tag processing reserves the `attributecollection` attribute to refer to the structure holding a collection of custom tag attributes. If `attributecollection` does not refer to such a collection, ColdFusion generates a template exception.

The following example uses an `attributecollection` attribute to pass two of four attributes:

```
<cfset zort=StructNew()>
<cfset zort.x = "-X-">
<cfset zort.y = "-Y-">
<cf_testtwo a="blab" attributecollection=#zort# foo="16">
```

If testtwo.cfm contains the following code:

```
---custom tag ---<br>
<cfoutput>#attributes.a# #attributes.x# #attributes.y#
  #attributes.foo#</cfoutput><br>
--- end custom tag ---
```

its output is the following statement:

```
---custom tag ---
blab -X- -Y- 16
--- end custom tag ---
```

One use for `attributecollection` is to pass the entire Attributes scope of one custom tag to another. This often happens when you have one custom tag that calls a second custom tag and you want to pass all attributes from the first tag to the second.

For example, you call a custom tag with the following code:

```
<cf_first attr1="foo" attr2="bar">
```

To pass all the attributes of the first custom tag to the second, you include the following statement in first.cfm:

```
<cf_second attributecollection="#attributes#">
```

Within the body of second.cfm, you reference the parameters passed to it as follows:

```
<cfoutput>#attributes.attr1#</cfoutput>
<cfoutput>#attributes.attr2#</cfoutput>
```

Managing custom tags

If you deploy custom tags in a multideveloper environment or distribute your tags publicly, you can use the following additional ColdFusion capabilities:

- Advanced security
- Template encoding

Securing custom tags

The ColdFusion security framework enables you to selectively restrict access to individual tags or to tag directories. This can be an important safeguard in team development. For more information, see Chapter 16, “Securing Applications,” on page 373.

Compiling custom tags

You can use the command-line utility `cfcompile` to precompile your custom tag files into Java class files or byte code. For more information, see "Using the cfcompile utility" in Chapter 5, "Deploying ColdFusion Applications" of *Configuring and Administering ColdFusion MX*.

Executing custom tags

The following sections provide information about executing custom tags, including information about handling end tags and processing body text.

Accessing tag instance data

When a custom tag page executes, ColdFusion keeps data related to the tag instance in the `thisTag` structure. You can access the `thisTag` structure from within your custom tag to control processing of the tag. The behavior is similar to the `File` tag-specific variable (sometimes called the File scope).

ColdFusion generates the variables in the following table and writes them to the `thisTag` structure:

Variable	Description
`ExecutionMode`	Contains the execution mode of the custom tag. Valid values are "start", "end", and "inactive".
`HasEndTag`	Distinguishes between custom tags that are called with and without end tags. Used for code validation. If the user specifies an end tag, `HasEndTag` is set to True; otherwise, it is set to False.
`GeneratedContent`	The content that has been generated by the tag. This includes anything in the body of the tag, including the results of any active content, such as ColdFusion variables and functions. You can process this content as a variable.
`AssocAttribs`	Contains the attributes of all nested tags if you use `cfassociate` to make them available to the parent tags. For more information, see "High-level data exchange" on page 255.

The following example accesses the `ExecutionMode` variable of the `thisTag` structure from within a custom tag:

```
<cfif thisTag.ExecutionMode is 'start'>
```

Handling end tags

The examples of custom tags shown so far in this chapter all reference a custom tag using just a start tag, as in:

```
<cf_date>
```

In this case, ColdFusion calls the custom tag page date.cfm to process the tag.

However, you can create custom tags that have both a start and an end tag. For example, the following tag has both a start and an end tag:

```
<cf_date>
  ...
</cf_date>
```

ColdFusion calls the custom tag page date.cfm twice for a tag that includes an end tag: once for the start tag and once for the end tag. As part of the date.cfm page, you can determine if the call is for the start or end tag, and perform the appropriate processing.

ColdFusion will also call the custom tag page twice if you use the shorthand form of an end tag:

```
<cf_date/>
```

You can also call a custom tag using the `cfmodule` tag, as shown in the following example:

```
<cfmodule ...>
  ...
</cfmodule>
```

If you specify an end tag to `cfmodule`, then ColdFusion calls your custom tag as if it had both a start and an end tag.

Determining if an end tag is specified

You can write a custom tag that requires users to include an end tag. If a tag must have an end tag provided, you can use `thisTag.HasEndTag` in the custom tag page to verify that the user included the end tag.

For example, in date.cfm, you could include the following code to determine whether the end tag is specified:

```
<cfif thisTag.HasEndTag is 'False'>
  <!--- Abort the tag--->
  <cfabort showError="An end tag is required.">
</cfif>
```

Determining the tag execution mode

The variable `thisTag.ExecutionMode` contains the mode of invocation of a custom tag page. The variable has one of the following values:

- **Start** Mode for processing the start tag.
- **End** Mode for processing the end tag.
- **Inactive** Mode when the custom tag uses nested tags. For more information, see "Nesting custom tags" on page 253.

If an end tag is not explicitly provided, ColdFusion invokes the custom tag page only once, in Start mode.

A custom tag page named bold.cfm that makes text bold could be written as follows:

```
<cfif thisTag.ExecutionMode is 'start'>
  <!--- Start tag processing --->
  <B>
```

```
<cfelse>
  <!--- End tag processing --->
  </B>
</cfif>
```

You then use this tag to convert the text to bold:

```
<cf_bold>This is bold text</cf_bold>
```

You can also use `cfswitch` to determine the execution mode of a custom tag:

```
<cfswitch expression=#thisTag.ExecutionMode#>
  <cfcase value= 'start'>
    <!--- Start tag processing --->
  </cfcase>
  <cfcase value='end'>
    <!--- End tag processing --->
  </cfcase>
</cfswitch>
```

Considerations when using end tags

How you code your custom tag to divide processing between the start tag and end tag is greatly dependent on the function of the tag. However, you can use the following rules to help you make your decisions:

- Use the start tag to validate input attributes, set default values, and validate the presence of the end tag if it is required by the custom tag.
- Use the end tag to perform the actual processing of the tag, including any body text passed to the tag between the start and end tags. For more information on body text, see "Processing body text" on page 252.
- Perform output in either the start or end tag; do not divide it between the two tags.

Processing body text

Body text is any text that you include between the start and end tags when you call a custom tag; for example:

```
<cf_happybirthdayMessge name="Ellen Smith" birthDate="June, 8, 1993">
  <P> Happy Birthday Ellen!</P>
  <P> May you have many more!</P>
</cf_happybirthdayMessge>
```

In this example, the two lines of code after the start tag are the body text.

You can access the body text within the custom tag using the `thisTag.GeneratedContent` variable. The variable contains all body text passed to the tag. You can modify this text during processing of the tag. The contents of the `thisTag.GeneratedContent` variable are returned to the browser as part of the tag's output.

The `thisTag.GeneratedContent` variable is always empty during the processing of a start tag. Any output generated during start tag processing is not considered part of the tag's generated content.

A custom tag can access and modify the generated content of any of its instances using the `thisTag.GeneratedContent` variable. In this context, the term *generated content* means the results of processing the body of a custom tag. This includes all text and HTML code in the body, the results of evaluating ColdFusion variables, expressions, and functions, and the results generated by descendant tags. Any changes to the value of this variable result in changes to the generated content.

As an example, consider a tag that comments out the HTML generated by its descendants. Its implementation could look like this:

```
<cfif thisTag.ExecutionMode is 'end'>
  <cfset thisTag.GeneratedContent ='<!--#thisTag.GeneratedContent#-->'>
</cfif>
```

Terminating tag execution

Within a custom tag, you typically perform error checking and parameter validation. As part of those checks, you can choose to abort the tag, using `cfabort`, if a required attribute is not specified or other severe error is detected.

The `cfexit` tag also terminates execution of a custom tag. However, the `cfexit` tag is designed to give you more flexibility when coding custom tags than `cfabort`. The `cfexit` tag's `method` attribute specifies where execution continues. The `cfexit` tag can specify that processing continues from the first child of the tag or continues immediately after the end tag marker.

You can also use the `method` attribute to specify that the tag body executes again. This enables custom tags to act as high-level iterators, emulating `cfloop` behavior.

The following table summarizes `cfexit` behavior:

Method attribute value	Location of cfexit call	Behavior
ExitTag (default)	Base page	Acts like `cfabort`
	ExecutionMode=start	Continue after end tag
	ExecutionMode=end	Continue after end tag
ExitTemplate	Base page	Acts like `cfabort`
	ExecutionMode=start	Continue from first child in body
	ExecutionMode=end	Continue after end tag
Loop	Base page	Error
	ExecutionMode=start	Error
	ExecutionMode=end	Continue from first child in body

Nesting custom tags

A custom tag can call other custom tags from within it's body text, thereby *nesting* tags. ColdFusion uses nested tags such as `cfgraph` and `cfgraphdata`, `cfhttp` and `cfhttpparam`, and `cftree` and `cftreeitem`. The ability to nest tags allows you to provide similar functionality.

The following example shows a `cftreeitem` tag nested within a `cftree` tag:

```
<cftree name="tree1"
  required="Yes"
  hscroll="No">
  <cftreeitem value=fullname
    query="engquery"
    queryasroot="Yes"
    img="folder,document">
</cftree>
```

The calling tag is known as an *ancestor*, *parent*, or *base* tag, while the tags that ancestor tags call are known as *descendant*, *child*, or *sub* tags. Together, the ancestor and all descendant tags are called *collaborating* tags.

In order to nest tags, the parent tag must have a closing tag.

The following table lists the terms that describe the relationships between nested tags:

Calling tag	Tag nested within the calling tag	Description
Ancestor	Descendant	An ancestor is any tag that contains other tags between its start and end tags. A descendant is any tag called by a tag.
Parent	Child	Parent and child are synonyms for ancestor and descendant.
Base tag	Sub tag	A base tag is an ancestor that you explicitly associate with a descendant, called a sub tag, with `cfassociate`.

You can create multiple levels of nested tags. In this case, the sub tag becomes the base tag for its own sub tags. Any tag with an end tag present can be an ancestor to another tag.

Nested custom tags operate through three modes of processing, which are exposed to the base tags through the variable `thisTag.ExecutionMode`:

- The *start* mode, in which the base tag is processed for the first time.
- The *inactive* mode, in which sub tags and other code contained within the base tag are processed. No processing occurs in the base tag during this phase.
- The *end* mode, in which the base tag is processed a second time. The end mode occurs when ColdFusion reaches the custom tag's end tag.

Passing data between nested custom tags

A key custom tag feature is for collaborating custom tags to exchange complex data without user intervention, while encapsulating each tag's implementation so that others cannot see it.

When you decide to you use nested tags, you must address the following issues:

- What data should be accessible?
- Which tags can communicate to which tags?
- How are the source and targets of the data exchange identified?
- What CFML mechanism is used for the data exchange?

What data is accessible?

To enable developers to obtain maximum productivity in an environment with few restrictions, CFML custom tags can expose all their data to collaborating tags.

When you develop custom tags, you should document all variables that collaborating tags can access and/or modify. When your custom tags collaborate with other custom tags, you should make sure that they do not modify any undocumented data.

To preserve encapsulation, put all tag data access and modification operations into custom tags. For example, rather than documenting that the variable MyQueryResults in a tag's implementation holds a query result and expecting users to manipulate MyQueryResults directly, create a nested custom tag that manipulates MyQueryResult. This protects the users of the custom tag from changes in the tag's implementation.

Variable scopes and special variables

Use the Request scope for variables in nested tags. The Request scope is available to the base page, all pages it includes, all custom tag pages it calls, and all custom tag pages called by the included pages and custom tag pages. Collaborating custom tags that are not nested in a single tag can exchange data using the request structure. The Request scope is represented as a structure named `Request`.

Where is data accessible?

Two custom tags can be related in a variety of ways in a page. Ancestor and descendant relationships are important because they relate to the order of tag nesting.

A tag's descendants are inactive while the page is executed; that is, the descendent tags have no instance data. A tag, therefore, can only access data from its ancestors, not its descendants. Ancestor data is available from the current page and from the whole runtime tag context stack. The tag context stack is the path from the current tag element up the hierarchy of nested tags, including those in included pages and custom tag references, to the start of the base page for the request. Both `cfinclude` tags and custom tags appear on the tag context stack.

High-level data exchange

While the ability to create nested custom tags is a tremendous productivity gain, keeping track of complex nested tag hierarchies can become a chore. The `cfassociate` tag lets the parent know what the children are up to. By adding this tag to a sub tag, you enable communication of its attributes to the base tag.

In addition, there are many cases in which descendant tags are used only as a means for data validation and exchange with an ancestor tag, such as `cfhttp`/`cfhttpparam` and `cftree`/`cftreeitem`. You can use the `cfassociate` tag to encapsulate this processing.

The `cfassociate` tag has the following format:

```
<cfassociate baseTag="tagName" dataCollection="collectionName">
```

The `baseTag` attribute specifies the name of the base tag that gets access to this tag's attributes. The `dataCollection` attribute specifies the name of the structure in which the base tag stores the sub-tag data. Its default value is AssocAttribs. You only need to specify a `dataCollection` attribute if the base tag can have more than one type of subtag. It is convenient for keeping separate collections of attributes, one per tag type.

Note: If the custom tag requires an end tag, the code processing the structure referenced by the `dataCollection` attribute must be part of end-tag code.

When `cfassociate` is encountered in a sub tag, the sub tag's attributes are automatically saved in the base tag. The attributes are in a structure appended to the end of an array whose name is thisTag.*collectionName*.

The `cfassociate` tag performs the following operations:

```
<!--- Get base tag instance data --->
  <cfset data = getBaseTagData(baseTag)>
  <!--- Create a string with the attribute collection name --->
  <cfset collection_Name = "data.#dataCollection#">
  <!--- Create the attribute collection, if necessary --->
  <cfif not isDefined(collectionName)>
  <cfset #collection_Name# = arrayNew(1)>
  </cfif>
  <!--- Append the current attributes to the array --->
  <cfset temp=arrayAppend(evaluate(collectionName), attributes)>
```

The code accessing sub-tag attributes in the base tag could look like the following:

```
<!--- Protect against no sub-tags --->
  <cfparam Name='thisTag.assocAttribs' default=#arrayNew(1)#>

  <!--- Loop over the attribute sets of all sub tags --->
  <cfloop index=i from=1 to=#arrayLen(thisTag.assocAttribs)#>

  <!--- Get the attributes structure --->
  <cfset subAttribs = thisTag.assocAttribs[i]>
  <!--- Perform other operations --->

  </cfloop>
```

Ancestor data access

The ancestor's data is represented by a structure object that contains all the ancestor's data.

The following functions provide access to ancestral data:

- `GetBaseTagList()` Returns a comma-delimited list of uppercase ancestor tag names, as a string. The first list element is the current tag, the next element is the parent tag name if the current tag is a nested tag. If the function is called for a top-level tag, it returns an empty string.
- `GetBaseTagData(TagName, InstanceNumber=1)` Returns an object that contains all the variables (not just the local variables) of the nth ancestor with a given name. By default, the closest ancestor is returned. If there is no ancestor by the given name, or if the ancestor does not expose any data (such as `cfif`), an exception is thrown.

Example: ancestor data access

This example creates two custom tags and a simple page that calls each of the custom tags. The first custom tag calls the second. The second tag reports on its status and provides information about its ancestors.

To create the calling page:

1. Create a ColdFusion page (the calling page) with the following content:

```
Call cf_nesttag1 which calls cf_nesttag2<br>
<cf_nesttag1>
<hr>

Call cf_nesttag2 directly<br>
<cf_nesttag2>
<hr>
```

2. Save the page as nesttest.cfm.

To create the first custom tag page:

1. Create a ColdFusion page with the following content:

```
<cf_nesttag2>
```

2. Save the page as nesttag1.cfm.

To create the second custom tag page:

1. Create a ColdFusion page with the following content:

```
<cfif thisTag.executionmode is 'start'>
  <!--- Get the tag context stack. The list will look something like
  "MYTAGNAME, CALLINGTAGNAME, ..." --->
  <cfset ancestorlist = getbasetaglist()>

  <!--- Output your own name. You are the first entry in the context stack.
  --->
  <cfoutput>
  <p>I'm custom tag #ListGetAt(ancestorlist,1)#</p>

  <!--- output all the contents of the stack a line at a time --->
  <cfloop index="loopcount" from="1" to="#listlen(ancestorlist)#">
  Ancestorlist entry #loopcount# n is
  #ListGetAt(ancestorlist,loopcount)#<br>
  </cfloop><br>
  </cfoutput>

  <!--- Determine whether you are nested inside a custom tag. Skip the first
  element of the ancestor list, i.e., the name of the custom tag I'm in --->
  <cfset incustomtag = ''>
  <cfloop index="elem"
    list="#listrest(ancestorlist)#"
    <cfif (left(elem, 3) eq 'cf_')>
      <cfset incustomtag = elem>
      <cfbreak>
    </cfif>
  </cfloop>
```

```
  <cfif incustomtag neq ''>
    <!--- Say you are there --->
    <cfoutput>
      I'm running in the context of a custom
      tag named #inCustomTag#.<p>
    </cfoutput>

    <!--- Get the tag instance data --->
    <cfset tagdata = getbasetagdata(incustomtag)>

    <!--- Find out the tag's execution mode --->
    I'm located inside the
    <cfif tagdata.thisTag.executionmode neq 'inactive'>
      custom tag code either because it is in
      its start or end execution mode.
    <cfelse>
      body of the tag
    </cfif>
    <p>
  <cfelse>
    <!--- Say you are lonely --->
    I'm not nested inside any custom tags. :^( <p>
  </cfif>
</cfif>
```

2. Save the page as nesttag2.cfm.
3. Open the file nesttest.cfm in your browser.

CHAPTER 12
Building Custom CFXAPI Tags

Sometimes, the best approach to application development is to develop elements of your application by building executables to run with Macromedia ColdFusion. Perhaps the application requirements go beyond what is currently feasible in CFML. Perhaps you can improve application performance for certain types of processing. Or, you have existing code that already solves an application problem and you want to incorporate it into your ColdFusion application.

To meet these types of requirements, you can use the ColdFusion Extension Application Programming Interface (CFX API) to develop custom ColdFusion tags. This chapter documents custom tag development using Java or C++.

Contents

What are CFX tags? 259

Before you begin developing CFX tags in Java 260

Writing a Java CFX tag 261

ZipBrowser example. 264

Approaches to debugging Java CFX tags 266

Developing CFX tags in C++ 269

What are CFX tags?

ColdFusion Extension (CFX) tags are custom tags written against the ColdFusion Extension Application Programming Interface. Generally, you create a CFX tag if you want to do something that is not possible in CFML, or if you want to improve the performance of a repetitive task.

One common use of CFX tags is to incorporate existing application functionality into a ColdFusion application. That means if you already have the code available, CFX tags make it easy to use it in your application.

CFX tags can do the following:

- Handle any number of custom attributes.
- Use and manipulate ColdFusion queries for custom formatting.

- Generate ColdFusion queries for interfacing with non-ODBC based information sources.
- Dynamically generate HTML to be returned to the client.
- Set variables within the ColdFusion application page from which they are called.
- Throw exceptions that result in standard ColdFusion error messages.

You can build CFX tags using C++ or Java.

Note: ColdFusion MX provides several different techniques to create reusable code, including custom tags. For information on all of these techniques, see Chapter 8, "Creating ColdFusion Elements," on page 161.

Before you begin developing CFX tags in Java

Before you begin developing CFX tags in Java, you must configure your Java development environment. Also, you might want to take a look at some examples before creating your own CFX tags. This section contains information about examples and how to configure your development environment.

Sample Java CFX tags

Before you begin developing a CFX tag in Java, you might want to study sample CFX tags. You can find the Java source files for the examples on Windows in the cfx\java\distrib\examples subdirectory of the main installation directory. On UNIX systems, the files are located in the cfx/java/examples directory. The following table describes the example tags:

Example	Action	Demonstrates
`HelloColdFusion`	Prints a personalized greeting.	The minimal implementation required to create a CFX tag.
`ZipBrowser`	Retrieves the contents of a zip archive.	How to generate a ColdFusion query and return it to the calling page.
`ServerDateTime`	Retrieves the date and time from a network server.	Attribute validation, using numeric attributes, and setting variables within the calling page.
`OutputQuery`	Returns a ColdFusion query in an HTML table.	How to handle a ColdFusion query as input, throw exceptions, and generate dynamic output.
`HelloWorldGraphic`	Generates a "Hello World!" graphic in JPEG format.	How to dynamically create and return graphics from a Java CFX tag.

Setting up your development environment to develop CFX tags in Java

You can use a wide range of Java development environments, including the Java Development Kit (JDK) from Sun, to build Java CFX tags. You can download the JDK from Sun http://java.sun.com/j2se.

Macromedia recommends that you use one of the commercial Java IDEs, so you have an integrated environment for development, debugging, and project management.

Configuring the classpath

To configure your development environment to build Java CFX tags, you must ensure that the supporting classes are visible to your Java compiler. These classes are located in the cfx.jar archive, located in one of the following directories:

Server configuration *cf_root*/wwwroot/WEB-INF/lib

J2EE configuration *cf_webapp_root*/WEB-INF/lib

Consult your Java development tool documentation to determine how to configure the compiler classpath for your particular environment.

The cfx.jar archive contains the classes in the `com.allaire.cfx` package, which are required for developing and deploying Java CFX tags.

When you create new Java CFX tags, you should compile them into the WEB-INF/classes directory. Doing this simplifies your development, debugging, and testing processes.

After you finish with development and testing, you can deploy your Java CFX tag anywhere on the classpath visible to ColdFusion. For more details on customizing the classpath, see "Customizing and configuring Java".

Customizing and configuring Java

Use the JVM and Java Settings page on the ColdFusion MX Administrator Server tab to customize your Java development environment, such as by customizing the classpath and Java system properties, or specifying an alternate JVM. For more information, see the ColdFusion MX Administrator online Help.

Writing a Java CFX tag

To create a Java CFX tag, create a class that implements the `Custom tag interface`. This interface contains one method, `processRequest`, which is passed `Request` and `Response` objects that are then used to do the work of the tag.

The example in the following procedure creates a very simple Java CFX tag named `cfx_MyHelloColdFusion` that writes a text string back to the calling page.

To create a Java CFX tag:

1. Create a new source file in your editor with the following code:

```
import com.allaire.cfx.* ;

public class MyHelloColdFusion implements CustomTag {
   public void processRequest( Request request, Response response )
     throws Exception {
     String strName = request.getAttribute( "NAME" ) ;
     response.write( "Hello, " + strName ) ;
   }
}
```

2. Save the file as MyHelloColdFusion.java in the WEB_INF/classes directory.

3. Compile the java source file into a class file using the Java compiler. If you are using the command-line tools bundled with the JDK, use the following command line, which you execute from within the classes directory:

```
javac -classpath cf_root\WEB-INF\lib\cfx.jar MyHelloColdFusion.java
```

Note: The previous command works only if the Java compiler (`javac.exe`) is in your path. If it is not in your path, specify the fully qualified path; for example, c:\jdk1.3.1_01\bin\javac on Windows or /usr/java/bin/javac on UNIX.

If you receive errors during compilation, check the source code to make sure you entered it correctly. If no errors occur, you successfully wrote your first Java CFX tag. For information on using your new tag in a ColdFusion page, see "Calling the CFX tag from a ColdFusion page" on page 262.

Calling the CFX tag from a ColdFusion page

You call Java CFX tags from within ColdFusion pages by using the name of the CFX tag that is registered on the ColdFusion MX Administrator CFX tags page. This name should be the prefix `cfx_` followed by the class name (without the .class extension).

To register a Java CFX tag in the ColdFusion MX Administrator:

1. In the ColdFusion MX Administrator, select Extensions > CFX Tags to open the CFX Tags page.
2. Click Register Java CFX.
3. Enter the tag name (for example, `cfx_MyHelloColdFusion`).
4. Enter the class name without the .class extension (for example, `MyHelloColdFusion`).
5. (Optional) Enter a description.
6. Click Submit.

You can now call the tag from a ColdFusion page.

To call a CFX tag from a ColdFusion page:

1. Create a ColdFusion page (.cfm) in your editor with the following content to call the HelloColdFusion custom tag:

```
<html>
<body>
  <cfx_MyHelloColdFusion NAME="Les">
</body>
</html>
```

2. Save the file in a directory configured to serve ColdFusion pages. For example, you can save the file as C:\inetpub\wwwroot\cfdocs\testjavacfx.cfm on Windows or /home/docroot/cfdocs/testjavacfx.cfm on UNIX.
3. If you have not already done so, register the CFX tag in the ColdFusion MX Administrator (see "Registering CFX tags" on page 270).
4. Request the page from your browser using the appropriate URL; for example:

   http://localhost/cfdocs/testjavacfx.cfm

ColdFusion processes the page and returns a page that displays the text “Hello, Les.” If an error is returned instead, check the source code to make sure you have entered it correctly.

To delete a CFX tag in the ColdFusion MX Administrator:

1. In the ColdFusion MX Administrator, select Extensions > CFX Tags to open the CFX Tags page.
2. For the tag you want to delete, click the Delete icon in the Controls column of the Registered CFX Tags list.

Processing requests

Implementing a Java CFX tag requires interaction with the `Request` and `Response` objects passed to the `processRequest` method. In addition, CFX tags that need to work with ColdFusion queries also interface with the `Query` object. The `com.allaire.cfx` package, located in the WEB-INF/lib/cfx.jar archive, contains the `Request`, `Response`, and `Query` objects.

This section provides an overview of these object types. For a complete description of these object types, see Chapter 8, “ColdFusion Java CFX Reference,” in *CFML Reference*.

For a complete example Java CFX tag that uses `Request`, `Response`, and `Query` objects, see “ZipBrowser example” on page 264.

Request object

The `Request` object is passed to the `processRequest` method of the `CustomTag` interface. The following table lists the methods of the `Request` object for retrieving attributes, including queries, passed to the tag and for reading global tag settings:

Method	Description
`attributeExists`	Checks whether the attribute was passed to this tag.
`debug`	Checks whether the tag contains the `debug` attribute.
`getAttribute`	Retrieves the value of the passed attribute.
`getAttributeList`	Retrieves a list of all attributes passed to the tag.
`getIntAttribute`	Retrieves the value of the passed attribute as an integer.
`getQuery`	Retrieves the query that was passed to this tag, if any.
`getSetting`	Retrieves the value of a global custom tag setting.

For detailed reference information on each of these interfaces, see *CFML Reference*.

Response object

The Response object is passed to the `processRequest` method of the CustomTag interface. The following table lists the methods of the Response object for writing output, generating queries, and setting variables within the calling page:

Method	Description
`write`	Outputs text to the calling page.
`setVariable`	Sets a variable in the calling page.
`addQuery`	Adds a query to the calling page.
`writeDebug`	Outputs text to the debug stream.

For detailed reference information on each of these interfaces, see *CFML Reference*.

Query object

The Query object provides an interface for working with ColdFusion queries. The following table lists the methods of the Query object for retrieving name, row count, and column names and methods for getting and setting data elements:

Method	Description
`getName`	Retrieves the name of the query.
`getRowCount`	Retrieves the number of rows in the query.
`getColumnIndex`	Retrieves the index of a query column.
`getColumns`	Retrieves the names of the query columns.
`getData`	Retrieves a data element from the query.
`addRow`	Adds a new row to the query.
`setData`	Sets a data element within the query.

For detailed reference information on each of these interfaces, see *CFML Reference*.

Life cycle of Java CFX tags

A new instance of the Java CFX object is created for each invocation of the Java CFX tag. This means that it is safe to store per-request instance data within the members of your `CustomTag` object. To store data and/or objects that are accessible to all instances of your `CustomTag`, use static data members. If you do so, you must ensure that all accesses to the data are thread-safe.

ZipBrowser example

The following example shows the use of the `Request`, `Response`, and `Query` objects. The example uses the java.util.zip package to implement a Java CFX tag called `cfx_ZipBrowser`, which is a zip file browsing tag.

Note: The Java source file that implements cfx_ZipBrowser, ZipBrowser.java, is included in the *cf_root*/cfx/java/distrib/examples (server configuration) or cf_webapp_root/WEB-INF/cfusion/cfx/java/distrib/examples (J2EE configuration) directory. Compile ZipBrowser.java to implement the tag.

The tag's `archive` attribute specifies the fully qualified path of the zip archive to browse. The tag's `name` attribute must specify the query to return to the calling page. The returned query contains three columns: Name, Size, and Compressed.

For example, to query an archive at the path C:\logfiles.zip for its contents and output the results, you use the following CFML code:

```
<cfx_ZipBrowser
  archive="C:\logfiles.zip"
  name="LogFiles" >

<cfoutput query="LogFiles">
#Name#,  #Size#, #Compressed# <BR>
</cfoutput>
```

The Java implementation of ZipBrowser is as follows:

```
import com.allaire.cfx.* ;
import java.util.Hashtable ;
import java.io.FileInputStream ;
import java.util.zip.* ;

public class ZipBrowser implements CustomTag {
   public void processRequest( Request request, Response response )
      throws Exception {
      // validate that required attributes were passed
      if (  !request.attributeExists( "ARCHIVE" ) ||
            !request.attributeExists( "NAME" ) ) {
         throw new Exception(
            "Missing attribute (ARCHIVE and NAME are both " +
            "required attributes for this tag)" ) ;
      }
   // get attribute values
      String strArchive = request.getAttribute( "ARCHIVE" ) ;
      String strName = request.getAttribute( "NAME" ) ;

  // create a query to use for returning the list of files
      String[] columns = { "Name", "Size", "Compressed" } ;
      int iName = 1, iSize = 2, iCompressed = 3 ;
      Query files = response.addQuery( strName, columns ) ;

  // read the zip file and build a query from its contents
      ZipInputStream zin =
         new ZipInputStream( new FileInputStream(strArchive) ) ;
      ZipEntry entry ;
      while ( ( entry = zin.getNextEntry()) != null ) {
         // add a row to the results
         int iRow = files.addRow() ;

         // populate the row with data
         files.setData( iRow, iName,
            entry.getName() ) ;
```

```
            files.setData( iRow, iSize,
               String.valueOf(entry.getSize()) ) ;
            files.setData( iRow, iCompressed,
               String.valueOf(entry.getCompressedSize()) ) ;

            // finish up with entry
            zin.closeEntry() ;
         }

         // close the archive
         zin.close() ;
      }
   }
```

Approaches to debugging Java CFX tags

Java CFX tags are not stand-alone applications that run in their own process, like typical Java applications. Rather, they are created and invoked from an existing process. This makes debugging Java CFX tags more difficult, because you cannot use an interactive debugger to debug Java classes that have been loaded by another process.

To overcome this limitation, you can use one of the following techniques:

- Debug the CFX tag while it is running within ColdFusion MX by outputting the debug information as needed.
- Debug the CFX tag using a Java IDE (Integrated Development Environment) that supports debugging features, such as setting breakpoints, stepping through your code, and displaying variable values.
- Debug the request in an interactive debugger offline from ColdFusion MX using the special com.allaire.cfx debugging classes.

Outputting debugging information

Before using interactive debuggers became the norm, programmers typically debugged their programs by inserting output statements in their programs to indicate information such as variable values and control paths taken. Often, when a new platform emerges, this technique comes back into vogue while programmers wait for more sophisticated debugging technology to develop for the platform.

If you need to debug a Java CFX tag while running against a live production server, this is the technique you must use. In addition to outputting debugging text using the `Response.write` method, you can also call your Java CFX tag with the `debug="On"` attribute. This attribute flags the CFX tag that the request is running in debug mode and therefore should output additional extended debugging information. For example, to call the `HelloColdFusion` CFX tag in debugging mode, use the following CFML code:

```
<cfx_HelloColdFusion name="Robert" debug="On">
```

To determine whether a CFX tag is invoked with the `debug` attribute, use the `Request.debug` method. To write debugging output in a special debugging block after the tag finishes executing, use the `Response.writeDebug` method. For information on using these methods, see Chapter 8, "ColdFusion Java CFX Reference," in *CFML Reference*.

Debugging in a Java IDE

You can use a Java IDE to debug your Java CFX tags. This means you can develop your Java CFX tag and debug it in a single environment.

To use a Java IDE to debug your CFX tag:

1. Start your IDE.
2. In the project properties (or your IDE's project setting), make sure your CFX class is in the *web_root*\WEB-INF\classes directory or in the system classpath.
3. Make sure the libraries *cf_root*/wwwroot/WEB-INF/lib/cfx.jar (*cf_webapp_root*/WEB-INF/lib/cfx.jar in the J2EE configuration) and *cf_root*/runtime/lib/jrun.jar (server configuration only) are included in your classpath.
4. In your project settings, set your main class to jrunx.kernel.JRun and application parameters to `-start default`.
5. Debug your application by setting breakpoints, single stepping, displaying variables, or by performing other debugging actions.

Using the debugging classes

To develop and debug Java CFX tags in isolation from the ColdFusion, you use three special debugging classes that are included in the `com.allaire.cfx` package. These classes lets you simulate a call to the `processRequest` method of your CFX tag within the context of the interactive debugger of a Java development environment. The three debugging classes are the following:

- `DebugRequest` An implementation of the `Request` interface that lets you initialize the request with custom attributes, settings, and a query.
- `DebugResponse` An implementation of the `Response` interface that lets you print the results of a request once it has completed.
- `DebugQuery` An implementation of the `Query` interface that lets you initialize a query with a name, columns, and a data set.

To use the debugging classes:

1. Create a `main` method for your Java CFX class.
2. Within the `main` method, initialize a `DebugRequest` and `DebugResponse`, and a `DebugQuery`. Use the appropriate attributes and data for your test.
3. Create an instance of your Java CFX tag and call its `processRequest` method, passing in the `DebugRequest` and `DebugResponse` objects.
4. Call the `DebugResponse.printResults` method to output the results of the request, including content generated, variables set, queries created, and so on.

After you implement a `main` method as described previously, you can debug your Java CFX tag using an interactive, single-step debugger. Specify your Java CFX class as the `main` class, set breakpoints as appropriate, and begin debugging.

Debugging classes example

The following example demonstrates how to use the debugging classes:

```
import java.util.Hashtable ;
import com.allaire.cfx.* ;

public class OutputQuery implements CustomTag {
   // debugger testbed for OutputQuery
   public static void main(String[] argv) {
      try {
         // initialize attributes
         Hashtable attributes = new Hashtable() ;
         attributes.put( "HEADER", "Yes" ) ;
         attributes.put( "BORDER", "3" ) ;

         // initialize query

         String[] columns =
            { "FIRSTNAME", "LASTNAME", "TITLE" } ;

         String[][] data =  {
            { "Stephen", "Cheng", "Vice President" },
            { "Joe", "Berrey", "Intern" },
            { "Adam", "Lipinski", "Director" },
            { "Lynne", "Teague", "Developer" } }  ;

         DebugQuery query =
            new DebugQuery( "Employees", columns, data ) ;

         // create tag, process debugging request, and print results
         OutputQuery tag = new OutputQuery() ;
         DebugRequest request = new DebugRequest( attributes, query ) ;
         DebugResponse response = new DebugResponse() ;
         tag.processRequest( request, response ) ;
         response.printResults() ;
      }
      catch( Throwable e ) {
         e.printStackTrace() ;
      }
   }

   public void processRequest(Request request, Response response) throws
  Exception {
      // ...code for processing the request...
   }
}
```

Developing CFX tags in C++

The following sections provide information to help you develop CFX tags in C++.

Sample C++ CFX tags

Before you begin development of a CFX tag in C++, you might want to study the two CFX tags included with ColdFusion MX. These examples will help you get started working with the CFXAPI. The two example tags are as follows:

- `CFX_DIRECTORYLIST` Queries a directory for the list of files it contains.
- `CFX_NTUSERDB` (Windows only) Lets you add and delete Windows NT users.

On Windows, these tags are located in the *cf_root*\cfx\examples directory. On UNIX, these tags are in the *cf_root*/coldfusion/cfx/examples directory.

Setting up your C++ development environment

The following compliers generate valid CFX code for UNIX platforms:

Platform	Compiler
Solaris	Sun C++ compiler 5.0 or higher (gcc does not work)
Linux	RedHat 6.2 gcc/egcs 1.1.2 compiler

Before you can use your C++ compiler to build custom tags, you must enable the compiler to locate the CFX API header file, cfx.h. In Windows, you do this by adding the CFX API include directory to your list of global include paths. In Windows, this directory is *cf_root*\cfx\include. On UNIX this directory is *cf_root*/cfx/include. On UNIX, you will need `-I <`*includepath*`>` on your compile line (see the Makefile for the directory list example in the cfx/examples directory).

Compiling C++ CFX tags

CFX tags built in Windows and on UNIX must be thread-safe. Compile CFX tags for Solaris with the `-mt` switch on the Sun compiler.

Locating your C++ library files on UNIX

On UNIX systems, your C++ library files can be in any directory as long as the directory is included in LD_LIBRARY_PATH or SHLIB_PATH (HP-UX only).

Implementing C++ CFX tags

CFX tags built in C++ use the tag request object, represented by the C++ `CCFXRequest class`. This object represents a request made from an application page to a custom tag. A pointer to an instance of a request object is passed to the main procedure of a custom tag. The methods available from the request object let the custom tag accomplish its work. For information about the CFX API classes and members, see Chapter 7, "ColdFusion C++ CFX Reference" in *CFML Reference*.

Note: Calling a non-existent C++ CFX procedure or entry point causes a JVM crash on UNIX.

Debugging C++ CFX tags

After you configure a debugging session, you can run your custom tag from within the debugger, set breakpoints, single-step, and so on.

Debugging in Windows

You can debug custom tags within the Visual C++ environment.

To debug C++ CFX tags in Windows:

1. Build your C++ CFX tag using the debug option.
2. Restart ColdFusion.
3. Start Visual C++.
4. Select Build > Start Debug > AttachProcess.
5. Select jrunsvc.exe.

   Macromedia recommends that you shut down all other Java programs.
6. Execute any ColdFusion page that calls the CFX tag.
7. Select File > Open to open a file in VisualDev in which to set a breakpoint.
8. Set a breakpoint in the CFX project.

   The best place is to put it in `ProcessRequest()`. Next time you execute the page you will hit the breakpoint.

Registering CFX tags

To use a CFX tag in your ColdFusion applications, first register it in the Extensions, CFX Tags page in the ColdFusion MX Administrator.

To register a C++ CFX tag:

1. In the ColdFusion MX Administrator, select Extensions > CFX Tags to open the CFX Tags page.
2. Click Register C++ CFX.
3. Enter the Tag name (for example, `cfx_MyNewTag`).
4. If the Server Library .dll field is empty, enter the filepath.
5. Accept the default Procedure entry.
6. Clear the Keep library loaded box while developing the tag.

   For improved performance, when the tag is ready for production use, you can select this option to keep the DLL in memory.
7. (Optional) Enter a description.
8. Click Submit.

You can now call the tag from a ColdFusion page.

To delete a CFX tag:

1. In the ColdFusion MX Administrator, select Extensions > CFX Tags to open the CFX Tags page.
2. For the tag you want to delete, click the Delete icon in the Controls column of the Registered CFX Tags list.

PART III
Developing CFML Applications

This part describes how to develop ColdFusion applications. It describes the elements of a ColdFusion application and how to structure an application, handle errors, use variables that are shared among pages, lock code segments, and secure your application. It also describes how to create a globalized application, and debug and troubleshoot application problems.

The following chapters are included:

Chapter 13: Designing and Optimizing a ColdFusion Application 275
Chapter 14: Handling Errors 307
Chapter 15: Using Persistent Data and Locking 339
Chapter 16: Securing Applications 373
Chapter 17: Developing Globalized Applications 403
Chapter 18: Debugging and Troubleshooting Applications 423

CHAPTER 13
Designing and Optimizing a ColdFusion Application

This chapter describes the elements that make your ColdFusion pages into an effective Internet application. It provides an overview of application elements, describes how you can structure an application on your server, and provides detailed information on using the Application.cfc and Application.cfm files. It also describes coding methods for optimizing application efficiency.

Contents

About applications 275

Elements of a ColdFusion application 276

Structuring an application 279

Defining the application and its event handlers in Application.cfc 282

Migrating from Application.cfm to Application.cfc 294

Using an Application.cfm page. 295

Optimizing ColdFusion applications 298

About applications

The term *application* can mean many things. An application can be as simple as a guest book or as sophisticated as a full Internet commerce system with catalog pages, shopping carts, and reporting.

n application, however, has a specific meaning in Macromedia ColdFusion MX. A ColdFusion application has the following characteristics:

- It consists of one or more ColdFusion pages that work together and share a common set of resources.
- All pages in the application share an application name and configuration settings as specified in an Application.cfc file or a `cfapplication` tag.
- All pages in the application share variables in the Application scope.
- You can write application-wide event handlers for specific events, such as request start or session end.

What appears to a user to be a single application (for example, a company's website), might consist of multiple ColdFusion MX applications.

ColdFusion MX applications are not J2EE applications. If you do not specify an application name in your Application.cfc file or `cfapplication` tag, however, the Application scope corresponds to the J2EE application servlet context.

ColdFusion MX applications end when the application has been inactive for the application time-out period or the server stops. When the application times out, ColdFusion MX releases all Application scope variables. You must, therefore, select a time-out period that balances the need for clearing Application scope memory and the overhead of recreating the scope. A typical application time-out is two days.

ColdFusion MX applications and sessions are independent of each other. For example, if an application times out while a user's session is active, the session continues and the session context, including the user's Session scope variables, is unaffected by the application ending and restarting.

Although there are no definite rules about how you represent your web application as a ColdFusion application or applications, the following guidelines are useful:

- Application pages share a common general purpose. For example, a web storefront is typically a single ColdFusion application.
- Many, but not necessarily all, pages in a ColdFusion application share data or common code elements, such as a single login mechanism.
- Application pages share a common look and feel, often enforced by using common code elements, such as the same header and footer pages, and a common error message template.

This chapter describes the tools that ColdFusion MX provides to create an application, and presents information on how you can develop and optimize your application.

Elements of a ColdFusion application

Before you develop a ColdFusion application, you must determine how to structure the application and how to handle application-wide needs and issues. In particular, you must consider all of the following:

- The overall application framework
- Reusable application elements
- Shared variables
- Application events and the Application.cfc file
- Application-level settings and functions
- Application security and user identification

The following sections introduce these application elements and provide references to more detailed information.

The application framework

The application framework is the overall structure of the application and how your directory structure and application pages reflect that structure. You can use a single application framework to structure multiple ColdFusion applications into a single website or Internet application. You can structure a ColdFusion application by using many methodologies. For example, the Fusebox application development methodology is one popular framework for developing ColdFusion web applications. (For more information on Fusebox, see www.fusebox.org.)

This chapter does not provide information on how to use or develop a specific application framework. However, it does discuss the tools that ColdFusion MX provides for building your framework, including the Application.cfc file, how an application's directory structure affects the application, and how you can map the directory structure. For more information on mapping the application framework, see "Structuring an application" on page 279.

Note: For one example of an application framework, see "ColdFusion Methodologies for Content Management," available at www.macromedia.com/devnet/server_archive/articles/cf_methodologies_for_content_mgmt.html.

Reusable application elements

ColdFusion provides a variety of reusable elements that you can use to provide commonly used functionality and extend CFML. These elements include the following:

- User-defined functions (UDFs)
- CFML custom tags
- ColdFusion components (CFCs)
- CFX (ColdFusion Extension) tags
- Pages that you include using the `cfinclude` tag

For an overview of these elements, and information about how to choose among them, see Chapter 8, "Creating ColdFusion Elements," on page 161.

Shared variables

The following ColdFusion variable scopes maintain data that lasts beyond the scope of the current HTTP request:

Variable scope	Variables available
Server	To all applications on a server and all clients
Application	To all pages in an application for all clients
Client	For a single client browser over multiple browser sessions in one application
Session	For a single client browser for a single browser session in one application

For more information on using these variables, including how to use locks to ensure that the data they contain remains accurate, see Chapter 15, "Using Persistent Data and Locking," on page 339.

Application events and the Application.cfc file.

In ColdFusion MX 7, Macromedia introduced *application events* and the Application.cfc file, which defines application settings and implements methods to handle the application events. Application events are specific occurrences during the life cycle of an application. Each time one of these events occurs, ColdFusion MX runs the corresponding method in your Application.cfc file (also referred to as the *application CFC*). You can implement application CFC methods to handle the following events:

Event	Trigger
Application start	ColdFusion starts processing the first request for a page in an application that is not running.
Application end	An application time-out setting is reached or the server shuts down.
Session start	A new session is created as a result of a request that is not in an existing session.
Session end	A session time-out setting is reached or, if using J2EE sessions, the user closes the browser.
Request start	ColdFusion receives a request, including HTTP requests, messages to the event gateway, SOAP requests, or Flash Remoting requests.
Request	Immediately after ColdFusion finishes processing the request start event. The handler for this event is intended for use as a filter for the request contents. For more information on the differences between request start and request events, see "Managing requests in Application.cfc" on page 288.
Request end	ColdFusion finishes processing all pages and CFCs for the request.
Exceptions	An exception occurs that is not handled in a try/catch block.

The Application.cfc file can also define application-wide settings, including the application name and whether the application supports Session variables.

For more information on using application events and the Application.cfc file, see "Defining the application and its event handlers in Application.cfc" on page 282.

Other application-level settings and functions

This section describes the techniques used prior to ColdFusion MX 7 to define application-level settings, variables, and functions. Macromedia recommends that you do not use these techniques in new code that you write; instead, you should use the Application.cfc file and its variables and methods, which provide more features and include logical, hierarchical structure.

If you do not have an Application.cfc file, ColdFusion processes the following two pages, if they are available, every time it processes any page in the application:

- The Application.cfm page is processed before each page in the application.
- The OnRequestEnd.cfm page is processed after each page in the application.

Note: UNIX systems are case-sensitive. To ensure that your pages work on UNIX, always capitalize the A in Application.cfm and the O, R, and E in OnRequestEnd.cfm.

The Application.cfm page can define the application. It can contain the `cfapplication` tag that specifies the application name, and code on this page is processed for all pages in the application. This page can define application-level settings, functions, and features.

The OnRequestEnd.cfm page is used in fewer applications than the Application.cfm page. It lets you provide common clean-up code that gets processed after all application pages, or specify dynamic footer pages.

Tip: The OnRequestEnd.cfm page does not execute if the page invokes a `cflocation` tag.

For more information on the Application.cfm and OnRequestEnd.cfm pages, see "Using an Application.cfm page" on page 295. For information on placing these pages in the application directory structure, see "Structuring an application" on page 279.

Note: You can create a ColdFusion application without using an Application.cfc, Application.cfm, or OnRequestEnd.cfm page. However, it is much easier to use the Application.cfm page than to have each page in the application use a `cfapplication` tag and define common application elements.

Application security and user identification

All applications must ensure that malicious users cannot make improper use of their resources. Additionally, many applications require user identification, typically to control the portions of a site that the user can access, to control the operations that the user can perform, or to provide user-specific content. ColdFusion provides the following forms of application security to address these issues:

Resource (file and directory-based) security Limits the ColdFusion resources, such as tags, functions, and data sources that application pages in particular directories can access. You must consider the resource security needs of your application when you design the application directory structure.

User (programmatic) security Provides an authentication (login) mechanism and a role-based authorization mechanism to ensure that users can only access and use selected features of the application. User security also incorporates a user ID, which you can use to customize page content. To implement user security, you include security code, such as the `cflogin` and `cfloginuser` tags, in your application.

For more on implementing security, see Chapter 16, "Securing Applications," on page 373.

Structuring an application

When you design a ColdFusion application, you must structure its contents into directories and files, also known as mapping the directory structure. This activity is an important step in designing a ColdFusion application. Before you start building the application, you must establish a root directory for the application. You can store application pages in subdirectories of the root directory.

The following sections describe how ColdFusion uses application-specific pages and how you can organize your application pages in a directory structure.

How ColdFusion MX finds and process application definition pages

ColdFusion MX uses the following rules to locate and process the Application.cfc, Application.cfm, and OnRequestEnd.cfm pages that define application-specific elements. The way ColdFusion MX locates these files helps determine how you structure an application.

Each time ColdFusion MX processes a page request it does the following:

1. When ColdFusion starts processing the request, it does the following:
   - It searches the page's directory for a file named Application.cfc. If one exists, it creates a new instance of the CFC, processes the initial events, and stops searching. (ColdFusion MX creates a new instance of the CFC and processes its initialization code for each request.)
   - If the requested page's directory does not have an Application.cfc file, it checks the directory for an Application.cfm file. If one exists, ColdFusion MX logically includes the Application.cfm page at the beginning of the requested page and stops searching further.
   - If the requested page's directory does not have an Application.cfc or Application.cfm file, ColdFusion MX searches up the directory tree and checks each directory first for an Application.cfc file and then, if one is not found, for an Application.cfm page, until it reaches the root directory (such as C:\). When it finds an Application.cfc or Application.cfm file, it processes the page and stops searching.
2. ColdFusion MX processes the requested page's contents.
3. When the request ends, ColdFusion MX does the following:
   - If you have an Application.cfc, ColdFusion MX processes the CFC's `onRequestEnd` method and releases the CFC instance.
   - If you do not have an Application.cfc, but do have an Application.cfm page, ColdFusion MX looks for an OnRequestEnd.cfm in the same directory as the Application.cfm page ColdFusion uses for the current page. ColdFusion does not search beyond that directory, so it does not run an OnRequestEnd.cfm page that resides in another directory. Also, the OnRequestEnd.cfm page does not run if there is an error or an exception on the application page, or if the application page executes the `cfabort` or `cfexit` tag.

The following rules determine how ColdFusion MX processes application pages and settings:

- ColdFusion processes only one Application.cfc or Application.cfm page for each request. If a ColdFusion page has a `cfinclude` tag pointing to an additional ColdFusion page, ColdFusion MX does not search for an Application.cfc or Application.cfm page when it includes the additional page.
- If a ColdFusion page has a `cfapplication` tag, it first processes any Application.cfc or Application.cfm, and then processes the `cfapplication` tag. The tag can override the settings from the application files, including the application name and the behaviors set by the `cfapplication` tag attributes.

- You can have multiple Application.cfc files, Application.cfm files, and `cfapplication` tags that use the same application name. In this case, all pages that have the same name share the same application settings and Application scope and can set and get all the variables in this scope. ColdFusion uses the parameter settings of the `cfapplication` tag or the most recently processed file, if the settings, such as the session time-out, differ among the files.

Note: If your application runs on a UNIX platform, which is case-sensitive, you must spell Application.cfc, Application.cfm, and OnRequestEnd.cfm with capital letters, as shown.

Defining the directory structure

Defining an application directory structure with an application-specific root directory has the following advantages:

Development The application is easier to develop and maintain, because the application page files are well-organized.

Portability You can easily move the application to another server or another part of a server without changing any code in the application page files.

Application-level settings Application pages that are under the same directory can share application-level settings and functions.

Security Application pages that are under the same directory can share web server security settings.

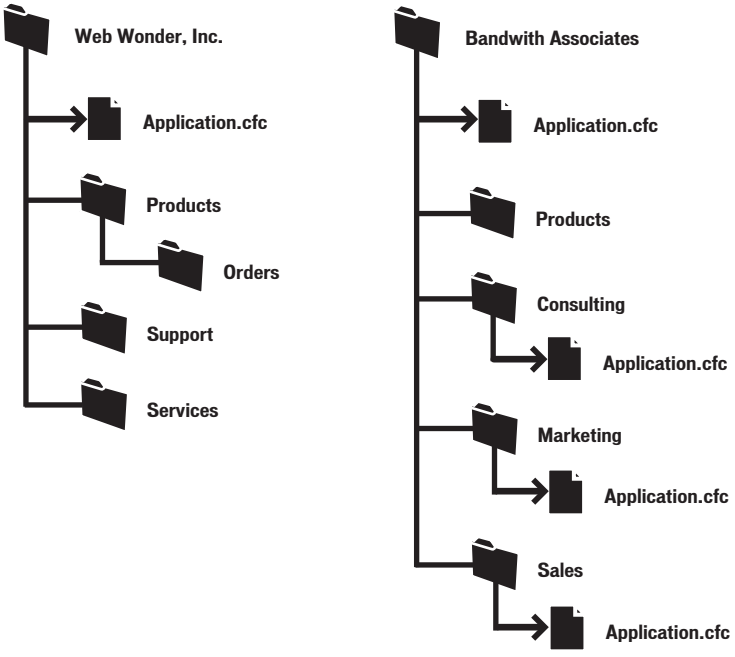
When you put your application in an application-specific directory hierarchy, you can use a single application definition (Application.cfc or Application.cfm) page in the application root directory, or put different application definition pages that govern individual sections of the application in different directories.

You can divide your logical web application into multiple ColdFusion applications by using multiple application definition pages with different application names. Alternatively, you can use multiple application definition pages that specify the same application name, but have different code, for different subsections of your application.

The directory trees in the following figure show two approaches to implementing an application framework:

- In the example on the left, a company named Web Wonders, Inc. uses a single Application.cfc file installed in the application root directory to process all application page requests.

- In the example on the right, Bandwidth Associates uses the settings in individual Application.cfc files to create individual ColdFusion applications at the departmental level. Only the Products application pages are processed using the settings in the root Application.cfc file. The Consulting, Marketing, and Sales directories each have their own Application.cfc file.



Defining the application and its event handlers in Application.cfc

The Application.cfc file defines application-wide settings and variables, and application event handlers:

- Application-wide settings and variables include page processing settings, default variables, data sources, style settings, and other application-level constants.
- Application event handlers are CFC methods that ColdFusion MX automatically executes when specific events occur during the lifetime of an application: application start and end, session start and end, request start, execution, and end, and exceptions.

Defining application-level settings and variables

When you create an application, you can set a number of application-wide properties and characteristics, including the following items:

- Application name
- Application properties, including Client-, Application-, and Session-variable management options
- Page processing options
- Default variables, data sources, style settings, and other application-level constants

This section describes the following topics:

- Naming the application
- Setting application properties
- Setting page processing options

For information on setting default variables, see "You can set default variables and application-level constants in Application.cfc. For example, you can do the following:" on page 287.

Naming the application

Define the application and give it a name by setting the This.name variable in the Application.cfc initialization section, before the method definitions. By using a specific application name, you define a set of pages as part of the same logical application.

ColdFusion supports unnamed applications, which are useful for ColdFusion applications that must interoperate with JSP tags and servlets. Consider creating an unnamed application *only* if your ColdFusion pages must share Application or Session scope data with existing JSP pages and servlets. You cannot have more than one unnamed application on a server instance. For more information on using unnamed applications, see "Sharing data between ColdFusion pages and JSP pages or servlets" on page 923.

Setting application properties

You can specify application properties by setting This scope variables in the Application.cfc initialization code. (These are the same properties that you can set in the `cfapplication` tag.) The following table lists the This scope variables that ColdFusion MX uses to set application properties and describes their uses.

Variable	Default	Description
applicationTimeout	Administrator value	Life span, as a real number of days, of the application, including all Application scope variables. Use the `createTimeSpan` function to generate this variable.
clientManagement	False	Whether the application supports Client scope variables.
clientStorage	Administrator value	Where Client variables are stored; can be cookie, registry, or the name of a data source.
loginStorage	Cookie	Whether to store login information in the Cookie scope or the Session scope.
scriptProtect	Administrator Value	Whether to protect variables from cross-site scripting attacks.
sessionManagement	False	Whether the application supports Session scope variables.
sessionTimeout	Administrator Value	Life span, as a real number of days, of the user session, including all Session variables. Use the `createTimeSpan` function to generate this variable.

Variable	Default	Description
setClientCookies	True	Whether to send CFID and CFTOKEN cookies to the client browser.
setDomainCookies	False	Whether to use domain cookies for the CFID and CFTOKEN values used for client identification, and for Client scope variables stored using cookies. If False, ColdFusion MX uses host-specific cookies. Set to True for applications running on clusters.

The following example code from the top of an Application.cfc sets the application name and properties:

```
<cfcomponent>
<cfset This.name = "TestApplication">
<cfset This.clientmanagement="True">
<cfset This.loginstorage="Session">
<cfset This.sessionmanagement="True">
<cfset This.sessiontimeout="#createtimespan(0,0,10,0)#">
<cfset This.applicationtimeout="#createtimespan(5,0,0,0)#">
```

For more information on these settings, see `cfapplication` in *CFML Reference*.

Setting page processing options

The `cfsetting` tag lets you specify the following page processing attributes that you might want to apply to all pages in your application:

Attribute	Use
showDebugOutput	Specifies whether to show debugging output. This setting cannot enable debugging if it is disabled in the ColdFusion MX Administrator. However, this option can ensure that debugging output is not displayed, even if the Administrator enables it.
requestTimeout	Specifies the page request time-out. If ColdFusion cannot complete processing a page within the time-out period, it generates an error. This setting overrides the setting in the ColdFusion MX Administrator. You can use this setting to increase the page time-out if your application or page frequently accesses external resources that might be particularly slow, such as external LDAP servers or web services providers.
enableCFOutputOnly	Disables output of text that is not included inside `cfoutput` tags. This setting can help ensure that extraneous text that might be in your ColdFusion pages does not get displayed.

Often, you use the `cfsetting` tag on individual pages, but you can also use it in your Application.cfc file. For example, you might use it in multi-application environment to override the ColdFusion MX Administrator settings in one application.

You can put an application-wide `cfsetting` tag in the component initialization code, normally following the This scope application property settings, as the following example shows:

```
<cfcomponent>
<cfscript>
  This.name="MyAppl";
  This.clientmanagement="True";
  This.loginstorage="Session" ;
  This.sessionmanagement="True" ;
  This.sessiontimeout=CreateTimeSpan(0,0,1,0);
</cfscript>
<cfsetting showdebugoutput="No" enablecfoutputonly="No">
```

The `cfsetting` tag in this example affects all pages in an application. You can override the application-wide settings in the event methods, such as `onRequestStart`, or on individual ColdFusion pages.

Using application event handlers

The following table briefly describes the application event CFC methods that you can implement, including when they are triggered. The following sections describe how to use these methods in more detail.

Method	When run
`onApplicationStart`	The application first starts: when the first request for a page is processed or the first CFC method is invoked by an event gateway instance, Flash Remoting request, or a web service invocation. This method is useful for setting application-wide (Application scope) variables, such as the names of data sources.
`onApplicationEnd`	The application ends: when the application times out or the server shuts down.
`onSessionStart`	A new session is created as a result of a request that is not in an existing session, including ColdFusion event gateway sessions. The application must enable sessions for this event to happen.
`onSessionEnd`	A session time-out setting is reached or, if using J2EE sessions, the user closes the browser. This event is **not** triggered when the application ends or the server shuts down.
`onRequestStart`	ColdFusion receives any of the following: a request, an HTTP request (for example, from a browser), a message to an event gateway, a SOAP request, or a Flash Remoting request.
`onRequest`	The `onRequestStart` event has completed. This method can act as a filter for the requested page content.
`onRequestEnd`	All pages and CFCs in the request have been processed: equivalent to the OnRequestEnd.cfm page.
`onError`	When an exception occurs that is not caught by a try/catch block.

When ColdFusion MX receives a request, it instantiates the Application CFC and runs the Application.cfc code in the following order:

- CFC initialization code at the top of the file
- `onApplicationStart`, if not run before for this application
- `onSessionStart`, if not run before for this session
- `onRequestStart`
- `onRequest`, or the requested page if there is no onRequest method
- `onRequestEnd`

The following methods are triggered by specific events:

- `onApplicationEnd`
- `onSessionEnd`
- `onError`

The `onApplicationEnd` and `onSessionEnd` methods do not execute in the context of a page request, so they cannot access request variables or display information to the user. The `OnError` method does not always execute in the context of a request; you can use its `Event` argument to determine the context.

Managing the application with Application.cfc

You use the `onApplicationStart` and `onApplicationEnd` methods to configure and manage the application; that is, to control resources that are used by multiple pages and requests and must be consistently available to all code in your application. Such resources can include data sources, application counters such as page hit variables, or style information for all pages.

The `onApplicationStart` method executes when ColdFusion MX gets the first request for a page in the application after the server starts. The `onApplicationEnd` method executes when the application server shuts down or if the application is inactive for the application time-out period.

The following sections describe some of the ways you can use these methods. For more information, see entries for `onApplicationStart` and `onApplicationEnd` in *CFML Reference*.

Defining application utility functions

Functions that you define in Application.cfc and do not put in a shared scope are, by default, available only to other methods in the CFC.

If your Application.cfc implements the `onRequest` method, any utility functions that you define in Application.cfc are also directly available in to the target page, because Application.cfc and the target page share the Variables scope.

If your application requires utility functions that are used by multiple pages, not just by the methods in Application.cfc, and you do not use an `onRequest` method, Macromedia recommends that you put them in a separate CFC and access them by invoking that CFC. As with other ColdFusion pages, Application.cfc can access any CFC in a directory path that is configured on the ColdFusion MX Administrator Mappings page. You can, therefore, use this technique to share utility functions across applications.

If your Application.cfc defines utility functions that you want available on request pages and does not use an `onRequest` method, you must explicitly put the functions in a ColdFusion MX scope, such as the Request scope, as the following code shows:

```
<cffunction name="theFunctionName" returntype="theReturnType">
  <!--- Function definition goes here. --->
</cffunction>

<cffunction name="OnRequestStart">
  <!--- OnRequestStart body goes here --->
    <cfset Request.theFunctionName=This.theFunctionName>
</cffunction>
```

On the request page, you would include the following code:

```
<cfset myVar=Request.theFunctionName(Argument1...)>
```

Functions that you define in this manner share the This scope and Variables scope with the Application.cfc file for the request.

Setting application default variables and constants in onApplicationStart

You can set default variables and application-level constants in Application.cfc. For example, you can do the following:

- Specify a data source and ensure that it is available
- Specify domain name
- Set styles, such as fonts or colors
- Set other application-level variables

You do not have to lock Application scope variables when you set them in the Application.cfc `onApplicationStart` method.

For details on implementing the `onApplicationStart` method, see `onApplicationStart` in *CFML Reference*.

Using the onApplicationEnd method

Use the `onApplicationEnd` method for any clean-up activities that your application requires when it shuts down, such as saving data in memory to a database, or to log the application end. You cannot use this method to display data on a user page, because it is not associated with any request. The application ends, even if this method throws an exception. An application that is used often is unlikely to execute this method, except when the server is shut down.

For details on implementing the `onApplicationEnd` method, see `onApplicationEnd` in *CFML Reference*.

Managing sessions in Application.cfc

You use the `onSessionStart` and `onSessionEnd` methods to configure and manage user sessions; that is, to control resources that are used by multiple pages while a user is accessing your site from during a single browser session. The session begins when a user first requests a page in your application, and ends either when the session times out or, if you are using J2EE session management, when the user closes the browser when the user closes the browser. For more information on Session scope and Session variables, see Chapter 15, "Using Persistent Data and Locking," on page 339.

Session resources include variables that store data that is needed throughout the session, such as account numbers, shopping cart contents, or CFCs that contain methods and data that are used by multiple pages in a session.

Note: Do not put the `cflogin` tag or basic login processing in the onSessionStart method, as the code executes only at the start of the session; it cannot handle user logouts, and cannot fully ensure security.

The following sections describe some of the ways you can use the `onSessionStart` and `onSessionEnd` methods. For more information, see the `onSessionStart` and `onSessionEnd` entries in *CFML Reference*.

Using the onSessionStart method

This method is useful for initializing session data, such as user settings or shopping cart contents, or for tracking the number of active sessions. You do not need to lock the Session scope when you set session variables in this method.

Using the onSessionEnd method

Use this method for any clean-up activities when the session ends. (For information on ending sessions, see "Ending a session" on page 356.) You can, for example, save session-related data, such as shopping cart contents or information about whether the user has not completed an order, in a database, or you can log the end of the session to a file. You cannot use this method to display data on a user page, because it is not associated with a request.

Note: Sessions do not end, and the `onSessionEnd` method is not called when an application ends. For more information, see the `onSessionEnd` entry in *CFML Reference*.

Managing requests in Application.cfc

ColdFusion MX provides three methods for managing requests: `onRequestStart`, `onRequest`, and `onRequestEnd`. ColdFusion processes requests, including these methods, as follows:

1. ColdFusion always processes `onRequestStart` at the start of the request.
2. If you implement an `onRequest` method, ColdFusion processes it; otherwise, it processes the requested page. If you implement an `onRequest` method, you must explicitly call the requested page in your `onRequest` method.
3. ColdFusion always processes `onRequestEnd` at the end of the request.

The following sections explain how you can use each of the Application.cfc request methods to manage requests. For more information, see entries for `onRequestStart`, `onRequest`, and `onRequestEnd` in *CFML Reference*.

Using the onRequestStart method

This method runs at the beginning of the request. It is useful for user authorization (login handling), and for request-specific variable initialization, such as gathering performance statistics.

If you use the `onRequestStart` method and do not use the `onRequest` method, ColdFusion MX automatically processes the request when it finishes processing the `onRequestStart` code.

Note: If you do not include an `onRequest` method in Application.cfm file, the `onRequestStart` method does not share a Variables scope with the requested page, but it does share Request scope variables.

User authentication

When an application requires a user to log in, put the authentication code, including the `cflogin` tag or code that calls this tag, in the `onRequestStart` method. Doing so ensures that the user is authenticated at the start of each request. For detailed information on security and creating logins, see Chapter 16, "Securing Applications," on page 373. For an example that uses authentication code generated by the Macromedia Dreamweaver CF Login Wizard, see `onRequestStart` in *CFML Reference*.

Using the onRequest method

The `onRequest` method differs from the `onRequestStart` method in one major way: the `onRequest` method intercepts the user's request. This difference has two implications:

- ColdFusion does not process the request unless you explicitly call it, for example, by using a `cfinclude` tag. This behavior lets you use the `onRequest` method to filter requested page content or to implement a switch that determines the pages or page contents to be displayed.
- When you use `cfinclude` to process request, the CFC instance shares the Variables scope with the requested page. As a result, any method in the CFC that executes can set the page's Variables scope variables, and the `onRequestEnd` method can access any Variable scope values that the included page has set or changed. Therefore, for example, the `onRequestStart` or `onRequest` method can set variables that are used on the page.

To use this method as a filter, put the `cfinclude` tag inside a `cfsavecontent` tag, as the following example shows:

```
<cffunction name="onRequest">
  <cfargument name = "targetPage" type="String" required=true/>
  <cfsavecontent variable="content">
    <cfinclude template=#Arguments.targetPage#>
  </cfsavecontent>
  <cfoutput>
    #replace(content, "report", "MyCompany Quarterly Report", "all")#
  </cfoutput>
</cffunction>
```

Using the onRequestEnd method

You use the `onRequestEnd` method for code that should run at the end of each request. (In ColdFusion versions through ColdFusion MX 6.1, you would use the OnRequestEnd.cfm page for such code.) Typical uses include displaying dynamic footer pages. For an example, see `onSessionEnd` in *CFML Reference*.

Note: If you do not include an `onRequest` method in Application.cfm file, the `onRequestEnd` method does not share a Variables scope with the requested page, but it does share Request scope variables.

Handling errors in Application.cfc

The following sections briefly describe how you can handle errors in Application.cfc. For more information on error pages and error handling, see Chapter 14, "Handling Errors," on page 307. For details on implementing the `onError` method, see `onError` in *CFML Reference*.

Application.cfc error handling techniques

Application.cfc can handle errors in any combination of the following ways:

- You can use try/catch error handling in the event methods, such as `onApplicationStart` or `onRequestStart`, to handle exceptions that happen in the event methods.
- You can implement the `onError` method. This method receives all exceptions that are not directly handled by try/catch handlers in CFML code. The method can use the `cfthrow` tag to pass any errors it does not handle to ColdFusion for handling.
- You can use `cferror` tags in the application initialization code following the `cfcomponent` tag, typically following the code that sets the application's This scope variables. These tags specify error processing if you do not implement an `onError` method, or if the `onError` method throws an error. You could implement an application-specific validation error handler, for example, by putting the following tag in the CFC initialization code:

  ```
  <cferror type="VALIDATION" template="validationerrorhandler.cfm">
  ```

- The ColdFusion MX default error mechanisms handle any errors that are not handled by the preceding techniques. These mechanisms include the site-wide error handler that you can specify in the ColdFusion MX Administrator and the built-in default error pages.

These techniques let you include application-specific information, such as contact information or application or version identifiers, in the error message, and let you display all error messages in the application in a consistent manner. You can use Application.cfc to develop sophisticated application-wide error-handling techniques, including error-handling methods that provide specific messages, or use structured error-handling techniques.

Note: The `onError` method can catch errors that occur in the `onSessionEnd` and `onApplicationEnd` application event methods. It will not display messages to the user, however, because there is no request context. The `onError` function can log errors that occur when the session or application ends.

Handling server-side validation errors in the onError method

Server-side validation errors are actually ColdFusion exceptions; as a result, if your application uses an `onError` method, this method gets the error and must handle it or pass it on to ColdFusion for handling.

To identify a server-side validation error, search the Arguments.*Exception*.StackTrace field for coldfusion.filter.FormValidationException. You can then handle the error directly in your `onError` routine, or throw the error so that either the ColdFusion default validation error page or a page specified by an `cferror` tag in your Application.cfc initialization code handles it.

Example: error Handling with the onError method

The following Application.cfc file has an `onError` method that handles errors as follows:

- If the error is a server-side validation error, the `onError` method throws the error for handling by ColdFusion MX, which displays its standard validation error message.
- For any other type of exception, the `onError` method displays the name of the event method in which the error occurred and dumps the exception information. In this example, because you generate errors on the CFM page only, and not in a Application.cfc method, the event name is always the empty string.

```
<cfcomponent>
<cfset This.name = "BugTestApplication">
<cffunction name="onError">
  <!--- The onError method gets two arguments:
       An exception structure, which is identical to a cfcatch variable.
       The name of the Application.cfc method, if any, in which the error
       happened.
  <cfargument name="Except" required=true/>
  <cfargument type="String" name = "EventName" required=true/>
  <!--- Log all errors in an application-specific log file. --->
  <cflog file="#This.Name#" type="error" text="Event Name: #Eventname#" >
  <cflog file="#This.Name#" type="error" text="Message: #except.message#">
  <!--- Throw validation errors to ColdFusion for handling. --->
  <cfif Find("coldfusion.filter.FormValidationException",
            Arguments.Except.StackTrace)>
    <cfthrow object="#except#">
  <cfelse>
    <!--- You can replace this cfoutput tag with application-specific
         error-handling code. --->
    <cfoutput>
      <p>Error Event: #EventName#</p>
      <p>Error details:<br>
      <cfdump var=#except#></p>
    </cfoutput>
  </cfif>
</cffunction>
</cfcomponent>
```

To test this example, put a CFML page with the following code in the same page as the Application.cfc file, and enter valid and invalid text in the text input field.

```
<cfform>
  This box does Integer validation:
  <cfinput name="intinput" type="Text" validate="integer"
  validateat="onServer"><br>
  Check this box to throw an error on the action page:
  <cfinput type="Checkbox" name="throwerror"><br>
  <cfinput type="submit" name="submitit">
 </cfform>
```

```
<cfif IsDefined("form.fieldnames")>
  <cfif IsDefined("form.throwerror")>
    <cfthrow type="ThrownError" message="This error was thrown from the
  bugTest action page.">
  <cfelseif form.intinput NEQ "">
    <h3>You entered the following valid data in the field</h3>
    <cfoutput>#form.intinput#</cfoutput>
  </cfif>
</cfif>
```

Note: For more information on server-side validation errors, see Chapter 28, "Validating Data," on page 659.

Example: a complete Application.cfc

The following example is a simplified Application.cfc file that illustrates the basic use of all application event handlers:

```
<cfcomponent>
<cfset This.name = "TestApplication">
<cfset This.Sessionmanagement=true>
<cfset This.Sessiontimeout="#createtimespan(0,0,10,0)#">
<cfset This.applicationtimeout="#createtimespan(5,0,0,0)#">

<cffunction name="onApplicationStart">
  <cftry>
    <!--- Test whether the DB that this application uses is accessible
        by selecting some data. --->
    <cfquery name="testDB" dataSource="cfdocexamples" maxrows="2">
      SELECT Emp_ID FROM employee
    </cfquery>
    <!--- If we get database error, report it to the user, log the error
      information, and do not start the application. --->
    <cfcatch type="database">
      <cfoutput>
        This application encountered an error.<br>
        Please contact support.
      </cfoutput>
      <cflog file="#This.Name#" type="error"
        text="cfdocexamples DB not available. message: #cfcatch.message#
        Detail: #cfcatch.detail# Native Error: #cfcatch.NativeErrorCode#">
      <cfreturn False>
    </cfcatch>
  </cftry>
  <cflog file="#This.Name#" type="Information" text="Application Started">
  <!--- You do not have to lock code in the onApplicationStart method that sets
  Application scope variables. --->
  <cfscript>
    Application.availableResources=0;
    Application.counter1=1;
    Application.sessions=0;
  </cfscript>
  <!--- You do not need to return True if you don't set the cffunction
  returntype attribute. --->
 </cffunction>
```

```
<cffunction name="onApplicationEnd">
  <cfargument name="ApplicationScope" required=true/>
  <cflog file="#This.Name#" type="Information"
    text="Application #ApplicationScope.applicationname# Ended">
</cffunction>


<cffunction name="onRequestStart">
  <!--- Authentication code, generated by the Dreamweaver Login Wizard,
    makes sure that a user is logged in, and if not displays a login page. --->
  <cfinclude template="mm_wizard_application_include.cfm">
<!--- If it's time for maintenance, tell users to come back later. --->
  <cfscript>
    if ((Hour(now()) gt 1) and (Hour(now()) lt 3)) {
      WriteOutput("The system is undergoing periodic maintenance.
        Please return after 3:00 AM Eastern time.");
      return false;
    } else {
      this.start=now();
    }
  </cfscript>
</cffunction>

<cffunction name="onRequest">
  <cfargument name = "targetPage" type="String" required=true/>
  <cfsavecontent variable="content">
    <cfinclude template=#Arguments.targetPage#>
  </cfsavecontent>
  <!--- This is a minimal example of an onRequest filter. --->
  <cfoutput>
    #replace(content, "report", "MyCompany Quarterly Report", "all")#
  </cfoutput>
</cffunction>

<!--- Display a different footer for logged in users than for guest users or
     users who have not logged in. --->

<cffunction name="onRequestEnd">
  <cfargument type="String" name = "targetTemplate" required=true/>
  <cfset theAuthuser=getauthuser()>
  <cfif ((theAuthUser EQ "guest") OR (theAuthUser EQ ""))>
    <cfinclude template="noauthuserfooter.cfm">
  <cfelse>
    <cfinclude template="authuserfooter.cfm">
  </cfif>
</cffunction>


<cffunction name="onSessionStart">
  <cfscript>
    Session.started = now();
    Session.shoppingCart = StructNew();
    Session.shoppingCart.items =0;
  </cfscript>
```

```
  <cflock timeout="5" throwontimeout="No" type="EXCLUSIVE" scope="SESSION">
    <cfset Application.sessions = Application.sessions + 1>
  </cflock>
  <cflog file="#This.Name#" type="Information" text="Session:
    #Session.sessionid# started">
</cffunction>

<cffunction name="onSessionEnd">
  <cfargument name = "SessionScope" required=true/>
  <cflog file="#This.Name#" type="Information" text="Session:
    #arguments.SessionScope.sessionid# ended">
</cffunction>

<cffunction name="onError">
  <cfargument name="Exception" required=true/>
  <cfargument type="String" name = "EventName" required=true/>
  <!--- Log all errors. --->
  <cflog file="#This.Name#" type="error" text="Event Name: #Eventname#">
  <cflog file="#This.Name#" type="error" text="Message: #exception.message#">
  <!--- Some exceptions, including server-side validation errors, do not
        generate a rootcause structure. --->
  <cfif isdefined("exception.rootcause")>
    <cflog file="#This.Name#" type="error"
      text="Root Cause Message: #exception.rootcause.message#">
  </cfif>
  <!--- Display an error message if there is a page context. --->
  <cfif NOT (Arguments.EventName IS onSessionEnd) OR
      (Arguments.EventName IS onApplicationEnd)>
    <cfoutput>
      <h2>An unexpected error occurred.</h2>
      <p>Please provide the following information to technical support:</p>
      <p>Error Event: #EventName#</p>
      <p>Error details:<br>
      <cfdump var=#exception#></p>
    </cfoutput>
  </cfif>
 </cffunction>

</cfcomponent>
```

Migrating from Application.cfm to Application.cfc

To migrate an existing application that uses Application.cfm to one that uses Application.cfc, do the following:

- Replace the `cfapplication` tag with CFC initialization code that sets the Application.cfc This scope variables that correspond to the tag attributes.
- Put in the `onApplicationStart` method any code that initializes Application scope variables, and any other application-specific code that executes only when the application starts. Often, such code in Application.cfm is inside a block that tests for the existence of an Application scope switch variable. Remove the variable test and the Application scope lock that surrounds the code that sets the Application scope variables.

- Put in the `onSessionStart` method any code that initializes Session scope variables, and any other application-specific code that executes only when the session starts. Remove any code that tests for the existence of Session scope variables to be for initialized and the Session scope lock that surrounds the code that sets the Session scope variables.
- Put in the `onRequestStart` method any `cflogin` tag and related authentication code.
- Put in the `onRequest` method any code that sets Variables scope variables and add a `cfinclude` tag that includes the page specified by the method's Arguments.Targetpage variable.
- Put in the `onRequestEnd` method any code you have in an OnRequestEnd.cfm page.
- Consider replacing `cferror` tags with an `onError` event method. If you do not do so, put the `cferror` tags in the CFC initialization code.

Using an Application.cfm page

If you do not use an Application.cfc file, you can use the Application.cfm page to define application-level settings and functions, as described in the following sections.

Naming the application

Use the `cfapplication` tag to specify the application name and define a set of pages as part of the same logical application. Although you can create an application by putting a `cfapplication` tag with the application name on each page, you normally put the tag in the Application.cfm file; for example:

```
<cfapplication name="SearchApp">
```

Note: The value that you set for the `name` attribute in the `cfapplication` tag is limited to 64 characters.

Setting the client, application, and session variables options

Use the `cfapplication` tag to specify client state and persistent variable use, as follows:

- To use Client scope variables, you must specify `clientManagement=True`.
- To use Session scope variables, you must specify `sessionManagment=True`.

You can also optionally do the following:

- Set application-specific time-outs for Application and Session scope variables. These settings override the default values set in the ColdFusion MX Administrator.
- Specify a storage method for Client scope variables. This setting overrides the method set in the ColdFusion MX Administrator.
- Specify not to use cookies on the client browser.

For more information on configuring these options, see Chapter 15, "Using Persistent Data and Locking," on page 339, and *CFML Reference*.

Defining page processing settings

The `cfsetting` tag lets you specify page processing attributes that you might want to apply to all pages in your application. For more information, see "Setting page processing options" on page 284.

Setting application default variables and constants

You can set default variables and application-level constants on the Application.cfm page. For example, you can specify the following values:

- A data source
- A domain name
- Style settings, such as fonts or colors
- Other important application-level variables

Often, an Application.cfm page uses one or more `cfinclude` tags to include libraries of commonly used code, such as user-defined functions, that are required on many of the application's pages.

Processing logins

When an application requires a user to log in, you typically put the `cflogin` tag on the Application.cfm page. For detailed information on security and creating logins, including an Application.cfm page that manages user logins, see Chapter 16, "Securing Applications," on page 373.

Handling errors

You can use the `cferror` tag on your Application.cfm page to specify application-specific error-handling pages for request, validation, or exception errors, as shown in the example in the following section. This way you can include application-specific information, such as contact information or application or version identifiers, in the error message, and you display all error messages in the application in a consistent manner.

For more information on error pages and error handling, see Chapter 14, "Handling Errors," on page 307.

Example: an Application.cfm page

The following example shows a sample Application.cfm file that uses several of the techniques typically used in Application.cfm pages. For the sake of simplicity, it does not show login processing; for a login example, see Chapter 16, "Securing Applications," on page 373.

```
<!--- Set application name and enable Client and Session variables. --->
<cfapplication name="Products"
  clientmanagement="Yes"
  clientstorage="myCompany"
  sessionmanagement="Yes">
```

```
<!--- Set page processing attributes. --->
<cfsetting showDebugOutput="No">

<!--- Set custom global error handling pages for this application.--->
<cferror type="request"
  template="requesterr.cfm"
  mailto="admin@company.com">
<cferror type="validation"
  template="validationerr.cfm">

<!--- Set the Application variables if they aren't defined. --->
<!--- Initialize local app_is_initialized flag to false. --->
<cfset app_is_initialized = False>
<!--- Get a read-only lock. --->
<cflock scope="application" type="readonly" timeout=10>
<!--- Read init flag and store it in local variable. --->
  <cfset app_is_initialized = IsDefined("Application.initialized")>
</cflock>
<!--- Check the local flag. --->
<cfif not app_is_initialized>
<!--- Application variables are not initialized yet.
    Get an exclusive lock to write scope. --->
  <cflock scope="application" type="exclusive" timeout=10>
    <!--- Check the Application scope initialized flag since another request
      could have set the variables after this page released the read-only
      lock. --->
    <cfif not IsDefined("Application.initialized")>
      <!--- Do initializations --->
      <cfset Application.ReadOnlyData.Company = "MyCompany">
      <!--- and so on --->
      <!--- Set the Application scope initialization flag. --->
      <cfset Application.initialized = "yes">
    </cfif>
  </cflock>
</cfif>

<!--- Set a Session variable.--->
<cflock timeout="20" scope="Session" type="exclusive">
  <cfif not IsDefined("session.pagesHit")>
    <cfset session.pagesHit=1>
  <cfelse>
    <cfset session.pagesHit=session.pagesHit+1>
  </cfif>
</cflock>

<!--- Set Application-specific Variables scope variables. --->
<cfset mainpage = "default.cfm">
<cfset current_page = "#cgi.path_info#?#cgi.query_string#">

<!--- Include a file containing user-defined functions called throughout
    the application. --->
<cfinclude template="commonfiles/productudfs.cfm">
```

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfapplication name="Products"`
`clientmanagement="Yes"`
`clientstorage="myCompany"`
`sessionmanagement="Yes">`
	Names the application, enables Client and Session scope variables, and sets the client variable store to the myCompany data source.

`<cfsetting showDebugOutput="No">`	Ensures that debugging output is not displayed, if the ColdFusion MX Administrator enables it.

`<cferror type="request"`
`template="requesterr.cfm"`
`mailto="admin@company.com">`
`<cferror type="validation"`
`template="validationerr.cfm">`
	Specifies custom error handlers for request and validation errors encountered in the application. Specifies the mailing address for use in the request error handler.

`<cfset app_is_initialized = False>` . . .	Sets the Application scope variables, if they are not already set. For a detailed description of the technique used to set the Application scope variables, see Chapter 15, "Using Persistent Data and Locking," on page 339.

`<cflock timeout="20"`
`scope="Session"`
`type="exclusive">`
`<cfif not`
`IsDefined("session.pagesHit")>`
`<cfset session.pagesHit=1>`
`<cfelse>`
`<cfset session.pagesHit=`
`session.pagesHit+1>`
`</cfif>`
`</cflock>`
	Sets the Session scope `pagesHit` variable, which counts the number of pages touched in this session. If the variable does not exist, creates it; otherwise, increments it.

`<cfset mainpage = "default.cfm">`
`<cfset current_page =`
 	
`"#cgi.path_info#?#cgi.query_string#`
`">`
	Sets two Variables scope variables that are used throughout the application. Creates the current_page variable dynamically; its value varies from request to request.

`<cfinclude template=` `"commonfiles/productudfs.cfm">`	Includes a library of user-defined functions that are used in most pages in the application.

Optimizing ColdFusion applications

You can optimize your ColdFusion application in many ways. Optimizing ColdFusion mostly involves good development and coding practices. For example, good database design and usage is a prime contributor to efficient ColdFusion applications.

In several places, this manual documents optimization techniques as part of the discussion of the related ColdFusion topic. This section provides information about general ColdFusion optimization tools and strategies, and particularly about using CFML caching tags for optimization. This section also contains information on optimizing database use, an important area for application optimization.

The ColdFusion MX Administrator provides caching options for ColdFusion pages and SQL queries. For information on these options, see the ColdFusion MX Administrator online Help and *Configuring and Administering ColdFusion MX*.

For information on debugging techniques that can help you identify slow pages, see Chapter 18, "Debugging and Troubleshooting Applications," on page 423.

For additional information on optimizing ColdFusion, see the Macromedia ColdFusion support center at www.macromedia.com/support/coldfusion.

Caching ColdFusion pages that change infrequently

Some ColdFusion pages produce output that changes infrequently. For example, you might have an application that extracts a vendor list from a database or produces a quarterly results summary. Normally, when ColdFusion gets a request for a page in the application, it does all the business logic and display processing that are required to produce the report or generate and display the list. If the results change infrequently, this can be an inefficient use of processor resources and bandwidth.

The `cfcache` tag tells ColdFusion to cache the HTML that results from processing a page request in a temporary file on the server. This HTML does not need to be generated each time the page is requested. When ColdFusion gets a request for a cached ColdFusion page, it retrieves the pregenerated HTML page without having to process the ColdFusion page. ColdFusion can also cache the page on the client. If the client browser has its own cached copy of the page from a previous viewing, ColdFusion instructs the browser to use the client's page rather than resending the page.

Note: The `cfcache` tag caching mechanism considers that each URL is a separate page. Therefore, http://www.mySite.com/view.cfm?id=1 and http://www.mySite.com/view.cfm?id=2 result in two separate cached pages. Because ColdFusion caches a separate page for each unique set of URL parameters, the caching mechanism accommodates pages for which different parameters result in different output.

Using the cfcache tag

You tell ColdFusion to cache the page results by putting a `cfcache` tag on your ColdFusion page before code that outputs text. The tag lets you specify the following information:

- Whether to cache the page results on the server, the client system, or both. The default is both. The default is optimal for pages that are identical for all users. If the pages contain client-specific information, or are secured with ColdFusion user security, set the `action` attribute in the `cfcache` tag to `ClientCache`.
- The directory on the server in which to store the cached pages. The default directory is *cf_root*/cache. It is a good practice to create a separate cache directory for each application. Doing so can prevent the `cfcache` tag `flush` action from inappropriately flushing more than one application's caches at a time.
- The time span that indicates how long the page lasts in the cache from when it is stored until it is automatically flushed.

You can also specify several attributes for accessing a cached page on the web server, including a user name and password (if required by the web server), the port, and the protocol (HTTP or HTTPS) to use to access the page.

Place the `cfcache` tag before any code on your page that generates output, typically at the top of the page body. For example, the following tag tells ColdFusion to cache the page on both the client and the server. On the server, the page is cached in the e:/temp/page_cache directory. ColdFusion retains the cached page for one day.

```
<cfcache timespan="#CreateTimespan(1, 0, 0, 0)#" directory="e:/temp/
  page_cache">
```

Caution: If an Application.cfm or Application.cfc page displays text (for example, if it includes a header page), use the `cfcache` tag on the Application.cfm page, in addition to the pages that you cache. Otherwise, ColdFusion displays the Application.cfm page output twice on each cached page.

Flushing cached pages

ColdFusion automatically flushes any cached page if you change the code on the page. It also automatically flushes pages after the expiration timespan passes.

You can use the `cfcache` tag with the `action="flush"` attribute to immediately flush one or more cached pages. You can optionally specify the directory that contains the cached pages to be flushed and a URL pattern that identifies the pages to flush. If you do not specify a URL pattern, all pages in the directory are flushed. The URL pattern can include asterisk (*) wildcards to specify parts of the URL that can vary.

When you use the `cfcache` tag to flush cached pages, ColdFusion deletes the pages cached on the server. If a flushed page is cached on the client system, it is deleted, and a new copy gets cached the next time the client tries to access the ColdFusion page.

The following example flushes all the pages in the e:/temp/page_cache/monthly directory that start with HR:

```
<cfcache action="flush" directory="e:/temp/page_cache/monthly" expirURL="HR*">
```

If you have a ColdFusion page that updates data that you use in cached pages, the page that does the updating includes a `cfcache` tag that flushes all pages that use the data.

For more information on the `cfcache` tag, see *CFML Reference*.

Caching parts of ColdFusion pages

In some cases, your ColdFusion page might contain a combination of dynamic information that ColdFusion must generate each time it displays the page, and information that it generates dynamically, but that change less frequently. In this case, you cannot use the `cfcache` tag to cache the entire page. Instead, use the `cfsavecontent` tag to cache the infrequently changed content.

The `cfsavecontent` tag saves the results of processing the tag body in a variable. For example, if the body of the `cfsavecontent` tag contains a `cfexecute` tag that runs an executable program that displays data, the variable saves the output.

You can use the `cfsavecontent` tag to cache infrequently changing output in a shared scope variable. If the information is used throughout the application, save the output in the Application scope. If the information is client-specific, use the Session scope. Because of the overhead of locking shared scope variables, use this technique only if the processing overhead of generating the output is substantial.

Before you use this technique, also consider whether other techniques are more appropriate. For example, query caching eliminates the need to repeat a common query. However, if the effort of processing the data or formatting the output is substantial, using the `cfsavecontent` tag can save processing time.

Using this technique, if the variable exists, the page uses the cached output. If the variable does not exist, the page gets the data, generates the output, and saves the results to the shared scope variable.

The following example shows this technique. It has two parts. The first part welcomes the user and prints out a random lucky number. This part runs and produces a different number each time a user opens the page. The second part performs a database query to get information that changes infrequently; in this example, a listing of the current special sale items. It uses the `cfsavecontent` tag to get the data only when needed.

Tip: If you use this technique frequently, consider incorporating it in a custom CFML tag.

```
<!--- Greet the user. --->
<cfoutput>
  Welcome to our home page.<br>
  The time is #TimeFormat(Now())#.<br>
  Your lucky number is: #RandRange(1,1000)#<br>
  <hr><br>
</cfoutput>

<!--- Set a flag to indicate whether the Application scope variable exists.--->
<cflock scope="application" timeout="20" type="readonly">
  <cfset IsCached = Not IsDefined("Application.ProductCache")>
</cflock>

<!--- If the flag is false, query the DB, and save an image of
  the results output to a variable. --->
<cfif not IsCached>
  <cfsavecontent variable="ProductCache">
  <!--- Perform database query. --->
    <cfquery dataSource="ProductInfo" name="specialQuery">
      SELECT ItemName, Item_link, Description, BasePrice
      FROM SaleProducts
    </cfquery>
<!--- Calculate sale price and display the results. --->
    <h2>Check out the following specials</h2>
    <table>
    <cfoutput query="specialQuery">
      <cfset salePrice= BasePrice * .8>
      <tr>
        <td>#ItemNAme#</td>
        <td>#Item_Link#</td>
```

```
        <td>#Description#</td>
        <td>#salePrice#</td>
      </tr>
    </cfoutput>
    </table>
  </cfsavecontent>

<!--- Save the results in the Application scope. --->
  <cflock scope="Application" type="Exclusive" timeout=30>
    <cfset Application.productCache = ProductCache>
  </cflock>
</cfif>

<!--- Use the Application scope variable to display the sale items. --->
<cflock scope="application" timeout="20" type="readonly">
  <cfoutput>#Application.ProductCache#</cfoutput>
</cflock>
```

Reviewing the code

The following table describes the code and its function:

Code	Description
`<cfoutput>` `  Welcome to our home page. ` `  The time is #TimeFormat(Now())#. ` `  Your lucky number is:` `  #RandRange(1,1000)# ` `  <hr> ` `</cfoutput>`	Displays the part of the page that must change each time.

`<cflock scope="application" timeout="20"`
`  type="readonly">`
`  <cfset IsCached = IsDefined`
`  ("Application.ProductCache")>`
`</cflock>`
	Inside a read-only lock, tests to see if the part of the page that changes infrequently is already cached in the Application scope, and sets a Boolean flag variable with the result.

`<cfif not IsCached>`
`  <cfsavecontent`
`  variable="ProductCache">`
	If the flag is False, uses a `cfsavecontent` tag to save output in a Variables scope variable. Using the Variables scope eliminates the need to do a query (which can take a long time) in an Application scope lock.

`<cfquery dataSource="ProductInfo"`
`  name="specialQuery">`
`  SELECT ItemName, Item_link,`
`  Description, BasePrice`
`  FROM SaleProducts`
`</cfquery>`
	Queries the database to get the necessary information.

Code	Description
Check out the following specials <table> <cfoutput query="specialQuery"> <cfset salePrice = BasePrice * .8> <tr> <td>#ItemNAme#</td> <td>#Item_Link#</td> <td>#Description#</td> <td>#salePrice#</td> </tr> </cfoutput> </table>	Displays the sale items in a table. Inside a cfoutput tag, calculates each item's sale price and displays the item information in a table row. Because this code is inside a cfsavecontent tag, ColdFusion does not display the results of the cfoutput tag. Instead, it saves the formatted output as HTML and text in the ProductCache variable.
</cfsavecontent>	Ends the cfsavecontent tag block.
<cflock scope="Application" type="Exclusive" timeout=30> <cfset Application.productCache = productcache> </cflock>	Inside an Exclusive cflock tag, saves the contents of the local variable ProductCache in the Application scope variable Application.productCache.
</cfif>	Ends the code that executes only if the Application.productCache variable does not exist.
<cflock scope="application" timeout="20" type="readonly"> <cfoutput>#Application.ProductCache#< /cfoutput> </cflock>	Inside a cflock tag, displays the contents of the Application.productCache variable.

Optimizing database use

Poor database design and incorrect or inefficient use of the database are among the most common causes of inefficient applications. Consider the different methods that are available for using databases and information from databases when you design your application. For example, if you need to average the price of a number of products from an SQL query, it is more efficient to use SQL to get the average than to use a loop in ColdFusion.

Two important ColdFusion MX tools for optimizing your use of databases are the cfstoredproc tag and the cfquery tag cachedWithin attribute.

Using stored procedures

The cfstoredproc tag lets ColdFusion MX use stored procedures in your database management system. A stored procedure is a sequence of SQL statements that is assigned a name, compiled, and stored in the database system. Stored procedures can encapsulate programming logic in SQL statements, and database systems are optimized to execute stored procedures efficiently. As a result, stored procedures are faster than cfquery tags.

You use the cfprocparam tag to send parameters to the stored procedure, and the cfprocresult tag to get the record sets that the stored procedure returns.

The following example executes a Sybase stored procedure that returns three result sets, two of which the example uses. The stored procedure returns the status code and one output parameter, which the example displays.

```
<!--- cfstoredproc tag --->
<cfstoredproc procedure = "foo_proc" dataSource = "MY_SYBASE_TEST"
  username = "sa" password = "" returnCode = "Yes">

  <!--- cfprocresult tags --->
  <cfprocresult name = RS1>
  <cfprocresult name = RS3 resultSet = 3>

  <!--- cfprocparam tags --->
  <cfprocparam type = "IN"
    CFSQLType = CF_SQL_INTEGER
      value = "1" dbVarName = @param1>
  <cfprocparam type = "OUT"CFSQLType = CF_SQL_DATE
    variable = FOO dbVarName = @param2>
<!--- Close the cfstoredproc tag. --->
</cfstoredproc>

<cfoutput>
  The output param value: '#foo#'<br>
</cfoutput>

<h3>The Results Information</h3>
<cfoutput query = RS1>
  #name#,#DATE_COL#<br>
</cfoutput>
<br>
<cfoutput>
  <hr>
  Record Count: #RS1.recordCount#<br>
  Columns: #RS1.columnList#<br>
  <hr>
</cfoutput>

<cfoutput query = RS3>
  #col1#,#col2#,#col3#<br>
</cfoutput>
<br>
<cfoutput>
  <hr><br>
  Record Count: #RS3.recordCount#<br>
  Columns: #RS3.columnList#<br>
  <hr>

  The return code for the stored procedure is: '#cfstoredproc.statusCode#'<br>
</cfoutput>
```

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfstoredproc procedure = "foo_proc"`
`  dataSource = "MY_SYBASE_TEST" username = "sa"`
`  password = "" returnCode = "Yes">`
	Runs the stored procedure foo_proc on the MY_SYBASE_TEST data source. Populates the `cfstoredproc statusCode` variable with the status code returned by stored procedure.

`<cfprocresult name = RS1>` `<cfprocresult name = RS3 resultSet = 3>`	Gets two record sets from the stored procedure: the first and third result sets it returns.

`<cfprocparam type = "IN"`
`  CFSQLType = CF_SQL_INTEGER`
`  value = "1" dbVarName = @param1>`
`<cfprocparam type = "OUT" CFSQLType =`
`  CF_SQL_DATE`
`  variable = FOO dbVarName = @param2>`
`</cfstoredproc>`
	Specifies two parameters for the stored procedure, an input parameter and an output parameter. Sets the input parameter to 1 and the ColdFusion variable that gets the output to FOO.
	Ends the `cfstoredproc` tag body.

`<cfoutput>` `  The output param value: '#foo#' ` `</cfoutput>` `<h3>The Results Information</h3>` `<cfoutput query = RS1>` `  #name#,#DATE_COL# ` `</cfoutput>` ` ` `<cfoutput>` `  <hr>` `  Record Count: #RS1.recordCount# ` `  Columns: #RS1.columnList# ` `  <hr>` `</cfoutput>` `<cfoutput query = RS3>` `  #col1#,#col2#,#col3# ` `</cfoutput>` ` ` `<cfoutput>` `  <hr> ` `  Record Count: #RS3.recordCount# ` `  Columns: #RS3.columnList# ` `  <hr>` `  The return code for the stored procedure is:` `  '#cfstoredproc.statusCode#' ` `</cfoutput>`	Displays the results of running the stored procedure: • The output parameter value, • The contents of the two columns in the first record set identified by the name and DATE_COL variables. You set the values of these variables elsewhere on the page. • The number of rows and the names of the columns in the first record set • The contents of the columns in the other record set identified by the col1, col2, and col3 variables. • The number of rows and the names of the columns in the record set. • The status value returned by the stored procedure.

For more information on creating stored procedures, see your database management software documentation. For more information on using the `cfstoredproc` tag, see *CFML Reference*.

Using the cfquery tag cachedWithin attribute

The `cfquery` tag `cachedWithin` attribute tells ColdFusion to save the results of a database query for a specific period of time. This way, ColdFusion accesses the database on the first page request, and does not query the database on further requests until the specified time expires. Using the `cachedWithin` attribute can significantly limit the overhead of accessing databases that do not change rapidly.

This technique is useful if the database contents only change at specific, known times, or if the database does not change frequently and the purpose of the query does not require absolutely up-to-date results.

You must use the `CreateTimeSpan` function to specify the `cachedWithin` attribute value (in days, hours, minutes, seconds format). For example, the following code caches the results of getting the contents of the Employees table of the cfdocexamples data source for one hour.

```
<cfquery datasource="cfdocexamples" name="master"
    cachedWithin=#CreateTimeSpan(0,1,0,0)#>
    SELECT * FROM Employees
  </cfquery>
```

Providing visual feedback to the user

If an application might take a while to process data, it is useful to provide visual feedback to indicate that something is happening, so the user does not assume that there is a problem and request the page again. Although doing this does not optimize your application's processing efficiency, it does make the application appear more responsive.

You can use the `cfflush` tag to return partial data to a user, as shown in Chapter 26, "Introduction to Retrieving and Formatting Data," on page 609.

You can also use the `cfflush` tag to create a progress bar. For information on this technique, see the technical article "Understanding Progress Meters in ColdFusion 5" at www.macromedia.com/v1/handlers/index.cfm?id=21216&method=full. (Although this article was written for ColdFusion 5, it also applies to ColdFusion MX.)

CHAPTER 14
Handling Errors

Macromedia ColdFusion MX includes many tools and techniques for responding to errors that your application encounters. These tools include error handling mechanisms and error logging tools. This chapter describes these tools and how to use them.

This chapter does not discuss techniques for preventing errors, including methods for specifying user input validation. It also does not discuss code debugging. For information on user input validation, see Chapter 26, "Introduction to Retrieving and Formatting Data," on page 609 and Chapter 27, "Building Dynamic Forms with cfform Tags," on page 631. For information on debugging, see Chapter 18, "Debugging and Troubleshooting Applications," on page 423.

Contents

About error handling in ColdFusion 307
Understanding errors 308
Error messages and the standard error format. 314
Determining error-handling strategies 314
Specifying custom error messages with the cferror tag 317
Logging errors with the cflog tag 321
Handling runtime exceptions with ColdFusion tags. 322

About error handling in ColdFusion

By default, ColdFusion generates its own error messages when it encounters errors. In addition, it provides a variety of tools and techniques for you to customize error information and handle errors when they occur. You can use any of the following error-management techniques.

- Specify custom pages for ColdFusion to display in each of the following cases:
  - When a ColdFusion page is missing (the Missing Template Handler page)
  - When an otherwise-unhandled exception error occurs during the processing of a page (the Site-wide Error Handler page)

  You specify these pages on the Settings page in the Server Settings are in the ColdFusion MX Administrator; for more information, see the ColdFusion MX Administrator Help.

- Use the `cferror` tag to specify ColdFusion pages to handle specific types of errors.
- Use the `cftry`, `cfcatch`, `cfthrow`, and `cfrethrow` tags to catch and handle exception errors directly on the page where they occur.
- In CFScript, use the `try` and `catch` statements to handle exceptions.
- Use the `onError` event in Application.cfc to handle exception errors that are not handled by try/catch code on the application pages.
- Log errors. ColdFusion logs certain errors by default. You can use the `cflog` tag to log other errors.

The remaining sections in this chapter provide the following information:

- The basic building blocks for understating types of ColdFusion errors and how ColdFusion handles them
- How to use the `cferror` tag to specify error-handling pages
- How to log errors
- How to handle ColdFusion exceptions

Note: This chapter discusses using the `cftry` and `cfcatch` tags, but not the equivalent CFScript `try` and `catch` statements. The general discussion of exception handling in this chapter applies to tags and CFScript statements. However, the code that you use and the information available in CFScript differs from those in the tags. For more information on handling exceptions in CFScript, see Chapter 9, "Handling errors in UDFs," on page 186.

Understanding errors

There are many ways to look at errors; for example, you can look at errors by their causes. You can also look at them by their effects, particularly by whether your application can recover from them. You can also look at them the way ColdFusion does. The following sections discuss these ways of looking at errors.

About error causes and recovery

Errors can have many causes. Depending on the cause, the error might be *recoverable*. A recoverable error is one for which your application can identify the error cause and take action on the problem. Some errors, such as time-out errors, might be recoverable without indicating to the user that an error was encountered. An error for which a requested application page does not exist is not recoverable, and the application can only display an error message.

Errors such as validation errors, for which the application cannot continue processing the request, but can provide an error-specific response, can also be considered recoverable. For example, an error that occurs when a user enters text where a number is required can be considered recoverable, because the application can recognize the error and redisplay the data field with a message providing information about the error's cause and telling the user to reenter the data.

Some types of errors might be recoverable in some, but not all circumstances. For example, your application can retry a request following a time-out error, but it must also be prepared for the case where the request always times out.

Error causes fall in the broad categories listed in the following table:

Category	Description
Program errors	Can be in the code syntax or the program logic. The ColdFusion compiler identifies and reports program syntax errors when it compiles CFML into Java classes. Errors in your application logic are harder to locate. For information on debugging tools and techniques, see Chapter 18, "Debugging and Troubleshooting Applications," on page 423. Unlike ColdFusion syntax errors, SQL syntax errors are only caught at runtime.
Data errors	Are typically user data input errors. You use validation techniques to identify errors in user input data and enable the user to correct the errors.
System errors	Can come from a variety of causes, including database system problems, time-outs due to excessive demands on your server, out-of-memory errors in the system, file errors, and disk errors.

Although these categories do not map completely to the way ColdFusion categorizes errors they provide a useful way of thinking about errors and can help you in preventing and handling errors in your code.

ColdFusion error types

Before you can effectively manage ColdFusion errors, you must understand how ColdFusion classifies and handles them. ColdFusion categorizes errors as detailed in the following table:

Type	Description
Exception	An error that prevents normal processing from continuing. All ColdFusion exceptions are, at their root, Java exceptions.
Missing template	An HTTP request for a ColdFusion page that cannot be found. Generated if a browser requests a ColdFusion page that does not exist. Missing template errors are different from missing include exceptions, which result from `cfinclude` tags or custom tag calls that cannot find their targets.
Form field data validation	Server-side form field validation errors are a special kind of ColdFusion MX exception. You specify server-side form validation by using `cfform` attributes or hidden HTML form fields. All other types of server-side validation, such as the `cfparam` tag generate runtime exceptions. For more information on validating form fields see Chapter 28, "Validating Data," on page 659. ColdFusion includes a built-in error page for server-side form field validation errors, and the `cferror` tag includes a `type` attribute that lets you handle these errors in a custom error page, but if you use `onError` processing in Application.cfc, or try/catch error handling, the error appears as an Application exception. For more information on handling Form field validation in Application.cfc see "Handling server-side validation errors in the onError method" on page 290.

Note: The onSubmit and onBlur form field validation techniques use JavaScript or Macromedia Flash validation on the client browser; this chapter does not discuss client-side validation errors.

Most ColdFusion errors are exceptions. The following sections describe them in detail.

About ColdFusion exceptions

You can categorize ColdFusion exceptions in two ways:

- When they occur
- Their type

When exceptions occur

ColdFusion errors can occur at two times, when the CFML is compiled into Java and when the resulting Java executes, called runtime exceptions.

Compiler exceptions

Compiler exceptions are programming errors that ColdFusion identifies when it compiles CFML into Java. Because compiler exceptions occur before the ColdFusion page is converted to executable code, you cannot handle them on the page that causes them. However, other pages can handle these errors. For more information, see "Handling compiler exceptions" on page 315

Runtime exception

A runtime exception occurs when the compiled ColdFusion Java code runs. It is an event that disrupts the application's normal flow of instructions. Exceptions can result from system errors or program logic errors. Runtime exceptions include:

- Error responses from external services, such as an ODBC driver or CORBA server
- CFML errors or the results of `cfthrow` or `cfabort` tags
- Internal errors in ColdFusion MX

ColdFusion exception types

ColdFusion exceptions have types that you specify in the `cferror`, `cfcatch`, and `cfthrow` error-handling tags. A `cferror` or `cfcatch` tag will handle only exceptions of the specified type. You identify an exception type by using an identifier from one (or more) of the following type categories:

- Basic
- Custom
- Advanced
- Java class

Note: Use only custom error type names and the `Application` basic type name in `cfthrow` tags. All other built-in exception type names identify specific types of system-identified errors, so you should not use them for errors that you identify yourself.

Basic exception types

All ColdFusion exceptions except for custom exceptions belong to a basic type category. These types consist of a broadly-defined categorization of ColdFusion exceptions. The following table describes the basic exception types:

Type	Type name	Description
Database failures	Database	Failed database operations, such as failed SQL statements, ODBC problems, and so on.
Missing include file errors	MissingInclude	Errors where files specified by the `cfinclude`, `cfmodule`, and `cferror` tags are missing. (A `cferror` tag generates a missingInclude error only when an error of the type specified in the tag occurs.) The `MissingInclude` error type is a subcategory of Template error. If you do not specifically handle the MissingInclude error type, but do handle the Template error type, the Template error handler catches these errors. `MissingInclude` errors are caught at runtime.
Template errors	Template	General application page errors, including invalid tag and attribute names. Most `Template` errors are caught at compile time, not runtime.
Object exceptions	Object	Exceptions in ColdFusion code that works with objects.
Security exceptions	Security	Catchable exceptions in ColdFusion code that works with security.
Expression exceptions	Expression	Failed expression evaluations; for example, if you try to add 1 and "a".
Locking exceptions	Lock	Failed locking operations, such as when a `cflock` critical section times out or fails at runtime.
Verity Search engine exception	SearchEngine	Exceptions generated by the Verity search engine when processing `cfindex`, `cfcollection`, or `cfsearch` tags.
Application-defined exception events raised by `cfthrow`	Application	Custom exceptions generated by a `cfthrow` tag that do not specify a type, or specify the type as `Application`.
All exceptions	Any	Any exceptions. Includes all types in this table and any exceptions that are not specifically handled in another error handler, including unexpected internal and external errors.

Note: The Any type includes all error with the Java object type of java.lang.Exception. It does not include java.lang.Throwable errors. To catch Throwable errors, specify java.lang.Throwable in the `cfcatch` tag `type` attribute.

Custom exceptions

You can generate an exception with your own type by specifying a custom exception type name, for example MyCustomErrorType, in a `cfthrow` tag. You then specify the custom type name in a `cfcatch` or `cferror` tag to handle the exception. Custom type names must be different from any built-in type names, including basic types and Java exception classes.

Advanced exception types

The Advanced exceptions consist of a set of specific, narrow exception types. These types are supported in ColdFusion MX for backward-compatibility. For a list of advanced exception types, see "Advanced Exception types" in Chapter 2, "ColdFusion Tags," in *CFML Reference*.

Java exception classes

Every ColdFusion exception belongs to, and can be identified by, a specific Java exception class in addition to its basic, custom, or advanced type. The first line of the stack trace in the standard error output for an exception identifies the exception's Java class.

For example, if you attempt to use an array function such as `ArrayIsEmpty` on an integer variable, ColdFusion generates an exception that belongs to the `Expression` exception basic type and the `coldfusion.runtime.NonArrayException` Java class.

In general, most applications do not need to use Java exception classes to identify exceptions. However, you can use Java class names to catch exceptions in non-CFML Java objects; for example, the following line catches all Java input/output exceptions:

```
<cfcatch type="java.io.IOException">
```

How ColdFusion handles errors

The following sections describe briefly how ColdFusion handles errors. The rest of this chapter expands on this information.

Missing template errors

If a user requests a page that the ColdFusion cannot find, and the Administrator Server Settings Missing Template Handler field specifies a Missing Template Handler page, ColdFusion uses that page to display error information. Otherwise, it displays a standard error message.

Form field validation errors

When a user enters invalid data in an HTML tag that uses onServer or hidden form field server-side data validation ColdFusion does the following:

1. If the Application CFC (Application.cfc) has an `onError` event handler method, ColdFusion calls the method.
2. If the Application.cfc initialization code or the Application.cfm page has a `cferror` that specifies a Validation error handler, ColdFusion displays the specified error page.
3. Otherwise, it displays the error information in a standard format that consists of a default header, a bulleted list describing the errors, and a default footer.

For more information on using hidden form field validation, see Chapter 28, "Validating Data," on page 659. For more information on Application.cfc, see Chapter 13, "Designing and Optimizing a ColdFusion Application," on page 275

Compiler exception errors

If ColdFusion encounters a compiler exception, how it handles the exception depends on whether the error occurs on a requested page or on an included page:

- If the error occurs on a page that is accessed by a `cfinclude` or `cfmodule` tag, or on a custom tag page that you access using the cf_ notation, ColdFusion handles it as a runtime exception in the page that accesses the tag. See the "Runtime exception errors" section, next, for a description of how these errors are handled.
- If the error occurs directly on the requested page, and the Administrator Settings Site-wide Error Handler field specifies an error handler page, ColdFusion displays the specified error page. Otherwise, ColdFusion reports the error using the standard error message format described in "Error messages and the standard error format" on page 314.

Runtime exception errors

If ColdFusion encounters a runtime exception, it does the action for the *first* matching condition in the following table:

Condition	Action
The code with the error is inside a `cftry` tag and the exception type is specified in a `cfcatch` tag.	Executes the code in the `cfcatch` tag. If the `cftry` block does not have a `cfcatch` tag for this error, tests for an appropriate `cferror` handler or site-wide error handler.
The ColdFusion MX application has an Application.cfc with an `onError` method	Executes the code in the `onError` method. For more information on using the `onError` method, see "Handling errors in Application.cfc" on page 290
A `cferror` tag specifies an exception error handler for the exception type.	Uses the error page specified by the `cferror` tag.
The Administrator Settings Site-wide Error Handler field specifies an error handler page.	Uses the custom error page specified by the Administrator setting.
A `cferror` tag specifies a Request error handler.	Uses the error page specified by the `cferror` tag.
The default case.	Uses the standard error message format

For example, if an exception occurs in CFML code that is not in a `cftry` block, and Application.cfc does not have an `onError` method, but a `cferror` tag specifies a page to handle this error type, ColdFusion uses the specified error page.

Error messages and the standard error format

If your application does not handle an error, ColdFusion displays a diagnostic message in the user's browser.

Error information is also written to a log file for later review. (For information on error logging, see "Logging errors with the cflog tag" on page 321.)

The standard error format consists of the information listed in the following table. ColdFusion does not always display all sections.

Section	Description
Error description	A brief, typically on-line, description of the error.
Error message	A detailed description of the error. The error message diagnostic information displayed depends on the type of error. For example, if you specify an invalid attribute for a tag, this section includes a list of all valid tag attributes.
Error location	The page and line number where ColdFusion encountered the error, followed by a short section of your CFML that includes the line. This section does not display for all errors. In some cases, the cause of an error can be several lines above the place where ColdFusion determines that there is a problem, so the line that initially causes the error might not be in the display.
Resources	Links to documentation, the Knowledge Base, and other resources that can help you resolve the problem.
Error environment information	Information about the request that caused the error. All error messages include the following: • User browser • User IP address • Date and time of error
Stack trace	The Java stack at the time of the exception, including the specific Java class of the exception. This section can be helpful if you must contact Macromedia Technical Support. The stack trace is collapsed by default. Click the heading to display the trace.

Tip: If you get a message that does not explicitly identify the cause of the error, check the key system parameters, such as available memory and disk space.

Determining error-handling strategies

ColdFusion provides you with many options for handling errors, particularly exceptions, as described in the section "How ColdFusion handles errors" on page 312. This section describes considerations for determining which forms of error handling to use.

Handling missing template errors

Missing template errors occur when ColdFusion receives an HTTP request for a page ending in .cfm that it cannot find. You can create your own missing template error page to present application-specific information or provide an application-specific appearance. You specify the missing template error page on the Administrator Settings page.

The missing error page can use CFML tags and variables. In particular, you can use the CGI.script_name variable in text such as the following to identify the requested page:

```
<cfoutput>The page #Replace(CGI.script_name, "/", "")# is not available.<br>
Make sure that you entered the page correctly.<br>
</cfoutput>
```

(In this code, the `Replace` function removes the leading slash sign from the script name to make the display more friendly.)

Handling form field validation errors

When you use server-side form field validation, the default validation error message describes the error cause plainly and clearly. However, you might want to give the error message a custom look or provide additional information such as service contact phone numbers and addresses. In this case, use the `cferror` tag with the `Validation` attribute in the Application.cfc initialization code or on the Application.cfm page to specify your own validation error handler. The section "Example of a validation error page" on page 320 provides an example of such a page. You can also put form field validation error handling code in the Application.cfc `onError` method.

Handling compiler exceptions

You cannot handle compiler exceptions directly on the page where they occur, because the exception is caught before ColdFusion starts running the page code. You should fix all compiler exceptions as part of the development process. Use the reported error message and the code debugging techniques discussed in Chapter 18, "Debugging and Troubleshooting Applications," on page 423 to identify and correct the cause of the error.

Compiler exceptions that occur on pages you access by using the `cfinclude` or `cfmodule` tags can actually be handled as runtime errors by surrounding the `cfinclude` or `cfmodule` tag in a `cftry` block. The compiler exception on the accessed page gets caught as a runtime error on the base page. However, you should avoid this "solution" to the problem, as the correct method for handling compiler errors is to remove them before you deploy the application.

Handling runtime exceptions

You have many choices for handling exceptions, and the exact path you take depends on your application and its needs. The following table provides a guide to selecting an appropriate technique:

Technique	Use
`cftry`	Place `cftry` blocks around specific code sections where exceptions can be expected and you want to handle those exceptions in a context-specific manner; for example, if you want to display an error message that is specific to that code. Use `cftry` blocks where you can recover from an exception. For example, you can retry an operation that times out, or access an alternate resource. You can also use the `cftry` tag to continue processing where a specific exception will not harm your application; for example, if a missing resource is not required. For more information, see "Handling runtime exceptions with ColdFusion tags" on page 322.
Application.cfc `onError` method	Implement the `onError` method in your Application.cfc to consistently handle application-specific exceptions that might be generated by multiple code sections in the application. For more information on error handling using Application.cfc, see "Handling errors in Application.cfc" on page 290.
`cferror with` exception-specific error handler pages	Use the `cferror` tag to specify error pages for specific exception types. These pages cannot recover from errors, but they can provide the user with information about the error's cause and steps that they can take to prevent the problem. For more information, see "Specifying custom error messages with the cferror tag" on page 317.
`cferror` with a Request error page	Use the `cferror` tag to specify a Request error handler that provides a customized, application-specific message for unrecoverable exceptions. Put the tag in the Application.cfc initialization code or on the Application.cfm page to make it apply to all pages in an application. A Request error page cannot use CFML tags, but it can display error variables. As a result, you can use it to display common error information, but you cannot provide error-specific instructions. Typically, Request pages display error variable values and application-specific information, including support contact information. For example code, see "Example of a request error page" on page 320.
Site-wide error handler page	Specify a site-wide error handler in the Administrator to provide consistent appearance and contents for all otherwise-unhandled exceptions in *all* applications on your server. Like the Request page, the site-wide error handler cannot perform error recovery. However, it can include CFML tags in addition to the error variables. Because a site-wide error handler prevents ColdFusion from displaying the default error message, it allows you to limit the information reported to users. It also lets you provide all users with default contact information or other instructions.

Specifying custom error messages with the cferror tag

Custom error pages let you control the error information that users see. You can specify custom error pages for different types of errors and handle different types of errors in different ways. For example, you can create specific pages to handle errors that could be recoverable, such as request time-outs. You can also make your error messages consistent with the look and feel of your application.

You can specify the following types of custom error message pages:

Type	Description
Validation	Handles server-side form field data validation errors. The validation error page cannot include CFML tags, but it can display error page variables. You can use this attribute only in the Application.cfc initialization code or on the Application.cfm page. It has no effect when used on any other page. Therefore, you can specify only one validation error page per application, and that page applies to all server-side validation errors.
Exception	Handles specific exception errors. You can specify individual error pages for different types of exceptions.
Request	Handles any exception that is not otherwise-handled. The request error page runs after the CFML language processor finishes. As a result, the request error page cannot include CFML tags, but can display error page variables. A request error page is useful as a backup if errors occur in other error handlers.

Specifying a custom error page

You specify the custom error pages with the `cferror` tag. For Validation errors, the tag must be in the Application.cfc initialization code or on the Application.cfm page. For Exception and Request errors, you can set the custom error pages on each application page. However, because custom error pages generally apply to an entire application, it is more efficient to put these `cferror` tags in the Application.cfc or Application.cfm file also. For more information on using these pages, see Chapter 13, "Designing and Optimizing a ColdFusion Application," on page 275.

The `cferror` tag has the attributes listed in the following table:

Attribute	Description
Type	The type of error that will cause ColdFusion to display this page: Exception, Request, or Validation.
Exception	Use only for the Exception type. The specific exception or exception category that will cause the page to be displayed. This attribute can specify any of the types described in "About ColdFusion exceptions" on page 310.
Template	The ColdFusion page to display.
MailTo	(Optional) An e-mail address. The `cferror` tag sets the error page `error.mailTo` variable to this value. The error page can use the `error.mailTo` value in a message that tells the user to send an error notification. ColdFusion does *not* send any message itself.

The following `cferror` tag specifies a custom error page for exceptions that occur in locking code and informs the error page of the of an e-mail address it can use to send a notification each time this type of error occurs:

```
<cferror type = "exception"
  exception = "lock"
  template = "../common/lockexcept.cfm"
  mailto = "server@mycompany.com">
```

For detailed information on the `cferror` tag, see *CFML Reference*.

Creating an error application page

The following table lists the rules and considerations that apply to error application pages:

Type	Considerations
Validation	• Cannot use CFML tags. • Can use HTML tags. • Can use the `Error.InvalidFields`, `Error.validationHeader`, and `Error.validationFooter` variables by enclosing them with number signs (#). • Cannot use any other CFML variables.
Request	• Cannot use CFML tags. • Can use HTML tags. • Can use nine CFML error variables, such as `Error.Diagnostics`, by enclosing them with number signs. • Cannot use other CFML variables.
Exception	• Can use full CFML syntax, including tags, functions, and variables. • Can use nine standard CFML Error variables and cfcatch variables. Use either `Error` or `cferror` as the prefix for both types of variables. • Can use other application-defined CFML variables. • To display any CFML variable, use the `cfoutput` tag.

The following table describes the variables available on error pages:

Error page type	Error variable	Description
Validation	error.invalidFields	Unordered list of validation errors that occurred. This includes any text that you specify in the value attribute or a hidden tag used to validate form input.
	error.validationHeader	Text for the header of the default validation message.
	error.validationFooter	Text for the footer of the default validation message.
Exception and request	error.browser	Browser that was running when the error occurred.
	error.dateTime	Date and time when the error occurred.
	error.diagnostics	Detailed error diagnostics.
	error.generatedContent	Any content that ColdFusion generated in response to the request prior to the error.
	error.HTTPReferer	Page containing the HTML link to the page on which the error occurred. This value is an empty string if the user specified the page directly in the browser.
	error.mailTo	E-mail address of the administrator who should be notified. This value is set in the `mailTo` attribute of the `cferror` tag.
	error.queryString	URL query string of the client's request, if any.
	error.remoteAddress	IP address of the remote client.
	error.template	Page being executed when the error occurred.
Exception only	error.message	Error message associated with the exception.
	error.rootCause	Java servlet exception reported by the JVM as the cause of the "root cause" of the exception. This variable is a Java object.
	error.tagContext	Array of structures containing information for each tag in the tag stack The tag stack consists of each tag that is currently open. For more information, see "Exception information in cfcatch blocks" on page 325
	error.type	Exception type. For more information, see "About ColdFusion exceptions" on page 310.

Exception error pages can also use all of the exception variables listed in the section "Exception information in cfcatch blocks" on page 325. To use these variables, replace the `cfcatch` prefix with `cferror` or `error`. For example, to use the exception message in an error page, refer to it as `error.message`.

In general, production Exception and Request pages should not display detailed error information, such as that supplied by the `error.diagnostics` variable. Typically, Exception pages e-mail detailed error information to an administrative address or log the information using the `cflog` tag instead of displaying it to the user. For more information on using the `cflog` tag, see "Logging errors with the cflog tag" on page 321.

Example of a request error page

The following example shows a custom error page for a request error:

```
<html>
<head>
<title>Products - Error</title>
</head>
<body>

<h2>Sorry</h2>

<p>An error occurred when you requested this page.</p>
<p>Please send e-mail with the following information to #error.mailTo# to
  report
  this error.</p>

<table border=1>
<tr><td><b>Error Information</b> <br>
  Date and time: #error.DateTime# <br>
  Page: #error.template# <br>
  Remote Address: #error.remoteAddress# <br>
  HTTP Referer: #error.HTTPReferer#<br>
</td></tr></table>

<p>We apologize for the inconvenience and will work to correct the problem.</p>
</body>
</html>
```

Example of a validation error page

The following example shows a simple custom error page for a validation error:

```
<html>
<head>
<title>Products - Error</title>
</head>
<body>

<h2>Data Entry Error</h2>

<p>You failed to correctly complete all the fields
in the form. The following problems occurred:</p>

#error.invalidFields#

</body>
</html>
```

Logging errors with the cflog tag

ColdFusion provides extensive capabilities for generating, managing, and viewing log files, as described in *Configuring and Administering ColdFusion MX*. It also provides the `cflog` tag which adds entries to ColdFusion logs.

ColdFusion automatically logs errors to the default logs if you use the default error handlers. In all other cases, you must use the `cflog` tag in your error handling code to generate log entries.

The `cflog` tag lets you specify the following information:

- A custom file or standard ColdFusion log file in which to write the message.
- Text to write to the log file. This can include the values of all available error and `cfcatch` variables.
- Message severity (type): Information Warning, Fatal, or Error.
- Whether to log any of the following: application name, thread ID, system date, or system time. By default, all get logged.

For example, you could use a `cflog` tag in an exception error-handling page to log the error information to an application-specific log file, as in the following page:

```
<html>
<head>
<title>Products - Error</title>
</head>
<body>

<h2>Sorry</h2>

<p>An error occurred when you requested this page.
The error has been logged and we will work to correct the problem.
We apologize for the inconvenience. </p>

<cflog type="Error"
     file="myapp_errors"
     text="Exception error --
       Exception type: #error.type#
       Template: #error.template#,
       Remote Address: #error.remoteAddress#,
       HTTP Reference: #error.HTTPReferer#
       Diagnostics: #error.diagnostics#">

</body>
</html>
```

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
`<cflog type="Error" file="myapp_errors" text="Exception error Exception type: #Error.type# Template: #Error.template#, Remote Address: #Error.remoteAddress#, HTTP Reference: #error.HTTPReferer# Diagnostics: #Error.diagnostics#">`	When this page is processed, log an entry in the file myapp_errors.log file in the ColdFusion log directory. Identify the entry as an error message and include an error message that includes the exception type, the path of the page that caused the error, the remote address that called the page, and the error's diagnostic message.

A log file entry similar to the following is generated if you try to call a nonexistent custom tag and this page catches the error (line breaks added for clarity):

```
"Error","web-13","12/19/01","11:29:07",MYAPP,"Exception error --
    Exception type: coldfusion.runtime.CfErrorWrapper
    Template: /MYStuff/MyDocs/exceptiontest.cfm,
    Remote Address: 127.0.0.1,
    HTTP Reference:
    Diagnostics: Cannot find CFML template for custom tag testCase. Cannot
    find CFML template for custom tag testCase. ColdFusion attempted looking
    in the tree of installed custom tags but did not find a custom tag with
    this name."
```

The text consists of a comma-delimited list of the following entries:

- Log entry type, specified by the `cflog type` attribute
- ID of the thread that was executing
- Date the entry was written to the log
- Time the entry was written to the log
- Application name, as specified in the Application.cfc initialization code (by setting the This.application variable) or by a `cfapplication` tag (for example, in an Application.cfm file).
- The message specified by the `cflog text` attribute.

Handling runtime exceptions with ColdFusion tags

Exceptions include any event that disrupts the normal flow of instructions in a ColdFusion page, such as failed database operations, missing include files, or developer-specified events. Ordinarily, when ColdFusion encounters an exception, it stops processing and displays an error message, or an error page specified by a `cferror` tag or the Site-wide Error Handler option on the Settings page in the Administrator. However, you can use the ColdFusion exception handling tags to catch and process runtime exceptions directly in ColdFusion pages.

This ability to handle exceptions directly in your application pages enables your application to do the following:

- Respond appropriately to specific errors within the context of the current application page
- Recover from errors whenever possible.

Exception-handling tags

ColdFusion provides the exception-handling tags listed in the following table:

Tag	Description
`cftry`	If any exceptions occur while processing the tag body, look for a `cfcatch` tag that handles the exception, and execute the code in the `cfcatch` tag body.
`cfcatch`	Execute code in the body of this tag if the exception caused by the code in the `cftry` tag body matches the exception type specified in this tag's attributes. Used in `cftry` tag bodies only.
`cfthrow`	Generate a user-specified exception.
`cfrethrow`	Exit the current `cfcatch` block and generates a new exception of the same type. Used only in `cfcatch` tag bodies.

Using cftry and cfcatch tags

The `cftry` tag lets you go beyond reporting error data to the user:

- You can include code that recovers from errors so your application can continue processing without alerting the user.
- You can create customized error messages that apply to the specific code that causes the error.

For example, you can use `cftry` to catch errors in code that enters data from a user registration form to a database. The `cfcatch` code could do the following:

1. Retry the query, so the operation succeeds if the resource was only temporarily unavailable.
2. If the retries fail:
   - Display a custom message to the user
   - Post the data to an email address so the data could be entered by company staff after the problem has been solved.

Code that accesses external resources such as databases, files, or LDAP servers where resource availability is not guaranteed is a good candidate for using try/catch blocks.

Try/catch code structure

In order for your code to directly handle an exception, the tags in question must appear within a `cftry` block. It is a good idea to enclose an entire application page in a `cftry` block. You then follow the `cftry` block with `cfcatch` blocks, which respond to potential errors. When an exception occurs within the `cftry` block, processing is thrown to the `cfcatch` block for that type of exception.

Here is an outline for using cftry and cfcatch to handle errors:

```
<cftry>
  Put your application code here ...
  <cfcatch type="exception type1">
    Add exception processing code here ...
  </cfcatch>
  <cfcatch type="exception type2">
    Add exception processing code here ...
  </cfcatch>
  ...
  <cfcatch type="Any">
    Add exception processing code appropriate for all other exceptions
  here ...
  </cfcatch>
</cftry>
```

Try/catch code rules and recommendations

Follow these rules and recommendations when you use cftry and cfcatch tags:

- The cfcatch tags must follow all other code in a cftry tag body.
- You can nest cftry blocks. For example, the following structure is valid:

```
<cftry>
  code that may cause an exception
  <cfcatch ...>
    <cftry>
      First level of exception handling code
      <cfcatch ...>
        Second level of exception handling code
      </cfcatch
    </cftry>
  </cfcatch>
</cftry>
```

  If an exception occurs in the first level of exception-handling code, the inner cfcatch block can catch and handle it. (An exception in a cfcatch block cannot be handled by cfcatch blocks at the same level as that block.)
- ColdFusion always responds to the latest exception that gets raised. For example, if code in a cftry block causes an exception that gets handled by a cfcatch block, and the cfcatch block causes an exception that has no handler, ColdFusion will display the default error message for the exception in the cfcatch block, and you will not be notified of the original exception.
- If an exception occurs when the current tag is nested inside other tags, the CFML processor checks the entire stack of open tags until it finds a suitable cftry/cfcatch combination or reaches the end of the stack.
- Use cftry with cfcatch to handle exceptions based on their point of origin within an application page, or based on diagnostic information.
- The entire cftry tag, including all its cfcatch tags, must be on a single ColdFusion page. You cannot put the <cftry> start tag on one page and have the </cftry> end tag on another page.

- For cases when a `cfcatch` block is not able to successfully handle an error, consider using the `cfrethrow` tag, as described in "Using the cfrethrow tag" on page 333.
- If an exception can be safely ignored, use a `cfcatch` tag with no body; for example:
  `<cfcatch Type = Database />`
- In particularly problematic cases, you might enclose an exception-prone tag in a specialized combination of `cftry` and `cfcatch` tags to immediately isolate the tag's exceptions.

Exception information in cfcatch blocks

Within the body of a `cfcatch` tag, the active exception's properties are available in a `cfcatch` object. The following sections describe the object contents.

Standard cfcatch variables

The following table describes the variables that are available in most `cfcatch` blocks:

Property variable	Description
`cfcatch.Detail`	A detailed message from the CFML compiler. This message, which can contain HTML formatting, can help to determine which tag threw the exception. The `cfcatch.Detail` value is available in the CFScript `cfcatch` statement as the `exceptionVariable` parameter.
`cfcatch.ErrorCode`	The `cfthrow` tag can supply a value for this code through the `errorCode` attribute. For `Type="Database"`, `cfcatch.ErrorCode` has the same value as `cfcatch.SQLState`. Otherwise, the value of `cfcatch.ErrorCode` is the empty string.
`cfcatch.ExtendedInfo`	Custom error message information. This is returned only to `cfcatch` tags for which the `type` attribute is `Application` or a custom type. Otherwise, the value of `cfcatch.ExtendedInfo` is the empty string.
`cfcatch.Message`	The exception's default diagnostic message, if one was provided. If no diagnostic message is available, this is an empty string. The `cfcatch.Message` value is included in the value of the CFScript `catch` statement `exceptionVariable` parameter.
`cfcatch.RootCause`	The Java servlet exception reported by the JVM as the cause of the "root cause" of the exception.
`cfcatch.TagContext`	An array of structures structure containing information for each tag in the tag stack The tag stack consists of each tag that is currently open.
`cfcatch.Type`	The exception's type, returned as a string.

Note: If you use `cfdump` to display the cfcatch variable, the display does not include variables that do not have values.

The `cfcatch.TagContext` variable contains an array of tag information structures. Each structure represents one level of the active tag context at the time when ColdFusion detected the exception. That is, there is one structure for each tag that is open at the time of the exception. For example, if the exception occurs in a tag on a custom tag page, the tag context displays information about the called custom tag and the tag in which the error occurs.

The structure at position 1 in the array represents the currently executing tag at the time the exception was detected. The structure at position `ArrayLen(cfcatch.tagContext)` represents the initial tag in the stack of tags that were executing when the compiler detected the exception.

The following table lists the tagContext structure attributes:

Entry	Description
Column	Obsolete (retained for backwards compatibility). Always 0.
ID	The tag in which the exception occurred. Exceptions in CFScript are indicated by two question marks (??). All custom tags, including those called directly, are identified as cfmodule.
Line	The line on the page in which the tag is located.
Raw_Trace	The raw Java stack trace for the error.
Template	The pathname of the application page that contains the tag.
Type	The type of page; it is always a ColdFusion page.

Database exceptions

The following additional variables are available whenever the exception type is database:

Property variable	Description
`cfcatch.NativeErrorCode`	The native error code associated with this exception. Database drivers typically provide error codes to assist in the diagnosis of failing database operations. The values assumed by `cfcatch.NativeErrorCode` are driver-dependent. If no error code is provided, the value of `cfcatch.nativeErrorCode` is -1. The value is 0 for queries of queries.
`cfcatch.SQLState`	The SQLState code associated with this exception. Database drivers typically provide error codes to assist in the diagnosis of failing database operations. SQLState codes are more consistent across database systems than native error codes. If the driver does not provide an SQLState value, the value of `cfcatch.SQLState` is -1.
`cfcatch.Sql`	The SQL statement sent to the data source.
`cfcatch.queryError`	The error message as reported by the database driver.
`cfcatch.where`	If the query uses the `cfqueryparam` tag, query parameter name-value pairs.

Expression exceptions

The following variable is only available for Expression exceptions:

Property variable	Description
`cfcatch.ErrNumber`	An internal expression error number, valid only when `type="Expression"`.

Locking exceptions

The following additional information is available for exceptions related to errors that occur in `cflock` tags:

Property variable	Description
`cfcatch.lockName`	The name of the affected lock. This is set to "anonymous" if the lock name is unknown.
`cfcatch.lockOperation`	The operation that failed. This is set to "unknown" if the failed operation is unknown.

Missing include exceptions

The following additional variable is available if the error is caused by a missing file specified by a `cfinclude` tag:

Property variable	Description
`cfcatch.missingFileName`	The name of the missing file.

Using the cftry tag: an example

The following example shows the `cftry` and `cfcatch` tags. It uses the cfdocexamples data source, which many of the examples in this manual use, and a sample included file, `includeme.cfm`.

If an exception occurs during the `cfquery` statement's execution, the application page flow switches to the `cfcatch type="Database"` exception handler. It then resumes with the next statement after the `cftry` block, once the `cfcatch type="Database"` handler completes. Similarly, the `cfcatch type="MissingInclude"` block handles exceptions raised by the `cfinclude` tag.

```
<!--- Wrap code you want to check in a cftry block --->
<cfset EmpID=3>
<cfparam name="errorCaught" default="">
<cftry>
  <cfquery name="test" datasource="cfdocexamples">
    SELECT Dept_ID, FirstName, LastName
    FROM Employee
    WHERE Emp_ID=#EmpID#
  </cfquery>

  <html>
  <head>
  <title>Test cftry/cfcatch</title>
  </head>
  <body>
  <cfinclude template="includeme.cfm">
  <cfoutput query="test">
    <p>Department: #Dept_ID#<br>
    Last Name: #LastName#<br>
    First Name: #FirstName#</p>
  </cfoutput>
```

```
<!--- Use cfcatch to test for missing included files. --->
<!--- Print Message and Detail error messages. --->
<!--- Block executes only if a MissingInclude exception is thrown. --->
  <cfcatch type="MissingInclude">
    <h1>Missing Include File</h1>
    <cfoutput>
    <ul>
      <li><b>Message:</b> #cfcatch.Message#
      <li><b>Detail:</b> #cfcatch.Detail#
      <li><b>File name:</b> #cfcatch.MissingFileName#
    </ul>
    </cfoutput>
    <cfset errorCaught = "MissingInclude">
  </cfcatch>

<!--- Use cfcatch to test for database errors.--->
<!--- Print error messages. --->
<!--- Block executes only if a Database exception is thrown. --->
  <cfcatch type="Database">
    <h1>Database Error</h1>
    <cfoutput>
    <ul>
      <li><b>Message:</b> #cfcatch.Message#
      <li><b>Native error code:</b> #cfcatch.NativeErrorCode#
      <li><b>SQLState:</b> #cfcatch.SQLState#
      <li><b>Detail:</b> #cfcatch.Detail#
      </ul>
    </cfoutput>
    <cfset errorCaught = "Database">
  </cfcatch>

<!--- Use cfcatch with type="Any" --->
<!--- to find unexpected exceptions. --->
  <cfcatch type="Any">
    <cfoutput>
      <hr>
      <h1>Other Error: #cfcatch.Type#</h1>
      <ul>
        <li><b>Message:</b> #cfcatch.Message#
        <li><b>Detail:</b> #cfcatch.Detail#
      </ul>
    </cfoutput>
    <cfset errorCaught = "General Exception">
  </cfcatch>
</body>
</html>
</cftry>
```

Use the following procedure to test the code.

To test the code:

1. Make sure there is no includeme.cfm file and display the page. The `cfcatch type="MissingInclude"` block displays the error.
2. Create a nonempty includeme.cfm file and display the page. If your database is configured properly, you should see an employee entry and not get any error.
3. In the `cfquery` tag, change the line:

   ```
   FROM Employee
   ```

   to:

   ```
   FROM Employer
   ```

   Display the page. This time the `cfcatch type="Database"` block displays an error message.
4. Change Employer to Employee.

   Change the `cfoutput` line:

   ```
   <p>Department: #Dept_ID#<br>
   ```

   to:

   ```
   <p>Department: #DepartmentID#<br>
   ```

   Display the page. This time the `cfcatch type="Any"` block displays an error message indicating an expression error.
5. Change DepartmentID back to Dept_ID and redisplay the page. The page displays properly.

   Open \CFusion\Log\MyAppPage.log in your text editor. You should see a header line, an initialization line, and four detail lines, similar to the following:

   ```
   "Severity","ThreadID","Date","Time","Application","Message"
   "Information","web-0","11/20/01", "16:27:08",,
     "cf_root\runtime\servers\default\logs\ MyAppPage.log initialized"
   "Information","web-0","11/20/01","16:27:08",,
     "Page: web_root/MYStuff/MyDocs/ cftryexample.cfm Error: MissingInclude"
   "Information","web-1","11/20/01","16:27:32",,"
     Page: web_root/MYStuff/MyDocs/ cftryexample.cfm Error: "
   "Information","web-0","11/20/01","16:27:49",,
     "Page: web_root/MYStuff/MyDocs/ cftryexample.cfm Error: Database"
   "Information","web-1","11/20/01","16:28:21",,
     "Page: web_root/MYStuff/MyDocs/ cftryexample.cfm Error: General Exception"
   "Information","web-0","11/20/01","16:28:49",,
     "Page: web_root/MYStuff/MyDocs/ cftryexample.cfm Error: "
   ```

Reviewing the code

The following table describes the code:

Code	Description
`<cfset EmpID=3>` `<cfparam name="errorCaught" default="">`	Initializes the employee ID to a valid value. An application would get the value from a form or other source. Sets the default `errorCaught` variable value to the empty string (to indicate no error was caught). There is no need to put these lines in a `cftry` block.

`<cftry>`
`<cfquery name="test"`
`datasource="cfdocexamples">`
`SELECT Dept_ID, FirstName, LastName`
`FROM Employee`
`WHERE Emp_ID=#EmpID#`
`</cfquery>`
	Starts the `cftry` block. Exceptions from here to the end of the block can be caught by `cfcatch` tags.
	Queries the cfdocexamples database to get the data for the employee identified by the `EmpID` variable.

`<html>` `<head>` `<title>Test cftry/cfcatch</title>` `</head>` `<body>` `<cfinclude template="includeme.cfm">` `<cfoutput query="test">` `<p>Department: #Dept_ID# ` `Last Name: #LastName# ` `First Name: #FirstName#</p>` `</cfoutput>`	Begins the HTML page. This section contains all the code that displays information if no errors occur. Includes the includeme.cfm page. Displays the user information record from the test query.

`<cfcatch type="MissingInclude">`
`<h1>Missing Include File</h1>`
`<cfoutput>`
`<ul>`
`<li><b>Message:</b> #cfcatch.Message#`
`<li><b>Detail:</b> #cfcatch.Detail#`
`<li><b>File name:</b>`
`#cfcatch.MissingFilename#`
`</ul>`
`</cfoutput>`
`<cfset errorCaught = "MissingInclude">`
`</cfcatch>`
	Handles exceptions thrown when a page specified by the `cfinclude` tag cannot be found.
	Displays `cfcatch` variables, including the ColdFusion basic error message, detail message, and the name of the file that could not be found.
	Sets the `errorCaught` variable to indicate the error type.

`<cfcatch type="Database">`
`<h1>Database Error</h1>`
`<cfoutput>`
`<ul>`
`<li><b>Message:</b> #cfcatch.Message#`
`<li><b>Native error code:</b>`
`#cfcatch.NativeErrorCode#`
`<li><b>SQLState:</b> #cfcatch.SQLState#`
`<li><b>Detail:</b> #cfcatch.Detail#`
`</ul>`
`</cfoutput>`
`<cfset errorCaught = "Database">`
`</cfcatch>`
	Handles exceptions thrown when accessing a database.
	Displays `cfcatch` variables, including the ColdFusion basic error message, the error code and SQL state reported by the databases system, and the detailed error message.
	Sets the `errorCaught` variable to indicate the error type.

Code	Description

`<cfcatch type="Any">`
`<cfoutput>`
`<hr>`
`<h1>Other Error: #cfcatch.Type#</h1>`
`<ul>`
`<li><b>Message:</b> #cfcatch.message#`
`<li><b>Detail:</b> #cfcatch.Detail#`
`</ul>`
`</cfoutput>`
`<cfset errorCaught = "General Exception">`
`</cfcatch>`
	Handles any other exceptions generated in the `cftry` block.
	Since the error can occur after information has displayed (in this case, the contents of the include file), draws a line before writing the message text.
	Displays the ColdFusion basic and detailed error message.
	Sets the `errorCaught` variable to indicate the error type.

`</body>`
`</html>`
`</cftry>`
	Ends the HTML page, then the `cftry` block.

Using the cfthrow tag

You can use the `cfthrow` tag to raise your own, custom exceptions. When you use the `cfthrow` tag, you specify any or all of the following information:

Attribute	Meaning
type	The type of error. It can be a custom type that has meaning only to your application, such as InvalidProductCode. You can also specify Application, the default type. You cannot use any of the predefined ColdFusion error types, such as Database or MissingTemplate.
message	A brief text message indicating the error.
detail	A more detailed text message describing the error.
errorCode	An error code that is meaningful to the application. This field is useful if the application uses numeric error codes.
extendedInfo	Any additional information of use to the application.

All of these values are optional. You access the attribute values in `cfcatch` blocks and Exception type error pages by prefixing the attribute with either `cfcatch` or `error`, as in `cfcatch.extendedInfo`. The default ColdFusion error handler displays the message and detail values in the Message pane and the remaining values in the Error Diagnostic Information pane.

Catching and displaying thrown errors

The `cfcatch` tag catches a custom exception when you use any of the following values for the `cfcatch type` attribute:

- The custom exception type specified in the `cfthrow` tag.
- A custom exception type that hierarchically matches the initial portion of the type specified in the `cfthrow` tag. For more information, see the next section, "Custom error type name hierarchy".
- `Application`, which matches an exception that is thrown with the `Application type` attribute or with no `type` attribute.

- `Any`, which matches any exception that is not caught by a more specific `cfcatch` tag.

Similarly, if you specify any of these types in a `cferror` tag, the specified error page will display information about the thrown error.

Because the `cfthrow` tag generates an exception, a Request error handler or the Site-wide error handler can also display these errors.

Custom error type name hierarchy

You can name custom exception types using a method that is similar to Java class naming conventions: domain name in reverse order, followed by project identifiers, as in the following example:

```
<cfthrow
  type="com.myCompany.myApp.Invalid_field.codeValue"
  errorcode="Dodge14B">
```

This fully qualified naming method is not required; you can use shorter naming rules, for example, just myApp.Invalid_field.codeValue, or even codeValue.

This naming method is *not* just a convention, however; ColdFusion MX uses the naming hierarchy to select from a possible hierarchy of error handlers. For example, assume you use the following `cfthrow` statement:

```
<cfthrow type="MyApp.BusinessRuleException.InvalidAccount">
```

Any of the following `cfcatch` error handlers would handle this error:

```
<cfcatch type="MyApp.BusinessRuleException.InvalidAccount">
<cfcatch type="MyApp.BusinessRuleException">
<cfcatch type="MyApp">
```

The handler that most exactly matches handles the error. Therefore, in this case, the `MyApp.BusinessRuleException.InvalidAccount` handler gets invoked. However, if you used the following `cfthrow` tag:

```
<cfthrow type="MyApp.BusinessRuleException.InvalidVendorCode
```

the `MyApp.BusinessRuleException` handler receives the error.

The type comparison is no case-sensitive.

When to use the cfthrow tag

Use the `cfthrow` tag when your application can identify and handle application-specific errors. One typical use for the `cfthrow` tag is in implementing custom data validation. The `cfthrow` tag is also useful for throwing errors from a custom tag page to the calling page.

For example, on a form action page or custom tag used to set a password, the application can determine whether the password entered is a minimum length, or contains both letters and number, and throw an error with a message that indicates the password rule that was broken. The `cfcatch` block handles the error and tells the user how to correct the problem.

Using the cfrethrow tag

The cfrethrow tag lets you create a hierarchy of error handlers. It tells ColdFusion to exit the current cfcatch block and "rethrow" the exception to the next level of error handler. Thus, if an error handler designed for a specific type of error cannot handle the error, it can rethrow the error to a more general-purpose error handler. The cfrethrow tag can only be used in a cfcatch tag body.

The cfrethrow tag syntax

The following pseudocode shows how you can use the cfrethrow tag to create an error-handling hierarchy:

```
<cftry>
  <cftry>
    Code that might throw a database error
    <cfcatch Type="Database">
      <cfif Error is of type I can Handle>
        Handle it
      <cfelse>
        <cfrethrow>
      </cfif
    </cfcatch>
  </cftry>
  <cfcatch Type="Any">
    General Error Handling code
  </cfcatch>
</cftry>
```

Although this example uses a Database error as an example, you can use any cfcatch type attribute in the innermost error type.

Follow these rules when you use the cfrethrow tag:

- Nest cftry tags, with one tag for each level of error handling hierarchy. Each level contains the cfcatch tags for that level of error granularity.
- Place the most general error catching code in the outermost cftry block.
- Place the most specific error catching code in the innermost cftry block.
- Place the code that can cause an exception error at the top of the innermost cftry block.
- End each cfcatch block except those in the outermost cftry block with a cfrethrow tag.

Example: using nested tags, cfthrow, and cfrethrow

The following example shows many of the techniques discussed in this chapter, including nested cftry blocks and the cfthrow and cfrethrow tags. The example includes a simple calling page and a custom tag page:

- The calling page does little more than call the custom tag with a single attribute, a name to be looked up in a database. It does show, however, how a calling page can handle an exception thrown by the custom tag.

- The custom tag finds all records in the cfdocexamples database with a matching last name, and returns the results in a Caller variable. If it fails to connect with the main database, it tries a backup database.

The calling page

The calling page represents a section from a larger application page. To keep things simple, the example hard-codes the name to be looked up.

```
<cftry>
  <cf_getEmps EmpName="Jones">
  <cfcatch type="myApp.getUser.noEmpName">
    <h2>Oops</h2>
    <cfoutput>#cfcatch.Message#</cfoutput><br>
  </cfcatch>
</cftry>
<cfif isdefined("getEmpsResult")>
  <cfdump var="#getEmpsResult#">
</cfif>
```

Reviewing the code

The following table describes the code:

Code	Description
`<cftry>` `<cf_getEmps EmpName="Jones">`	In a `cftry` block, calls the cf_getEmps custom tag (getEmps.cfm).
`<cfcatch type="myApp.getUser.noEmpName">` `<h2>Oops</h2>` `<cfoutput>#cfcatch.Message#</cfoutput> ` `</cfcatch>` `</cftry>`	If the tag throws an exception indicating that it did not receive a valid attribute, catches the exception and displays a message, including the message variable set by the `cfthrow` tag in the custom tag.

`<cfif isdefined("getEmpsResult")>`
`<cfdump var="#getEmpsResult#">`
`</cfif>`
	If the tag returns a result, uses the `cfdump` tag to display it. (A production application would not use the `cfdump` tag.)

The custom tag page

The custom tag page searches for the name in the database and returns any matching records in a getEmpsResult variable in the calling page. It includes several nested `cftry` blocks to handle error conditions. For a full description, see "Reviewing the code" on page 336, following the example:

Save the following code as getEmps.cfm in the same directory as the calling page.

```
<!--- If the tag didn't pass an attribute, throw an error to be handled by
       the calling page --->
<cfif NOT IsDefined("attributes.EmpName")>
  <cfthrow Type="myApp.getUser.noEmpName"
    message = "Last Name was not supplied to the cf_getEmps tag.">
  <cfexit method = "exittag">
<!--- Have a name to look up --->
<cfelse>
<!--- Outermost Try Block --->
```

```
  <cftry>

<!--- Inner Try Block --->
    <cftry>
<!--- Try to query the main database and set a caller variable to the result -
  -->
      <cfquery Name = "getUser" DataSource="cfdocexamples">
        SELECT *
        FROM Employee
        WHERE LastName = '#attributes.EmpName#'
      </cfquery>
      <cfset caller.getEmpsResult = getuser>
<!--- If the query failed with a database error, check the error type
      to see if the database was found --->
      <cfcatch type= "Database">
        <cfif (cfcatch.SQLState IS "S100") OR (cfcatch.SQLState IS
          "IM002")>

<!--- If the database wasn't found, try the backup database --->
<!--- Use a third-level Try block --->
          <cftry>
            <cfquery Name = "getUser" DataSource="cfdocexamplesBackup">
              SELECT *
              FROM Employee
              WHERE LastName = '#attributes.EmpName#'
            </cfquery>
            <cfset caller.getEmpsResult = getuser>

<!--- If still get a database error, just return to the calling page
      without setting the caller variable. There is no cfcatch body.
      This might not be appropriate in some cases.
      The Calling page ends up handling this case as if a match was not
      found --->
            <cfcatch type = "Database" />
<!--- Still in innermost try block. Rethrow any other errors to the next
      try block level --->
            <cfcatch type = "Any">
              <cfrethrow>
            </cfcatch>
          </cftry>

<!--- Now in second level try block.
      Throw all other types of Database exceptions to the next try
        block level --->
        <cfelse>
          <cfrethrow>
        </cfif>
      </cfcatch>
<!--- Throw all other exceptions to the next try block level --->
      <cfcatch type = "Any">
          <cfrethrow>
      </cfcatch>
    </cftry>

<!--- Now in Outermost try block.
```

```
      Handle all unhandled exceptions, including rethrown exceptions, by
        displaying a message and exiting to the calling page.--->
      <cfcatch Type = "Any">
        <h2>Sorry</h2>
        <p>An unexpected error happened in processing your user inquiry.
          Please report the following to technical support:</p>
        <cfoutput>
          Type: #cfcatch.Type#
          Message: #cfcatch.Message#
        </cfoutput>
        <cfexit method = "exittag">
      </cfcatch>
    </cftry>
  </cfif>
```

Reviewing the code

The following table describes the code:

Code	Description

`<cfif NOT IsDefined("attributes.EmpName")>`
`  cfthrow Type="myApp.getUser.noEmpName"`
`  message = "Last Name was not supplied to the cf_getEmps tag.">`
`  <cfexit method = "exittag">`
	Makes sure the calling page specified an `EmpName` attribute. If not, throws a custom error that indicates the problem and exits the tag. The calling page handles the thrown error.

`<cfelse>` `  <cftry>`	If the tag has an `EmpName` attribute, does the remaining work inside an outermost try block. The `cfcatch` block at its end handles any otherwise-uncaught exceptions.

`<cftry>`
`  <cfquery Name = "getUser"`
`  DataSource="cfdocexamples">`
`  SELECT *`
`  FROM Employee`
`  WHERE LastName = '#attributes.EmpName#'`
`  </cfquery>`
`  <cfset caller.getEmpsResult = getuser>`
	Starts a second nested try block. This block catches exceptions in the database query.
	If there are no exceptions, sets the calling page's `getEmpsResult` variable with the query results.

`<cfcatch type= "Database">`
`  <cfif (cfcatch.sqlstate IS "S100") OR`
`  (cfcatch.sqlstate IS "IM002")>`
`  <cftry>`
`  <cfquery Name = "getUser" DataSource=`
`  "cfdocexamplesBackup"`
`  SELECT *`
`  FROM Employee`
`  WHERE LastName =`
`  '#attributes.EmpName#'`
`  </cfquery>`
`  <cfset caller.getEmpsResult = getuser>`
	If the query threw a Database error, checks to see if the error was caused by an inability to access the database (indicated by an `SQLState` variable value of S100 or IM002).
	If the database was not found, starts a third nested try block and tries accessing the backup database. This try block catches exceptions in this second database access.
	If the database inquiry succeeds, sets the calling page's `getEmpsResult` variable with the query results.

Code	Description
`<cfcatch type = "Database" />`	If the second database query failed with a database error, gives up silently. Because the Database type `cfcatch` tag does not have a body, the tag exits. The calling page does not get a `getEmpsResult` variable. It cannot tell whether the database had no match or an unrecoverable database error occurred, but it does know that no match was found.

`<cfcatch type = "Any">`
`<cfrethrow>`
`</cfcatch>`
`</cftry>`
	If the second database query failed for any other reason, throws the error up to the next try block.
	Ends the innermost try block

`<cfelse>`
`<cfrethrow>`
`</cfif>`
`</cfcatch>`
	In the second try block, handles the case in which the first database query failed for a reason other than a failure to find the database.
	Rethrows the error up to the next level, the outermost try block.

`<cfcatch type = "Any">`
`<cfrethrow>`
`</cfcatch>`
`</cftry>`
	In the second try block, catches any errors other exceptions and rethrows them up to the outermost try block.
	Ends the second try block.

`<cfcatch Type = "Any">`
`<h2>Sorry</h2>`
`<p>An unexpected error happened in processing`
`your user inquiry.`
`Please report the following to technical support:</p>`
`<cfoutput>`
`Type: #cfcatch.Type#`
`Message: #cfcatch.Message#`
`</cfoutput>`
`<cfexit method = "exittag">`
`</cfcatch>`
`</cftry>`
`</cfif>`
	In the outermost try block, handles any exceptions by displaying an error message that includes the exception type and the exception's error message. Because there was no code to try that is not also in a nested try block, this `cfcatch` tag handles only errors that are rethrown from the nested blocks.
	Exits the custom tag and returns to the calling page.
	Ends the catch block, try block, and initial `cfif` block.

Testing the code

To test the various ways errors can occur and be handled in this example, try the following:

- In the calling page, change the attribute name to any other value; for example, My Attrib. Then change it back.
- In the first `cfquery` tag, change the data source name to an invalid data source; for example, NoDatabase.
- With an invalid first data source name, change the data source in the second `cfquery` tag to cfdocexamples.
- Insert `cfthrow` tags throwing custom exculpations in various places in the code and observe the effects.

CHAPTER 15
Using Persistent Data and Locking

Macromedia ColdFusion MX provides several variable scopes in which data persists past the life of a single request. These are the Client, Application, Session, and Server scopes. These scopes let you save data over time and share data between pages and even applications. (This chapter refers to these scopes as *persistent* scopes.) In particular, you can use the Client and Session scopes to maintain information about a user across multiple requests.

ColdFusion MX lets you lock access to sections of code to ensure that ColdFusion does not attempt to run the code, or access the data that it uses, simultaneously or in an unpredictable order. This locking feature is important for ensuring the consistency of all shared data, including data in external sources in addition to data in persistent scopes.

This chapter describes how to use persistent scopes to develop an application and how to use locking to ensure data consistency.

Contents

About persistent scope variables 340
Managing the client state 342
Configuring and using client variables 346
Configuring and using session variables 352
Configuring and using application variables 357
Using server variables 359
Locking code with cflock 360
Examples of cflock 368

About persistent scope variables

ColdFusion MX provides four variable scopes, described in the following table, that let you maintain data that must be available to multiple applications or users or must last beyond the scope of the current request.

Variable scope	Description
Client	Contains variables that are available for a single client browser over multiple browser sessions in an application. For information about browser sessions, see, "What is a session?" on page 352. Useful for client-specific information, such as client preferences, that you want to store for a significant period of time. Data is stored as cookies, database entries, or Registry values. Client variables can time out after an extended period. Although do not have to use the Client scope prefix in the variable name, code that uses the prefix is more efficient and easier to maintain.
Session	Contains variables that are available for a single client browser for a single browser session in an application. Useful for client-specific information, such as shopping cart contents, that you want to persist while the client is visiting your application. Data is stored in memory and times out after a period of inactivity or when the server shuts down. ColdFusion MX Administrator lets you select between two kinds of session management, Standard ColdFusion Session management and J2EE session management. For information about types of session management, see "ColdFusion and J2EE session management" on page 352. You must use the Session scope prefix in the variable name.
Application	Contains variables that are available to all pages in an application for all clients. Useful for application-specific information, such as contact information, that can vary over time and should be stored in a variable. Data is stored in memory and times out after a period of inactivity or when the server shuts down. You must use the Application scope prefix in the variable name.
Server	Contains variables that are available to all applications in a server and all clients. Useful for information that applies to all pages on the server, such as an aggregate page-hit counter. Data is stored in memory. The variables do not time out, but you can delete variables you create, and all server variables are automatically deleted when the server stops running. You must use the Server scope prefix in the variable name.

The following sections provide information that is common to all or several of these variables. Later sections describe how to use the Client, Session, Application, and Server scopes in your applications, and provide detailed information about locking code.

ColdFusion persistent variables and ColdFusion structures

All persistent scopes are available as ColdFusion structures. As a result, you can use ColdFusion structure functions to access and manipulate Client, Session, Application, and Server scope contents. This chapter does not cover using these functions in detail, but does mention features or limitations that apply to specific scopes.

Note: Although you can use the `StructClear` function to clear your data from the Server scope, the function does not delete the names of the variables, only their values, and it does not delete the contents of the Server.os and Server.ColdFusion structures. Using the `StructClear` function to clear the Session, or Application scope clears the entire scope, including the built-in variables. Using the `StructClear` function to clear the Client scope clears the variables from the server memory, but does not delete the stored copies of the variables.

ColdFusion persistent variable issues

Variables in the Session, Application, and Server scopes are kept in ColdFusion server memory. This storage method has several implications:

- All variables in these scopes are lost if the server stops running.
- Variables in these scopes are not shared by servers in a cluster.
- To prevent race conditions and ensure data consistency, lock access to all code that changes variables in these scopes or reads variables in these scopes with values that can change.

Note: If you use J2EE session management and configure the J2EE server to retain session data between server restarts, ColdFusion retains session variables between server restarts.

Additionally, you must be careful when using client variables in a server cluster, where an application can run on multiple servers.

Locking memory variables

Because ColdFusion is a multithreaded system in which multiple requests can share Session, Application, and Server scope variables, it is possible for two or more requests to try to access and modify data at the same time. ColdFusion runs in a J2EE environment, which prevents simultaneous data access, so multiple requests do not cause severe system errors. However, such requests can result in inconsistent data values, particularly when a page might change more than one variable.

To prevent data errors with session, application, and server variables, lock code that writes and reads data in these scopes. For more information, see "Locking code with cflock" on page 360.

Using variables in clustered systems

Because memory variables are stored in memory, they are not available to all servers in a cluster. As a result, you generally do not use Session, Application, or Server scope variables in clustered environment. However, you might use these scope variables in a clustered system in the following circumstances:

- If the clustering system supports "sticky" sessions, in which the clustering system ensures that each user session remains on a single server. In this case, you can use session variables as you would on a single server.
- You can use Application and Server scope variables in a cluster for write-once variables that are consistently set, for example, from a database.

To use client variables on a clustered system, store the variables as cookies or in a database that is available to all servers. If you use database storage, on one server only, select the Purge Data for Clients that Remain Unvisited option on the Client Variables, Add/Edit Client Store page in the Server Settings area in the ColdFusion MX Administrator.

For more information on using client and session variables in clustered systems, see "Managing client identity information in a clustered environment" on page 346.

Managing the client state

Because the web is a stateless system, each connection that a browser makes to a web server is unique to the web server. However, many applications must keep track of users as they move through the pages within the application. This is the definition of *client state management*.

ColdFusion provides tools to maintain the client state by seamlessly tracking variables associated with a browser as the user moves from page to page within the application. You can use these variables in place of other methods for tracking client state, such as URL parameters, hidden form fields, and HTTP cookies.

About client and session variables

ColdFusion provides two tools for managing the client state: client variables and session variables. Both types of variables are associated with a specific client, but you manage and use them differently, as described in the following table:

Variable type	Description
Client	Data is saved as cookies, database entries, or Registry entries. Data is saved between server restarts, but is initially accessed and saved more slowly than data stored in memory. Each type of data storage has its own time-out period. You can specify the database and Registry data time-outs in ColdFusion MX Administrator. ColdFusion sets Cookie client variables to expire after approximately 10 years. Data is stored on a per-user and per-application basis. For example, if you store client variables as cookies, the user has a separate cookie for each ColdFusion application provided by a server. Client variables must be simple variables, such as numbers, dates, or strings. They cannot be arrays, structures, query objects, or other objects. Client variable names can include periods. For example, My.ClientVar is a valid name for a simple client variable. Avoid such names, however, to ensure code clarity. You do not have to prefix client variables with the scope name when you reference them, However, if you do not use the Client prefix, you might unintentionally refer to a variable with the same name in another scope. Using the prefix also optimizes performance and increases program clarity. You do not lock code that uses client variables. You can use client variables that are stored in cookies or a common database in clustered systems.
Session	Data is stored in memory so it is accessed quickly. Data is lost when the client browser is inactive for a time-out period. You specify the time-out in the ColdFusion MX Administrator, the Application.cfc initialization code, or Application.cfm. As with client variables, data is available to a single client and application only. Variables can store any ColdFusion data type. You must prefix all variable names with the Session scope name. Lock code that uses session variables to prevent race conditions. You can use session variables in clustered systems only if the systems support "sticky" sessions, where a session is limited to a single server.

Session variables are normally better than client variables for values that need to exist for only a single browser session. You should reserve client variables for client-specific data, such as client preferences that you want available for multiple browser sessions.

Maintaining client identity

Because the web is a stateless system, client management requires some method for maintaining knowledge of the client between requests. Normally you do this using cookies, but you can also do it by passing information between application pages. The following sections describe how ColdFusion maintains client identity in a variety of configurations and environments, and discuss issues that can arise with client state management.

About client identifiers

To use client and session variables, ColdFusion must be able to identify the client. It normally does so by setting the following two cookie values on the client's system:

- **CFID** A sequential client identifier
- **CFToken** A random-number client security token

These cookies uniquely identify the client to ColdFusion, which also maintains copies of the variables as part of the Session and Client scopes. You can configure your application so that it does not use client cookies, but in this case, you must pass these variables to all the pages that your application calls. For more information about maintaining client and session information without using cookies, see "Using client and session variables without cookies" on page 344.

You can configure ColdFusion MX to use J2EE servlet session management instead of ColdFusion session management for session variables. This method of session management does not use `CFID` and `CFToken` values, but does use a client-side `jsessionid` session management cookie. For more information on using J2EE session management, see "ColdFusion and J2EE session management" on page 352.

Using client and session variables without cookies

Often, users disable cookies in their browsers. In this case, ColdFusion cannot maintain the client state automatically. You can use client or session variables without using cookies, by passing the client identification information between application pages. However, this technique has significant limitations, as follows:

- Client variables are effectively the same as session variables, except that they leave unusable data in the client data store.

  Because the client's system does not retain any identification information, the next time the user logs on, ColdFusion cannot identify the user with the previous client and must create a new client ID for the user. Any information about the user from a previous session is not available, but remains in client data storage until ColdFusion deletes it. If you clear the Purge Data for Clients that Remain Unvisited option in the ColdFusion MX Administrator, ColdFusion never deletes this data.

  Therefore, do not use client variables, if you allow users to disable cookies. To retain client information without cookies, require users to login with a unique ID. You can then save user-specific information in a database with the user's ID as a key.
- ColdFusion creates a new session each time the user requests a page directly in the browser, because the new request contains no state information to indicate the session or client.

Note: You can prevent ColdFusion from sending client information to the browser as cookies by setting This.setClientCookies variable in Application.cfc or the `setClientCookies` attribute of the `cfapplication` tag to No.

To use ColdFusion session variables without using cookies, each page must pass the CFID and CFToken values to any page that it calls as part of the request URL. If a page contains any HTML href a= links, cflocation tags, form tags, or cfform tags the tags must pass the CFID and CFToken values in the tag URL. To use J2EE session management, you must pass the jsessionid value in page requests. To use ColdFusion client variables and J2EE session variables, you must pass the CFID, CFToken, and jsessionid values in URLs.

ColdFusion provides the URLSessionFormat function, which does the following:

- If the client does not accept cookies, automatically appends all required client identification information to a URL.
- If the client accepts cookies, does not append the information.

The URLSessionFormat function automatically determines which identifiers are required, and sends only the required information. It also provides a more secure and robust method for supporting client identification than manually encoding the information in each URL, because it only sends the information that is required, when it is required, and it is easier to code.

To use the URLSessionFormat function, enclose the request URL in the function. For example, the following cfform tag posts a request to another page and sends the client identification, if required:

```
<cfform method="Post" action="#URLSessionFormat("MyActionPage.cfm")#>
```

Tip: If you use the same page URL in multiple URLSessionFormat functions, you can gain a small performance improvement and simplify your code if you assign the formatted page URL to a variable, for example:

```
<cfset myEncodedURL=URLSessionFormat(MyActionPage.cfm)>
<cfform method="Post" action="#myEncodedURL#">
```

Client identifiers and security

The following client identifier issues can have security implications:

- Ensuring the uniqueness and complexity of the CFToken identifier
- Limiting the availability of Session identifiers

The next sections discuss these issues.

Ensuring CFToken uniqueness and security

By default, ColdFusion uses an eight-digit random number in the CFToken identifier. This CFToken format provides a unique, secure identifier for users under most circumstances. (In ColdFusion MX, the method for generating this number uses a cryptographic-strength random number generator that is seeded only when the server starts.)

However, in the ColdFusion MX Administrator, you can enable the Settings page to produce a more complex CFToken identifier. If you enable the Use UUID for cftoken option, ColdFusion creates the CFToken value by prepending a 16-digit random hexadecimal number to a ColdFusion UUID. The resulting CFToken identifier looks similar to the following:

```
3ee6c307a7278c7b-5278BEA6-1030-C351-3E33390F2EAD02B9
```

Providing Session security

ColdFusion uses the same client identifiers for the Client scope and the standard Session scope. Because the `CFToken` and `CFID` values are used to identify a client over a period of time, they are normally saved as cookies on the user's browser. These cookies persist until the client's browser deletes them, which can be a considerable length of time. As a result, hackers could have more access to these variables than if ColdFusion used different user identifiers for each session.

A hacker who has the user's `CFToken` and `CFID` cookies could gain access to user data by accessing a web page during the user's session using the stolen `CFToken` and `CFID` cookies. While this scenario is unlikely, it is theoretically possible.

You can remove this vulnerability by selecting the Use J2EE Session Variables option on the ColdFusion MX Administrator Memory Variables page. The J2EE session management mechanism creates a new session identifier for each session, and does not use either the `CFToken` or the `CFID` cookie value.

Managing client identity information in a clustered environment

To maintain your application's client identity information in a clustered server environment, you must specify This.setdomaincookies="True" in the Application.cfc initialization code, or use the `cfapplication setdomaincookies` attribute in your Application.cfm page.

The `setdomaincookies` attribute specifies that the server-side copies of the `CFID` and `CFToken` variables used to identify the client to ColdFusion are stored at the domain level (for example, .macromedia.com). If `CFID` and `CFToken` variable combinations already exist on each host in the cluster, ColdFusion migrates the host-level variables on each cluster member to the single, common domain-level variable. Following the setting or migration of host-level cookie variables to domain-level variables, ColdFusion creates a new cookie variable (`CFMagic`) that tells ColdFusion that domain-level cookies have been set.

If you use client variables in a clustered system, you must also use a database or cookies to store the variables.

Configuring and using client variables

Use client variables for data that is associated with a particular client and application and that must be saved between user sessions. Use client variables for long-term information such as user display or content preferences.

Enabling client variables

To enable client variables, you specify This.clientmanagement="True" in the Application.cfc initialization code, or set the `cfapplication` tag `clientmanagement` attribute to Yes in the Application.cfm file. For example, to enable client variables in an application named SearchApp, you can use the following line in the application's Application.cfm page:

```
<cfapplication NAME="SearchApp" clientmanagement="Yes">
```

Choosing a client variable storage method

By default, Coldfusion stores client variables in the Registry. In most cases, however, it is more appropriate to store the information as client cookies or in a SQL database.

The ColdFusion MX Administrator Client Variables page controls the default client variable location. You can override the default location by specifying a This.clientStorage value in Application.cfc or by setting the `clientStorage` attribute in the `cfapplication` tag.

You can specify the following values for the client storage method:

- `Registry`(default). Client variables are stored under the key HKEY_LOCAL_MACHINE\SOFTWARE\Macromedia\ColdFusion\CurrentVersion\Clients.
- Name of a data source configured in ColdFusion MX Administrator
- `Cookie`

Generally, it is most efficient to store client variables in a database. Although the Registry option is the default, the Registry has significant limitations for client data storage. The Registry cannot be used in clustered systems and its use for client variables on UNIX is not supported in ColdFusion MX.

Using cookie storage

When you set the client storage method to Cookie, the cookie that ColdFusion creates has the application's name. Storing client data in a cookie is scalable to large numbers of clients, but this storage mechanism has some limitations. In particular, if the client turns off cookies in the browser, client variables do not work.

Consider the following additional limitations before implementing cookie storage for client variables:

- Any Client variable that you set after a `cfflush` tag is not sent to the browser, so the variable value does not get saved.
- Some browsers allow only 20 cookies to be set from a particular host. ColdFusion uses two of these cookies for the `CFID` and `CFToken` identifiers, and also creates a cookie named `cfglobals` to hold global data about the client, such as `HitCount`, `TimeCreated`, and `LastVisit`. This limits you to 17 unique applications per client-host pair.
- Some browsers set a size limit of 4K bytes per cookie. ColdFusion encodes nonalphanumeric data in cookies with a URL encoding scheme that expands at a 3-1 ratio, which means you should not store large amounts of data per client. ColdFusion throws an error if you try to store more than 4,000 encoded bytes of data for a client.

Configuring database storage

When you specify a database for client variable storage, do not always have to manually create the data tables that store the client variables.

If ColdFusion can identify that the database you are using supports SQL creation of database tables, you only need to create the database in advance. When you click the Add button on the Select Data Source to Add as Client Store box on the Memory Variables page, the Administrator displays a Add/Edit Client Store page which contains a Create Client Database Tables selection box. Select this option to have ColdFusion create the necessary tables in your database. (The option does not appear if the database already has the required tables.)

If your database does not support SQL creation of tables, or if you are using the ODBC socket [Macromedia] driver to access your database, you must use your database tool to create the client variable tables. Create the CDATA and CGLOBAL tables.

The CDATA table must have the following columns:

Column	Data type
cfid	CHAR(64), TEXT, VARCHAR, or any data type capable of taking variable length strings up to 64 characters
app	CHAR(64), TEXT, VARCHAR, or any data type capable of taking variable length strings up to 64 characters
data	MEMO, LONGTEXT, LONG VARCHAR, CLOB, or any data type capable of taking long, indeterminate-length strings

The CGLOBAL table must have the following columns:

Column	Data type
cfid	CHAR(64), TEXT, VARCHAR, or any data type capable of taking variable length strings up to 64 characters
data	MEMO, LONGTEXT, LONG VARCHAR, CLOB, or any data type capable of taking long, indeterminate-length strings
lvisit	TIMESTAMP, DATETIME, DATE, or any data type that stores date and time values

Note: Different databases use different names for their data types. The names in the preceding tables are common, but your database might use other names.

To improve performance, you should also create indexes when you create these tables. For the CDATA table, index these cfid and app columns. For the CGLOBAL table, index the cfid column.

Specifying client variable storage in your application

The override the default client variable storage location, set the This.clientstorage variable in the Application.cfc initialization code, or use the `cfapplication` tag `clientStorage` attribute.

The following lines from an Application.cfc file tell ColdFusion to store the client variables in the mydatasource data source:

```
<cfscript>
  This.name"SearchApp";
  This.clientManagement="Yes";
```

```
  This.clientStorage="mydatasource";
</cfscript>
```

The following code from an Application.cfm file does the same thing as the previous example:

```
<cfapplication name"SearchApp"
  clientmanagement="Yes"
  clientstorage="mydatasource">
```

Using client variables

When you enable client variables for an application, you can use them to keep track of long-term information that is associated with a particular client.

Client variables must be simple data types: strings, numbers, lists, Booleans, or date and time values. They cannot be arrays, record sets, XML objects, query objects, or other objects. If you must store a complex data type as a client variable, you can use the `cfwddx` tag to convert the data to WDDX format (which is represented as a string), store the WDDX data, and use the `cfwddx` tag to convert the data back when you read it. For more information on using WDDX, see Chapter 35, "Using WDDX," on page 879.

Note: When saving client variable data in WDDX format, in the case of the registry and SQL Server, the limit is about 4K; with ORACLE, the limit is about 2K.

Creating a client variable

To create a client variable and set its value, use the `cfset` or `cfparam` tag and use the Client scope identifier as a variable prefix; for example:

```
<cfset Client.FavoriteColor="Red">
```

After you set a client variable this way, it is available for use within any page in your application that is accessed by the client for whom the variable is set.

The following example shows how to use the `cfparam` tag to check for the existence of a client parameter and set a default value if the parameter does not already exist:

```
<cfparam name="Client.FavoriteColor" default="Red">
```

Accessing and changing client variables

You use the same syntax to access a client variable as for other types of variables. You can use client variables anywhere you use other ColdFusion variables.

To display the favorite color that has been set for a specific user, for example, use the following code:

```
<cfoutput>
  Your favorite color is #Client.FavoriteColor#.
</cfoutput>
```

To change the client's favorite color, for example, use code such as the following:

```
<cfset Client.FavoriteColor = Form.FavoriteColor>
```

Standard client variables

The Client scope has the following built-in, read-only variables that your application can use:

Variable	Description
Client.CFID	The client ID, normally stored on the client system as a cookie.
Client.CFToken	The client security token, normally stored on the client system as a cookie.
Client.URLToken	Value depends on whether J2EE session management is enabled. No session management or ColdFusion session management: A combination of the `CFID` and `CFToken` values, in the form `CFID=`*`IDNum`*`&CFTOKEN=`*`tokenNum`*. This variable is useful if the client does not support cookies and you must pass the `CFID` and `CFToken` variables from page to page. J2EE session management: A combination of `CFID`, `CFToken`, and session ID values in the form `CFID=`*`IDNum`*`&CFTOKEN=`*`tokenNum`*`&jsessionid=`*`SessionID`*.
Client.HitCount	The number of page requests made by the client.
Client.LastVisit	The last time the client visited the application.
Client.TimeCreated	The time the `CFID` and `CFToken` variables that identify the client to ColdFusion were first created.

Note: ColdFusion lets you delete or change the values of the built-in client variables. As a general rule, avoid doing so.

You use the `Client.CFID`, `Client.CFToken`, and `Client.URLToken` variables if your application supports browsers that do not allow cookies. For more information on supporting browsers that do not allow cookies, see "Using client and session variables without cookies" on page 344.

You can use the `Client.HitCount` and time information variables to customize behavior that depends on how often users visit your site and when they last visited. For example, the following code shows the date of a user's last visit to your site:

```
<cfoutput>
  Welcome back to the Web SuperShop. Your last
  visit was on #DateFormat(Client.LastVisit)#.
</cfoutput>
```

Getting a list of client variables

To obtain a list of the custom client parameters associated with a particular client, use the `GetClientVariablesList` function, as follows:

```
<cfoutput>#GetClientVariablesList()#</cfoutput>
```

The `GetClientVariablesList` function returns a comma-separated list of the names of the client variables for the current application. The standard system-provided client variables (`CFID`, `CFToken`, `URLToken`, `HitCount`, `TimeCreated`, and `LastVisit`) are not returned in the list.

Deleting client variables

To delete a client variable, use the `StructDelete` function or the `DeleteClientVariable` function. For example, the following lines are equivalent:

```
<cfset IsDeleteSuccessful=DeleteClientVariable("MyClientVariable")>
<cfset IsDeleteSuccessful=StructDelete(Client, "MyClientVariable")>
```

The Client Variables page in the ColdFusion MX Administrator lets you set a time-out period of inactivity after which ColdFusion removes client variables stored in either the Registry or a data source. (The default value is 10 days for client variables stored in the Registry, and 90 days for client variables stored in a data source.)

Note: You cannot delete the system-provided client variables (`CFID`, `CFToken`, `URLToken`, `HitCount`, `TimeCreated`, and `LastVisit`).

Using client variables with cflocation

If you use the `cflocation` tag to redirect ColdFusion to a path that ends with .dbm or .cfm, the `Client.URLToken` variable is automatically appended to the URL. You can prevent this behavior by adding the attribute `addtoken="No"` to the `cflocation` tag.

Caching client variable

When ColdFusion reads or writes client variables, it caches the variables in memory to help decrease the overhead of accessing the client data. As a result, ColdFusion only accesses the client data store when you read its value for the first time or, for values you set, when the request ends. Additional references to the client variable use the cached value in ColdFusion memory, thereby processing the page more quickly.

Exporting the client variable database

If your client variable database is stored in the Windows system Registry and you need to move it to another machine, you can export the Registry key that stores your client variables and take it to your new server. The system Registry lets you export and import Registry entries.

To export your client variable database from the Registry in Windows:

1. Open the Registry editor.
2. Find and select the following key:

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Macromedia\ColdFusion\CurrentVersion\
     Clients
   ```

3. On the Registry menu, click Export Registry File.
4. Enter a name for the Registry file.

After you create a Registry file, you can copy it to a new machine and import it by clicking Import Registry File on the Registry editor Registry menu.

Note: On UNIX systems, the registry entries are kept in /opt/coldfusion/registry/cf.registry, a text file that you can copy and edit directly.

Configuring and using session variables

Use session variables when you need the variables for a single site visit or set of requests. For example, you might use session variables to store a user's selections in a shopping cart application. (Use client variables if you need a variable in multiple visits.)

Caution: Put code that uses session variables inside `cflock` tags in circumstances that might result in race conditions from multiple accesses to the same variable. For information on using cflock tags see "Locking code with cflock" on page 360.

What is a session?

A **session** refers to all the connections that a single client might make to a server in the course of viewing any pages associated with a given application. Sessions are specific to both the individual user and the application. As a result, every user of an application has a separate session and has access to a separate set of session variables.

This logical view of a session begins with the first connection to an application by a client and ends after that client's last connection. However, because of the stateless nature of the web, it is not always possible to define a precise point at which a session ends. A session should end when the user finishes using an application. In most cases, however, a web application has no way of knowing if a user has finished or is just lingering over a page.

Therefore, sessions always terminate after a time-out period of inactivity. If the user does not access a page of the application within this time-out period, ColdFusion interprets this as the end of the session and clears any variables associated with that session.

The default time-out for session variables is 20 minutes. You can change the default time-out on the Memory Variables page in the Server Settings area in the ColdFusion MX Administrator.

You can also set the time-out period for session variables inside a specific application (thereby overruling the Administrator default setting) by setting the Application.cfc This.sessionTimeout variable or by using the `cfapplication` tag `sessionTimeout` attribute. However, you cannot set a time-out value for that is greater than the maximum session time-out value set on the Administrator Memory Variables page.

For detailed information on ending sessions and deleting session variables, see "Ending a session" on page 356.

ColdFusion and J2EE session management

The ColdFusion server can use either of the following types of session management:

- ColdFusion session management
- J2EE servlet session management

ColdFusion session management uses the same client identification method as ColdFusion client management.

J2EE session management provides the following advantages over ColdFusion session management:

- J2EE session management uses a session-specific session identifier, `jsessionid`, which is created afresh at the start of each session.
- You can share session variables between ColdFusion pages and JSP pages or Java servlets that you call from the ColdFusion pages.
- The session automatically ends when the user closes all browser windows.
- The Session scope is serializable (convertible into a sequence of bytes that can later be fully restored into the original object). With ColdFusion session management, the Session scope is not serializable. Only serializable scopes can be shared across servers.

Therefore, consider using J2EE session management in any of the following cases:

- You want to maximize session security, particularly if you also use client variables
- You want to share session variables between ColdFusion pages and JSP pages or servlets in a single application.
- You want to be able to manually terminate a session while maintaining the client identification cookie for use by the Client scope.
- You want to support clustered sessions; for example, to support session failover among servers.

Configuring and enabling session variables

To use session variables, you must enable them in two places:

- ColdFusion MX Administrator
- The Application.cfc initialization code This.sessionManagement variable or the active `cfapplication` tag.

ColdFusion MX Administrator, Application.cfc, and the `cfapplication` tag also provide facilities for configuring session variable behavior, including the variable time-out.

Selecting and enabling session variables in ColdFusion MX Administrator

To use session variables, they must be enabled on the ColdFusion MX Administrator Memory Variables page. (They are enabled by default.) You can also use the Administrator Memory Variables page to do the following:

- Select to use ColdFusion session management (the default) or J2EE session management.
- Change the default session time-out. Application code can override this value. The default value for this time-out is 20 minutes.
- Specify a maximum session time-out. Application code cannot set a time-out greater than this value. The default value for this time-out is two days.

Enabling session variables in your application

You must also enable session variables in the initialization code of your Application.cfc file or in the `cfapplication` tag in your Application.cfm file.

Do the following in the Application.cfc initialization code, below the `cfcomponent` tag, to enable session variables:

- Set This.sessionManagement="Yes".
- Set This.name to specify the application's name.
- Optionally, set This.sessionTimeout to set an application-specific session time-out value. Use the `CreateTimeSpan` function to specify the number of days, hours, minutes, and seconds for the time-out.

Do the following in the Application.cfm file to enable session variables:

- Set `sessionManagement="Yes"`
- Use the `name` attribute to specify the application's name.
- Optionally, use the `sessionTimeout` attribute to set an application-specific session time-out value. Use the `CreateTimeSpan` function to specify the number of days, hours, minutes, and seconds for the time-out.

The following sample code enables session management for the GetLeadApp application and sets the session variables to time out after a 45-minute period of inactivity:

```
<cfapplication name="GetLeadApp"
  sessionmanagement="Yes"
  sessiontimeout=#CreateTimeSpan(0,0,45,0)#>
```

Storing session data in session variables

Session variables are designed to store session-level data. They are a convenient place to store information that all pages of your application might need during a user session, such as shopping cart contents or score counters.

Using session variables, an application can initialize itself with user-specific data the first time a user accesses one of the application's pages. This information can remain available while that user continues to use that application. For example, you can retrieve information about a specific user's preferences from a database once, the first time a user accesses any page of an application. This information remains available throughout that user's session, thereby avoiding the overhead of retrieving the preferences repeatedly.

Standard session variables

If you use ColdFusion session variables, the Session scope has four built-in, read-only variables that your application can use. If you use J2EE session management, the Session scope has two built-in variables. Generally, you use these variables in your ColdFusion pages only if your application supports browsers that do not allow cookies. For more information on supporting browsers that do not allow cookies, see "Using client and session variables without cookies" on page 344. The following table describes the built-in session variables.

Variable	Description
Session.CFID	ColdFusion session management only: the client ID, normally stored on the client system as a cookie.
Session.CFToken	ColdFusion session management only: the client security token, normally stored on the client system as a cookie.
Session.URLToken	ColdFusion session management: A combination of the `CFID` and `CFToken` values in the form `CFID=`*`IDNum`*`&CFTOKEN=`*`tokenNum`*. Use this variable if the client does not support cookies and you must pass the `CFID` and `CFToken` variables from page to page. J2EE session management: A combination of the `CFID` and `CFToken` cookies and the J2EE session ID, in the form `CFID=`*`IDNum`*`&CFTOKEN=`*`tokenNum`*`&jsessionid=`*`SessionID`*.
Session.SessionID	A unique identifier for the session. ColdFusion session management: a combination of the application name and `CFID` and `CFToken` values. J2EE session management: the `jsessionid` value.

Note: ColdFusion lets you delete or change the values of the built-in session variables. As a general rule, avoid doing so.

If you enable client variables and ColdFusion session management, ColdFusion uses the same values for the Client and Session scope `CFID`, `CFToken`, and `URLtoken` variables. ColdFusion gets the values for these variables from the same source, the client's `CFID` and `CFTOKEN` cookies.

If you use J2EE session management, the Session scope does not include the `Session.CFID` or `Session.CFToken` variables, but does include the `Session.URLToken` and `Session.SessionID` variables. In this case, the `Session.SessionID` is the J2EE session ID and `Session.URLToken` consists of the string `jsessionid=` followed by the J2EE session ID.

Getting a list of session variables

Use the `StructKeyList` function to get a list of session variables, as follows:

```
<cflock timeout=20 scope="Session" type="Readonly">
  <cfoutput> #StructKeyList(Session)# </cfoutput>
</cflock>
```

Caution: Always put code that accesses session variables inside `cflock` tags.

Creating and deleting session variables

Use a standard assignment statement to create a new session variable, as follows:

```
<cflock timeout=20 scope="Session" type="Exclusive">
  <cfset Session.ShoppingCartItems = 0>
</cflock>
```

Use the `structdelete` tag to delete a session variable; for example:

```
<cflock timeout=20 scope="Session" type="Exclusive">
  <cfset StructDelete(Session, "ShoppingCartItems")>
</cflock>
```

Note: If you set session variables on a CFML template that uses the `cflocation` tag, ColdFusion might not set the variables. For more information, see Macromedia TechNote 22712 at www.macromedia.com/v1/Handlers/index.cfm?ID=22712&Method=Full.

Accessing and changing session variables

You use the same syntax to access a session variable as for other types of variables. However, you must lock any code that accesses or changes session variables.

For example, to display the number of items in a user's shopping cart, use favorite color that has been set for a specific user, for example, use the following code:

```
<cflock timeout=20 scope="Session" type="Exclusive">
  <cfoutput>
    Your shopping cart has #Session.ShoppingCartItems# items.
  </cfoutput>
</cflock>
```

To change increase the number of items in the shopping cart, use the following code:

```
<cflock timeout=20 scope="Session" type="Exclusive">
  <cfset Session.ShoppingCartItems = Session.ShoppingCartItems + 1>
</cflock>
```

Ending a session

The following rules apply to ending a session and deleting Session scope variables:

- If you use ColdFusion session management, ColdFusion automatically ends sessions and deletes all Session scope variables if the client is inactive for the session time-out period. The session does not end when the user closes the browser.
- If you use J2EE session management, ColdFusion MX ends the session and deletes all Session scope variables when the user closes the browser, or if the client is inactive for the session time-out period. If the session times out, however, the browser continues to send the same session ID, and ColdFusion will reuse this ID for sessions with this browser instance, as long as the browser remains active.
- Logging a user out does not end the session or delete Session scope variables.
- In many cases, you can effectively end a session by clearing the Session scope, as shown in the following line. The following list, however, includes important limitations and alternatives:

  ```
  <cfset StructClear(Session)>
  ```

- Clearing the Session scope does not clear the session ID, and future requests from the browser continue to use the same session ID until the browser exits. It also does not log the user out, even if you use Session scope storage for login information. Always use the `cflogout` tag to log users out.
- If you use J2EE session management, you can invalidate the session, as follows:

  ```
  <cfset getPageContext().getSession().invalidate()>
  ```

  This line creates a pointer to the servlet page context and calls an internal method to reset the session. This clears all session information, including the session ID Session scope variables, and if you are using session login storage, the login information, for future request. However, the session information does remain available until the end of the current request. After you invalidate a session, attempts by the browser to access the application will generate an invalid session exception until the session times out.

  Note: You cannot destroy the session and create a session on the same request, as creating a new session involves sending session cookies back.

- If you do not use client cookies, the Session scope and login state is available to your application only as long as you pass the session's CFID, CFTOKEN, and, for J2EE sessions, jsessionid values in the URL query string. After you stop using these values, however, the session data remains in memory until the session time-out period elapses.

Configuring and using application variables

Application variables are available to all pages within an application, that is, pages that have the same application name. Because application variables are persistent, you easily can pass values between pages. You can use application variables for information including the application name, background color, data source names, or contact information.

You set the application name in the `cfapplication` tag, normally on your application's Application.cfm page. The application name is stored in the `Application.applicationName` variable.

Unlike client and session variables, application variables do not require that a client name (client ID) be associated with them. They are available to any clients that use pages in the application.

Caution: Put code that uses application variables inside `cflock` tags in circumstances that might result in race conditions from multiple accesses to the same variable. For information on using `cflock` tags, see "Locking code with cflock" on page 360.

The following sections describe how to configure and use application variables.

Configuring and enabling application variables

To use application variables, do the following:

- Ensure that they are enabled in the ColdFusion MX Administrator. (They are enabled by default.)
- Specify the application name by setting the This.name variable in the initialization code of the Application.cfc or by setting the `name` attribute of the `cfapplication` tag for the current page.

  Note: ColdFusion supports unnamed applications for compatibility with J2EE applications. For more information, see Chapter 37, "Unnamed ColdFusion Application and Session scopes," on page 924.

The ColdFusion MX Administrator also lets you specify the following information:

- A default variable time-out. If all pages in an application are inactive for the time-out period, ColdFusion deletes all the application variables. The Application.cfc file or `cfapplication` tag can override this value for a specific application. The default value for this time-out is two days.
- A maximum time-out. The application code cannot set a time-out greater than this value. The default value for this time-out is two days.

You can set the time-out period for application variables within a specific application by using the This.applicationTimeout variable of Application.cfc or the `applicationTimeout` attribute of the `cfapplication` tag.

Storing application data in application variables

Application variables are a convenient place to store information that all pages of your application might need, no matter which client is running that application. Using application variables, an application could, for example, initialize itself when the first user accesses any page of that application. This information can then remain available indefinitely, thereby avoiding the overhead of repeated initialization.

Because the data stored in application variables is available to all pages of an application, and remains available until a specific period of inactivity passes or the ColdFusion server shuts down, application variables are convenient for application-global, persistent data.

However, because all clients running an application see the same set of application variables, these variables are not appropriate for client-specific or session-specific information. To target variables for specific clients, use client or session variables.

Using application variables

Generally, application variables should hold information that you write infrequently. In most cases, the values of these variables are set once, most often when an application first starts. Then the values of these variables are referenced many times throughout the life of the application or the course of a session.

In circumstances that might result in race conditions from multiple accesses to the same variable, put code that writes to Application scope variables or reads Application scope variables with data that can change inside `cflock` tags.

Because each Application scope variable is shared in memory by all requests in the application, these variables can become bottlenecks if used inappropriately. Whenever a request is reading or writing an Application scope variable, any other requests that use the variable must wait until the code accessing the variable completes. This problem is increased by the processing time required for locking. If many users access the application simultaneously and you use Application scope variables extensively, your application performance might degrade. If your application uses many application variables, consider whether the variables must be in the Application scope or whether they can be Session or Request scope variables.

The application scope has one built-in variable, `Application.applicationName`, which contains the application name you specify in the `cfapplication` tag.

You access and manipulate application variables the same way you use session variables, except that you use the variable prefix Application, not Session, and specify Session as the lock scope. For examples of using session variables see "Creating and deleting session variables" on page 356 and "Accessing and changing session variables" on page 356.

For information on locking write-once read-many application variables efficiently, see "Locking application variables efficiently" on page 367

Using server variables

Server variables are associated with a single ColdFusion server. They are available to all applications that run on the server. Use server variables for data that must be accessed across clients and applications, such as global server hit counts.

Server variables do not time out, but they are lost when the server shuts down. You can delete server variables.

Server variables are stored on a single server. As a result, you should not use server variables if you use ColdFusion on a server cluster.

You access and manipulate server variables the same way use Session and application variables, except you use the variable prefix Server.

Caution: Put code that uses server variables inside `cflock` tags in circumstances that might result in race conditions from multiple accesses to the same variable. You do not have to lock access to built-in server variables.

ColdFusion provides the following standard built-in read-only server variables:

Variable	Description
Server.ColdFusion.AppServer	The name of the J2EE application server ColdFusion is using. For ColdFusion MX server editions, which have an integrated application server, the name is JRun4.
Server.ColdFusion.Expiration	The date on which the ColdFusion MX license expires if it is the trial version.
Server.ColdFusion.ProductLevel	The server product level, such as Enterprise.
Server.ColdFusion.ProductName	The name of the product (ColdFusion MX).

Variable	Description
Server.ColdFusion.ProductVersion	The version number for the server that is running, such as 6,0,0.
Server.ColdFusion.Rootdir	Directory under which ColdFusion is installed, such as C:\cfusion.
Server.ColdFusion.SerialNumber	The serial number assigned to this server installation.
Server.ColdFusion.SupportedLocales	The locales, such as English (US) and Spanish (Standard), supported by the server.
Server.OS.AdditionalInformation	Additional information provided by the operating system, such as the Service Pack number.
Server.OS.arch	The processor architecture, such as x86 for Intel Pentium processors.
Server.OS.BuildNumber	The specific operating system build, such as 1381
Server.OS.Name	The name of the operating system, such as Windows NT.
Server.OS.Version	The version number of the operating system, such as 4.0.

Locking code with cflock

The `cflock` tag controls simultaneous access to ColdFusion code. The `cflock` tag lets you do the following:

- Protect sections of code that access and manipulate shared data in the Session, Application, and Server scopes.
- Ensure that file updates do not fail because files are open for writing by other applications or ColdFusion tags.
- Ensure that applications do not try to simultaneously access ColdFusion extension tags written using the CFX API that are not thread-safe. This is particularly important for CFX tags that use shared (global) data structures without protecting them from simultaneous access (not thread-safe). However, Java CFX tags can also access shared resources that could become inconsistent if the CFX tag access is not locked.
- Ensure that applications do not try to simultaneously access databases that are not thread-safe. (This is not necessary for most database systems.)

ColdFusion MX is a multithreaded web application server that can process multiple page requests at a time. As a result, the server can attempt to access the same information or resources simultaneously, as the result of two or more requests.

Although ColdFusion MX is thread-safe and does not try to modify a variable simultaneously, it does not ensure the correct order of access to information. If multiple pages, or multiple invocations of a page, attempt to write data simultaneously, or read and write it at the same time, the resulting data can be inconsistent, as shown in the following "Sample locking scenarios" section.

Similarly, ColdFusion MX cannot automatically ensure that two sections of code do not attempt to access external resources such as files, databases, or CFX tags that cannot properly handle simultaneous requests. Nor can ColdFusion MX ensure that the order of access to these shared resources is consistent and results in valid data.

By locking code that accesses such resources so that only one thread can access the resource at a time, you can prevent race conditions.

Sample locking scenarios

The following examples present scenarios in which you need to lock ColdFusion code. These scenarios show only two of the circumstances where locking is vital.

Reading and writing a shared variable

If you have an application-wide value, such as a counter of the total number of tickets sold, you might have code such as the following on a login page:

```
<cfset Application.totalTicketsSold = Application.totalTicketsSold +
  ticketOrder>
```

When ColdFusion executes this code, it performs the following operations:

1. Retrieves the current value of Application.totalTicketsSold from temporary storage.
2. Increments this value.
3. Stores the result back in the Application scope.

Suppose that ColdFusion processes two ticket orders at approximately the same time, and that the value of Application.totalTicketsSold is initially 160. The following sequence might happen:

1. Order 1 reads the total tickets sold as 160.
2. Order 2 reads the total tickets sold as 160.
3. Order 1 adds an order of 5 tickets to 160 to get 165.
4. Order 2 adds an order of 3 tickets to 160 to get 163.
5. Order 1 saves the value 165 to Application.totalTicketsSold
6. Order 2 saves the value 163 to Application.totalTicketsSold

The application now has an inaccurate count of the tickets sold, and is in danger of selling more tickets than the auditorium can hold.

To prevent this from happening, lock the code that increments the counter, as follows:

```
<cflock scope="Application" timeout="10" type="Exclusive">
  <cfset Application.totalTicketsSold = Application.totalTicketsSold +
  ticketOrder>
</cflock>
```

The `cflock` tag ensures that while ColdFusion performs the processing in the tag body, no other threads can access the Application scope. As a result, the second transaction is not processed until the first one completes. The processing sequence looks something like the following:

1. Order 1 reaches the lock tag, which gets an Application scope lock.
2. Order 1 reads the total tickets sold as 160.
3. Order 2 reaches the lock tag. Because there is an active Application scope lock, ColdFusion waits for the lock to free.
4. Order 1 adds an order of 5 tickets to 160 to get 165.
5. Order 1 saves the value 165 to Application.totalTicketsSold.
6. Order 1 exits the lock tag. The Application scope lock is now free.
7. Order 2 gets the Application scope lock and can begin processing.
8. Order 2 reads the total tickets sold as 165.
9. Order 2 adds an order of 3 tickets to 165 to get 168.
10. Order 2 saves the value 168 to Application.totalTicketsSold.
11. Order 2 exits the lock tag, which frees the Application scope lock. ColdFusion can process another order.

The resulting Application.totalTickesSold value is now correct.

Ensuring consistency of multiple variables

Often an application sets multiple shared scope variables at one time, such as a number of values submitted by a user on a form. If the user submits the form, clicks the back button, and then resubmits the form with different data, the application might end up with a mixture of data from the two submissions, in much the same manner as shown in the previous section.

For example, an application might store information about order items in a Session scope shopping cart. If the user submits an item selection page with data specifying sage green size 36 shorts, and then resubmits the item specifying sea blue size 34 shorts, the application might end up with a mixture of information from the two orders, such as sage green size 34 shorts.

By putting the code that sets all of the related session variables in a single `cflock` tag, you ensure that all the variables get set together. In other words, setting all of the variables becomes an **atomic**, or single, operation. It is similar to a database transaction, where everything in the transaction happens, or nothing happens. In this example, the order details for the first order all get set, and then they are replaced with the details from the second order.

For more examples of using locking in applications, see "Examples of cflock" on page 368.

Using the cflock tag with write-once variables

You do not need to use `cflock` when you read a variable or call a user-defined function name in the Session, Application, or Server scope if it is set in *only one place* in the application, and is only read (or called, for a UDF) everywhere else. Such data is called *write-once*. If you set an Application or Session scope variable in Application.cfm and never set it on any other pages, you must lock the code that sets the variable, but do not have to lock code on other pages that reads the variable's value. If you set the variable in the corresponding start method in Application.cfc (for example, `onApplicationStart` for Application scope variables), you do not have to lock the code that sets the variable.

However, although leaving code that uses write-once data unlocked can improve application performance, it also has risks. You must ensure that the variables are truly written only once. For example, you must ensure that the variable is not rewritten if the user refreshes the browser or clicks a back button. Also, it can be difficult to ensure that you, or future developers, do not later set the variable in more than one place in the application.

Using the cflock tag

The `cflock` tag ensures that concurrently executing requests do not run the same section of code simultaneously and thus manipulate shared data structures, files, or CFX tags inconsistently. It is important to remember that `cflock` protects code sections that access or set data, *not* the variables themselves.

You protect access to code by surrounding it in a `cflock` tag; for example:

```
<cflock scope="Application" timeout="10" type="Exclusive">
  <cfif not IsDefined("Application.number")>
    <cfset Application.number = 1>
  </cfif>
</cflock>
```

Lock types

The `cflock` tag offers two modes of locking, specified by the `type` attribute:

Exclusive locks (the default lock type) Allow only one request to process the locked code. No other requests can run code inside the tag while a request has an exclusive lock.

Enclose all code that creates or modifies session, application, or server variables in exclusive `cflock` tags.

Read-only locks Allow multiple requests to execute concurrently if no exclusive locks with the same scope or name are executing. No requests can run code inside the tag while a request has an exclusive lock.

Enclose code that only reads or tests session, application, or server variables in read-only `cflock` tags. You specify a read-only lock by setting the `type="readOnly"` attribute in the `cflock` tag, for example:

```
<cflock scope="Application" timeout="10" type="readOnly">
  <cfif IsDefined("Application.dailyMessage")>
    <cfoutput>#Application.dailyMessage#<br></cfoutput>
```

```
  </cfif>
</cflock>
```

Although ColdFusion does not prevent you from setting shared variables inside read-only lock tag, doing so loses the advantages of locking. As a result, you must be careful not to set any session, application, or server variables inside a read-only `cflock` tag body.

Note: You cannot upgrade or downgrade a lock from one type to another. In other words, do not nest an exclusive lock in a read-only lock of the same name or scope; the exclusive lock will always time out. Also, do not nest a read-only lock inside an exclusive lock with the same name or scope; doing so has no effect.

Lock scopes and names

The `cflock` tag prevents simultaneous access to sections of code, not to variables. If you have two sections of code that access the same variable, they must be synchronized to prevent them from running simultaneously. You do this by identifying the locks with the same `scope` or `name` attributes.

Note: ColdFusion does not require you to identify exclusive locks. If you omit the identifier, the lock is anonymous and you cannot synchronize the code in the `cflock` tag block with any other code. Anonymous locks do not cause errors when they protect a resource that is used in a single code block, but they are bad programming practice. You must always identify read-only locks.

Controlling access to data with the scope attribute

When the code that you are locking accesses session, application, or server variables, synchronize access by using the `cflock scope` attribute.

You can set the attribute to any of the following values:

Scope	Meaning
Server	All code sections with this attribute on the server share a single lock.
Application	All code sections with this attribute in the same application share a single lock.
Session	All code sections with this attribute that run in the same session of an application share a single lock.

If multiple code sections share a lock, the following rules apply:

- When code is running in a `cflock` tag block with the `type` attribute set to `Exclusive`, code in `cflock` tag blocks with the same `scope` attribute is not allowed to run. They wait until the code with the exclusive lock completes.
- When code in a `cflock` tag block with the type `readOnly` is running, code in other `cflock` tag blocks with the same `scope` attribute and the `readOnly type` attribute can run, but any blocks with the same `scope` attribute and an `Exclusive` type cannot run and must wait until all code with the read-only lock completes. However, if a read-only lock is active and code with an exclusive lock with the same scope or name is waiting to execute, read-only requests using the same scope or name that are made after the exclusive request is queued must wait until code with the exclusive lock executes and completes.

Controlling locking access to files and CFX tags with the name attribute

The `cflock name` attribute provides a second way to identify locks. Use this attribute when you use locks to protect code that manges file access or calls non-thread-safe CFX code.

When you use the `name` attribute, specify the same name for each section of code that accesses a specific file or a specific CFX tag.

Controlling and minimizing lock time-outs

You must include a `timeout` attribute in your `cflock` tag. The `timeout` attribute specifies the maximum time, in seconds, to wait to obtain the lock if it is not available. By default, if the lock does not become available within the time-out period, ColdFusion generates a Lock type exception error, which you can handle using `cftry` and `cfcatch` tags.

If you set the `cflock throwOnTimeout` attribute to No, processing continues after the time-out at the line after the `</cflock>` end tag. Code in the `cflock` tag body does not run if the time-out occurs before ColdFusion can acquire the lock. Therefore, never use the `throwOnTimeout` attribute for CFML that must run.

Normally, it does not take more than a few seconds to obtain a lock. Very large time-outs can block request threads for long periods of time and radically decrease throughput. Always use the smallest time-out value that does not result in a significant number of time-outs.

To prevent unnecessary time-outs, lock the minimum amount of code possible. Whenever possible, lock only code that sets or reads variables, not business logic or database queries. One useful technique is to do the following:

1. Perform a time-consuming activity outside of a `cflock` tag
2. Assign the result to a Variables scope variable
3. Assign the Variables scope variable's value to a shared scope variable inside a `cflock` block.

For example, if you want to assign the results of a query to a session variable, first get the query results using a Variables scope variable in unlocked code. Then, assign the query results to a session variable inside a locked code section. The following code shows this technique:

```
<cfquery name="Variables.qUser" datasource="#request.dsn#">
  SELECT FirstName, LastName
  FROM Users
  WHERE UserID = #request.UserID#
</cfquery>
<cflock scope="Session" timeout="5" type="exclusive">
  <cfset Session.qUser = Variables.qUser>
</cflock>
```

Considering lock granularity

When you design your locking strategy, consider whether you should have multiple locks containing small amounts of code or few locks with larger blocks of code. There is no simple rule for making such a decision, and you might do performance testing with different options to help make your decision. However, you must consider the following issues:

- If the code block is larger, ColdFusion will spend more time inside the block, which might increase the number of times an application waits for the lock to released.
- Each lock requires processor time. The more locks you have, the more processor time is spent on locking code.

Nesting locks and avoiding deadlocks

Inconsistent nesting of `cflock` tags and inconsistent naming of locks can cause deadlocks (blocked code). If you are nesting locks, you must consistently nest `cflock` tags in the same order and use consistent lock scopes (or names).

A *deadlock* is a state in which no request can execute the locked section of the page. All requests to the protected section of the page are blocked until there is a time-out. The following table shows one scenario that would cause a deadlock:

User 1	User 2
Locks the Session scope.	Locks the Application scope.
Tries to lock the Application scope, but the Application scope is already locked by User 2.	Tries to lock the Session scope, but the Session scope is already locked by User 1.

Neither user's request can proceed, because it is waiting for the other to complete. The two are deadlocked.

Once a deadlock occurs, neither of the users can do anything to break the deadlock, because the execution of their requests is blocked until the deadlock is resolved by a lock time-out.

You can also cause deadlocks if you nest locks of different types. An example of this is nesting an exclusive lock inside a read-only lock of the same scope or same name.

In order to avoid a deadlock, lock code sections in a well-specified order, and name the locks consistently. In particular, if you need to lock access to the Server, Application, and Session scopes, you must do so in the following order:

1. Lock the Session scope. In the `cflock` tag, specify `scope="Session"`.
2. Lock the Application scope. In the `cflock` tag, specify `scope="Application"`.
3. Lock the Server scope. In the `cflock` tag, specify `scope="Server"`.
4. Unlock the Server scope.
5. Unlock the Application scope.
6. Unlock the Session scope.

Note: You can skip any pair of lock and unlock steps in the preceding list if you do not need to lock a particular scope. For example, you can omit steps 3 and 4 if you do not need to lock the Server scope.

Copying shared variables into the Request scope

You can avoid locking some shared-scope variables multiple times during a request by doing the following:

1. Copy the shared-scope variables into the Request scope in code with an exclusive lock in the Application.cfc `onRequestStart` method or the Application.cfm page.
2. Use the Request scope variables on your ColdFusion pages for the duration of the request.
3. Copy the variables back to the shared scope in code with an exclusive lock in the Application.cfc `onRequestEnd` method on the OnRequestEnd.cfm page.

With this technique the "last request wins." For example, if two requests run simultaneously, and both requests change the values of data that was copied from the shared scope, the data from the last request to finish is saved in the shared scope, and the data from the previous request is not saved.

Locking application variables efficiently

The need to lock application variables can reduce server performance, because all requests that use Application scope variables must wait on a single lock. This issue is a problem even for write-once read-many variables, because you still must ensure the variable exists, and possibly set the value before you can read it.

You can minimize this problem by using a technique such as the following to test for the existence of application variables and set them if they do not exist:

1. Use an Application scope flag variable to indicate if the variable or variables are initialized. In a read-only lock, check for the existence of the flag, and assign the result to a local variable.
2. Outside the `cflock` bock, test the value of the local variable
3. If it the local variable indicates that the application variables are not initialized, get an exclusive Application scope lock.
4. Inside the lock, again test the Application scope flag, to make sure another page has not set the variables between step one and step four.
5. If the variables are still not set, set them and set the Application scope flag to true.
6. Release the exclusive lock.

The following code shows this technique:

```
<!--- Initialize local flag to false. --->
<cfset app_is_initialized = False>
<!--- Get a readonly lock --->
<cflock scope="application" type="readonly">
  <!--- read init flag and store it in local variable --->
  <cfset app_is_initialized = IsDefined("APPLICATION.initialized")>
</cflock>
<!--- Check the local flag --->
<cfif not app_is_initialized >
<!--- Not initialized yet, get exclusive lock to write scope --->
  <cflock scope="application" type="exclusive">
    <!--- Check nonlocal flag since multiple requests could get to the
```

```
        exclusive lock --->
    <cfif not IsDefined("APPLICATION.initialized") >
      <!--- Do initializations --->
      <cfset APPLICATION.varible1 = someValue >
       ...
      <!--- Set the Application scope initialization flag --->
      <cfset APPLICATION.initialized = "yes">
    </cfif>
  </cflock>
</cfif>
```

Examples of cflock

The following examples show how to use `cflock` blocks in a variety of situations.

Example with application, server, and session variables

This example shows how you can use `cflock` to guarantee the consistency of data updates to variables in the Application, Server, and Session scopes.

This example does not handle exceptions that arise if a lock times out. As a result, users see the default exception error page on lock time-outs.

The following sample code might be part of the Application.cfm file:

```
<cfapplication name="ETurtle"
  sessiontimeout=#createtimespan(0,1,30,0)#
  sessionmanagement="yes">

<!--- Initialize the Session and Application
variables that will be used by E-Turtleneck. Use
the Session lock scope for the session variables. --->

<cflock scope="Session"
  timeout="10" type ="Exclusive">
  <cfif not IsDefined("session.size")>
    <cfset session.size = "">
  </cfif>
  <cfif not IsDefined("session.color")>
    <cfset session.color = "">
  </cfif>
</cflock>

<!--- Use the Application scope lock for the Application.number variable.
This variable keeps track of the total number of turtlenecks sold.
The following code implements the scheme shown in the Locking Application
variables effectively section --->

<cfset app_is_initialized = "no">
<cflock scope="Application" type="readonly">
  <cfset app_is_initialized = IsDefined("Application.initialized")>
</cflock>
<cfif not app_is_initialized >
  <cflock scope="application" timeout="10" type="exclusive">
    <cfif not IsDefined("Application.initialized") >
```

```
      <cfset Application.number = 1>
      <cfset Application.initialized = "yes">
    </cfif>
  </cflock>
</cfif>

<!--- Always display the number of turtlenecks sold --->

<cflock scope="Application"
  timeout="10"
  type ="ReadOnly">
  <cfoutput>
  E-Turtleneck is proud to say that we have sold
  #Application.number# turtlenecks to date.
  </cfoutput>
</cflock>
```

The remaining sample code could appear inside the application page where customers place orders:

```
<html>
<head>
<title>cflock Example</title>
</head>

<body>
<h3>cflock Example</h3>

<cfif IsDefined("Form.submit")>

<!--- Lock session variables --->
<!--- Note that we use the automatically generated Session
  ID as the order ID --->
<cflock scope="Session"
  timeout="10" type="ReadOnly">
  <cfoutput>Thank you for shopping E-Turtleneck.
  Today you have chosen a turtleneck in size
  <b>#form.size#</b> and in the color <b>#form.color#</b>.
  Your order ID is #Session.sessionID#.
  </cfoutput>
</cflock>

<!--- Lock session variables to assign form values to them. --->

<cflock scope="Session"
  timeout="10"
  type="Exclusive">
  <cfparam name=Session.size default=#form.size#>
  <cfparam name=Session.color default=#form.color#>
</cflock>
<
!--- Lock the Application scope variable application.number to
update the total number of turtlenecks sold. --->
```

```
<cflock scope="Application"
  timeout="30" type="Exclusive">
  <cfset application.number=application.number + 1>
</cflock>

<!--- Show the form only if it has not been submitted. --->
<cfelse>
<form action="cflock.cfm" method="Post">

<p> Congratulations! You have just selected
the longest-wearing, most comfortable turtleneck
in the world. Please indicate the color and size
you want to buy.</p>

<table cellspacing="2" cellpadding="2" border="0">
<tr>
<td>Select a color.</td>
<td><select type="Text" name="color">
     <option>red
     <option>white
     <option>blue
     <option>turquoise
     <option>black
     <option>forest green
     </select>
  </td>
</tr>
<tr>
  <td>Select a size.</td>
  <td><select type="Text" name="size">
    <option>small
    <option>medium
    <option>large
    <option>xlarge
    </select>
  </td>
</tr>
<tr>
  <td></td>
  <td><input type="Submit" name="submit" value="Submit">
  </td>
</tr>
</table>
</form>
</cfif>

</body>
</html>
```

Note: In this simple example, the Application.cfm page displays the Application.number variable value. Because the Application.cfm file is processed before any code on each ColdFusion page, the number that displays after you click the submit button does not include the new order. One way you can resolve this problem is by using the OnRequestEnd.cfm page to display the value at the bottom of each page in the application.

Example of synchronizing access to a file system

The following example shows how to use a `cflock` block to synchronize access to a file system. The `cflock` tag protects a `cffile` tag from attempting to append data to a file already open for writing by the same tag executing on another request.

If an append operation takes more than 30 seconds, a request waiting to obtain an exclusive lock to this code might time out. Also, this example uses a dynamic value for the `name` attribute so that a different lock controls access to each file. As a result, locking access to one file does not delay access to any other file.

```
<cflock name=#filename# timeout=30 type="Exclusive">
  <cffile action="Append"
    file=#fileName#
    output=#textToAppend#>
</cflock>
```

Example of protecting ColdFusion extensions

The following example shows how you can build a custom tag wrapper around a CFX tag that is not thread-safe. The wrapper forwards attributes to the non-thread-safe CFX tag that is used inside a `cflock` tag.

```
<cfparam name="Attributes.AttributeOne" default="">
<cfparam name="Attributes.AttributeTwo" default="">
<cfparam name="Attributes.AttributeThree" default="">

<cflock timeout=5
    type="Exclusive"
    name="cfx_not_thread_safe">
  <cfx_not_thread_safe attributeone=#attributes.attributeone#
    attributetwo=#attributes.attributetwo#
    attributethree=#attributes.attributethree#>
</cflock>
```

CHAPTER 16
Securing Applications

This chapter provides an overview of Macromedia ColdFusion security features, including brief descriptions of several ColdFusion security features and references to additional documentation not included in this chapter.

This chapter includes a summary of *resource security* (ColdFusion MX Standard) or *sandbox security* (ColdFusion MX Enterprise), where security restricts access to specific resources, such as tags and files. The information describes how you use the ColdFusion MX Administrator to configure sandbox or resource security, and discusses structuring an application to take advantage of this security.

This chapter also includes detailed description of *user security*, which depends on a user identity. It explains in detail how to implement user security in ColdFusion applications.

Other chapters discuss specific security issues as part of the context of their topics. For links to some of the more significant security features, see the cross references in "ColdFusion security features" on page 374. Also see the security entries in the Index. For detailed information on using Administrator-controlled security features, see *Configuring and Administering ColdFusion MX*.

This chapter does not discuss general security issues or web server security concepts and issues in any detail. For example, it does not discuss web server security management issues, such as enabling HTTPS protocol support. For information on enabling web server security features, see your web server documentation. Many books and other resources are available on web and application security.

Contents

ColdFusion security features 374
About resource and sandbox security 375
About user security 376
Using ColdFusion security tags and functions 381
Security scenarios 386
Implementing user security 388

ColdFusion security features

ColdFusion provides scalable, granular security for building and deploying your ColdFusion applications.

ColdFusion provides the following types of security resources:

Development ColdFusion MX Administrator is protected by a password. Additionally, you can specify a password for access to data sources from Macromedia Dreamweaver MX. For more information on configuring Administrator security passwords, see the ColdFusion MX Administrator online Help.

CFML features The CFML language includes specific features that you can use to enhance application security. These include the following features:

- `The cfqueryparam` tag This tag helps prevent users from injecting malicious SQL expressions. For more information on using this tag for database security, see "Enhancing security with cfqueryparam" on page 472,
- Scriptprotect setting This setting helps protect against cross-site scripting attacks. You can set this value with the ColdFusion MX Administrator Enable Global Script Protection setting, in the Application.cfc This.scriptprotect variable, or in the corresponding `cfapplication` tag `scriptprotect` attribute. For more information on this feature, see `cfapplication` in *CFML Reference*. For information on Application.cfc see "Defining the application and its event handlers in Application.cfc" on page 282.
- Encryption and hashing functions The `Encrypt`, `Decrypt`, and `Hash` functions let you select a secure algorithm for encrypting and decrypting data or generating a hash "fingerprint." You can select from among several secure algorithms that are supported by the underlying Java security mechanisms; for encryption, these include, AES, Blowfish, DES and Triple DES. For more information, see the `Encrypt`, `Decrypt`, and `Hash`, functions in *CFML Reference*.
- Data validation tools ColdFusion includes a variety of tools for validating form input and other data values, including ways to ensure that users do not submit malicious form data. For information on data validation see Chapter 28, "Validating Data," on page 659; for specific information on security and validation, see "Security considerations" on page 664.

Note: This chapter does not discuss development or CFML language security features. For additional information on security in ColdFusion programs, see the security entries in the index.

Resource/Sandbox The ColdFusion MX Administrator can limit access to ColdFusion resources, including selected tags and functions, data sources, files, and host addresses. In the Standard Edition, you configure a single set of resource limitations that apply to all your ColdFusion applications.

In the Enterprise Edition, you can have multiple sandboxes, based on the location of your ColdFusion pages, each with its own set of resource limitations. You can confine applications to secure areas, thereby flexibly restricting the access that the application has to resources.

User ColdFusion applications can require users to log in to use application pages. You can assign users to roles (sometimes called groups); ColdFusion pages can determine the logged-in user's roles or ID and selectively determine what to do based on this information. User security is also called authentication and authorization security.

Note: You can also use the `cfencode` utility, located in the *cf_root*/bin directory, to obscure ColdFusion pages that you distribute. Although this technique cannot prevent persistent hackers from determining the contents of your pages, it does prevent inspection of the pages.

About resource and sandbox security

ColdFusion provides two levels of resource-based security:

- ColdFusion MX Standard refers to its resource-based security as resource security. It lets you specify a single set of limitations on access to ColdFusion resources that apply to all ColdFusion applications.
- ColdFusion MX Enterprise refers to its resource-based security as sandbox security. Sandbox security is a superset of resource security. Sandbox security lets you create multiple *sandboxes*, each corresponding to a different directory. For each sandbox, you specify a set of resource limitations that apply to all ColdFusion pages in the sandbox directory and its subdirectories. If you create a sandbox that is a subdirectory of a sandbox, the subdirectory's rules override the parent directory's rules.

The ColdFusion MX Administrator Resource Security page (in Standard) and Sandbox Security page (in Enterprise) let you enable resource-based security. In ColdFusion Standard, the page lets you configure the resource settings that apply to all your ColdFusion applications. In ColdFusion Enterprise, the page lets you create sandboxes and configure the resource limitations for each sandbox individually.

Resource control

ColdFusion lets you control access to the following resources:

Resource	Description
Data sources	Enables access to specified data sources.
CF tags	Prevents pages from using CFML tags that access external resources. You can prevent pages in the directory from using any or all of the following tags: `cfcollection`, `cfcontent`, `cfcookie`, `cfdirectory`, `cfdocument`, `cfexecute`, `cffile`, `cfftp`, `cfgridupdate`, `cfhttp`, `cfhttpparam`, `cfindex`, `cfinsert`, `cfinvoke`, `cfldap`, `cflog`, `cfmail`, `cfobject`, `cfobjectcache`, `cfpop`, `cfquery`, `cfregistry`, `cfreport`, `cfschedule`, `cfsearch`, `cfstoredproc`, `cftransaction`, `cfupdate`
CF functions	Prevents pages from using CFML functions that access external resources. You can prevent pages from using any or all of the following functions: `CreateObject` (COM), `CreateObject` (Java), `CreateObject` (Web Service), `DirectoryExists`. `ExpandPath`, `FileExists`, `GetBaseTemplatePath`, `GetDirectoryFromPath`, `GetFileFromPath`, `GetGatewayHelper`, `GetProfileString`, `GetTempDirectory`, `GetTempFile`, `GetTemplatePath`, `SendGatewayMessage`, `SetProfileString`

Resource	Description
Files/directories	Sets read, write, execute, and delete access to specified directories, directory trees, or files.
Server/ports	Controls access from ColdFusion to IP addresses and port numbers. You can specify host names or numeric addresses, and you can specify individual ports and port ranges..

Note: For more information on configuring resource and sandbox security, see *Configuring and Administering ColdFusion MX* and the ColdFusion MX Administrator online Help.

Sandbox security

In ColdFusion Enterprise, sandbox security lets you apply different sets of rules to different directory structures. You can use it to partition a shared hosting environment so that a number of applications with different purposes, and possibly different owners, run securely on a single server. When multiple applications share a host, you set up a separate directory structure for each application, and apply rules that let each application access only its own data sources and files.

Sandbox security also lets you structure and partition an application to reflect the access rights that are appropriate to different functional components. For example, if your application has both employee inquiry functions and HR functions that include creating, accessing, and modifying sensitive data, you could structure the application as follows:

- HR pages go in one directory with access rules that enable most activities.
- Employee pages go in another directory whose rules limit the files they can modify and the tags they can use.
- Pages required for both HR and employee functions go in a third directory with appropriate access rules.

About user security

User security lets your application use security rules to determine what it shows. It has two elements:

Authentication Ensures that a valid user is logged-in, based on an ID and password provided by the user. ColdFusion (or, in some cases if you use web server authentication, the web server) maintains the user ID information while the user is logged-in.

Authorization Ensures that the logged-in user is allowed to use a page or perform an operation. Authorization is typically based on one or more *roles* (sometimes called groups) to which the user belongs. For example, in an employee database, all users could be members of either the employee role or the contractor role. They could also be members of roles that identify their department, position in the corporate hierarchy, or job description. For example, someone could be a member of some or all of the following roles:

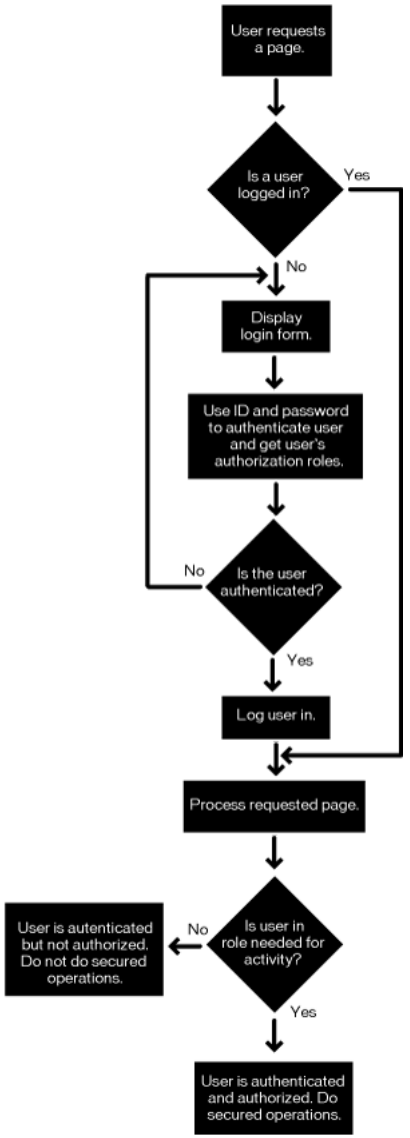
- Employees
- Human Resources
- Benefits
- Managers

Roles enable you to control access in your application resources without requiring the application to maintain knowledge about individual users. For example, suppose you use ColdFusion for your company's intranet. The Human Resources department maintains a page on the intranet on which all employees can access timely information about the company, such as the latest company policies, upcoming events, and job postings. You want everyone to be able to read the information, but you want only certain authorized Human Resources employees to be able to add, update, or delete information.

Your application gets the user's roles from the user information data store when the user logs in, and then enables access to specific pages or features based on the roles. Typically, you store user information in a database, LDAP directory, or other secure information store.

You can also use the user ID for authorization. For example, you might want to let employees view customized information about their salaries, job levels, and performance reviews. You certainly would not want one employee to view sensitive information about another employee, but you would want managers to be able to see, and possibly update, information about their direct reports. By employing both user IDs and roles, you can ensure that only the appropriate people can access or work with sensitive data.

The following figure shows a typical flow of control for user authentication and authorization. Following sections expand on this diagram to describe how you implement user security in ColdFusion.



Authenticating users

You can use either, or both, of the following forms of authentication to secure your ColdFusion application:

- Web server authentication, where the web server authenticates the user and does not allow access to the website by users without valid login IDs
- Application authentication, where the ColdFusion application authenticates the user and does not allow access to the application by users without valid login IDs

About web server authentication

All major web servers support basic HTTP authentication. Some web servers also support other authentication methods, including Digest HTTP authentication and Microsoft NTLM authentication.

Note: Macromedia Dreamweaver MX and Studio MX do not support NTLM security with RDS. Therefore, you cannot use RDS with these applications if the ColdFusion RDS servlet (*cf_root*/CFIDE/main/ide.cfm) is in a directory that is protected using NTLM security.

In web server authentication, the web server requires the user to log in to access pages in a particular directory, as follows:

1. When the user first requests a page in the secured directory, the web server notifies the browser that the requested page requires credentials (a user ID and password).

   Basic HTTP authentication sends the user ID and password in a base64-encoded string with each request. Use SSL (Secure Sockets Layer) for all page transactions, to protect the user ID and password from unauthorized access. For more information on SSL and the keytool utility, see "About LDAP Server Security" on page 544.
2. The browser prompts the user for the credentials.
3. The user supplies the credentials and the browser send the information back to the web server along with the original request.
4. The web server checks the user ID and password, using its own user authentication mechanism.
5. If the user logs in successfully, the browser caches the authentication information and sends it in an HTTP Authorization header with every subsequent page request from the user.
6. The web server processes the requested page and all future page requests from the browser that contain the HTTP Authorization header, if it is valid for the requested page.

You can use web server authentication without using any ColdFusion security features. In this case, you configure and manage all user security through the web server's interfaces.

You can also use web server authentication with ColdFusion application authentication, and thus you can use ColdFusion security for authorization. If the web server uses basic HTML authentication, the ColdFusion `cflogin` tag provides access to the user ID and password that the user entered to log in to the web server. If the web server uses Digest or NTLM authentication, the `cflogin` tag normally gets the user ID, but not the password.

As a result, your application can rely on the web server to authenticate the user against its user and password information, and does not have to display a login page. You use the `cflogin` and `cfloginuser` tags to log the user into the ColdFusion user security system, and use the `IsUserInRole` and `GetAuthUser` functions to ensure user authorization. For more information on this form of security, see "A web server authentication security scenario" on page 386.

Note: If a user has logged in using web server authentication and has not logged in using ColdFusion application authentication, the `GetAuthUser` tag returns the web server user ID. You could use this feature to combine web server authentication with application authorization based on the user's ID.

About application authentication

With application authentication, you do not rely on the web server to enforce application security. The application performs all user authentication and authorization. The application displays a login page, checks the user's identity and login against its own authorization store, such as an LDAP directory or database, and logs the user into ColdFusion using the `cfloginuser` tag. The application can then use the `IsUserInRole` and `GetAuthUser` functions to check the user's roles or identity for authorization before running a ColdFusion page or specific code on a page. For an example of application authentication use, see "An application authentication security scenario" on page 387.

ColdFusion authentication storage and persistence

How ColdFusion application authentication information is maintained by the browser and ColdFusion, and therefore how long it is available, depends on the following:

- Whether the user's browser enables cookies
- Whether the application supports the Session scope for login storage

Note: For detailed information on Session scope, see "Configuring and using session variables" on page 352. Cookie scope contains the cookies that are sent by the browser; for more information on using cookies, see `cfcookie` in *CFML Reference*.

Authentication and cookies

Because HTTP is connectionless, a login can last beyond a single web page viewing only if the browser provides a unique identifier that software on the server can use to confirm that the current user is authenticated. Normally, this is done by using memory-only cookies that are automatically destroyed when the user closes all open browser windows. The specific cookies and how they are used depend on whether the application supports the Session scope for login storage.

Note: For information on user logins without cookies, see "Using ColdFusion security without cookies" on page 381.

Using the Session scope

If you do the following, ColdFusion maintains login information in the Session scope instead of the Cookie scope:

- Enable the Session scope in the ColdFusion MX Administrator and the Application.cfc initialization code or `cfapplication` tag.
- Specify `loginStorage="Session"` in the Application.cfc initialization code or `cfapplication` tag.

When ColdFusion maintains login information in the Session scope, it stores the authentication details in a Session.cfauthorization variable, and ColdFusion uses the session cookie information to identify the user. Session-based authentication has the following advantages over less persistent login storage:

- After the user logs in, the user ID and password are not passed between the server and the browser.
- The login information and the session share a single time-out. There is no need to manually synchronize sessions and logins.
- If you use server clusters, the Session scope login ID can be available across the cluster. For more information on server clustering, see *Configuring and Administering ColdFusion MX*.

If you do not enable the Session scope, the authentication information is not kept in a persistent scope. Instead, the detailed login information is put in a memory-only cookie (CFAUTHORIZATION_*applicationName*) with a base64-encoded string that contains the user name, password, and application name. The client sends this cookie to the web server each time it makes a page request while the user is logged-in. Use SSL for all page transactions to protect the user ID and password from unauthorized access.

Using ColdFusion security without cookies

You can implement a limited-lifetime form of ColdFusion security if the user's browser does not support cookies. In this case you do not use the `cflogin` tag, only the `cfloginuser` tag. It is the only time you should use the `cfloginuser` tag outside a `cflogin` tag.

Without browser cookies, the effect of the `cfloginuser` tag is limited to a single HTTP request. You must provide your own authentication mechanism and call `cfloginuser` on each page on which you use ColdFusion login identification.

Using ColdFusion security tags and functions

ColdFusion provides the following tags and functions for user security:

Tag or function	Purpose
`cflogin`	A container for user authentication and login code. The body of the tag runs only if the user is not logged in. When using application-based security, you put code in the body of the `cflogin` tag to check the user-provided ID and password against a data source, LDAP directory, or other repository of login identification. The body of the tag includes a `cfloginuser` tag (or a ColdFusion page that contains a `cfloginuser` tag) to establish the authenticated user's identity in ColdFusion.
`cfloginuser`	Identifies (logs in) a user to ColdFusion. Specifies the user's ID, password, and roles. This tag is typically used inside a `cflogin` tag. The `cfloginuser` tag requires three attributes, `name`, `password`, and `roles`, and does not have a body. The `roles` attribute is a comma-delimited list of role identifiers to which the logged-in user belongs. All spaces in the list are treated as part of the role names, so you should not follow commas with spaces. While the user is logged-in to ColdFusion, security functions can access the user ID and role information.

Tag or function	Purpose
cflogout	Logs out the current user. Removes knowledge of the user ID and roles from the server. If you do not use this tag, the user is automatically logged out as described in "Logging out users" on page 385. The cflogout tag does not take any attributes, and does not have a body.
cfNTauthenticate	Authenticates a user name and password against the NT domain on which ColdFusion server is running, and optionally retrieves the user's groups.
cffunction	If you include a roles attribute, the function executes only when there is a logged-in user who belongs to one of the specified roles.
IsUserInRole	Returns True if the current user is a member of the specified role.
GetAuthUser	Returns the ID of the currently logged-in user. This tag first checks for a login made with cfloginuser tag. If none exists, it checks for a web server login (cgi.remote_user.

Using the cflogin tag

The cflogin tag executes only if there is no currently logged-in user. It has the following three optional arguments that control the characteristics of a ColdFusion login:

Attribute	Use
idleTimeout	If no page requests occur during the idleTimeout period, ColdFusion logs out the user. The default is 1800 seconds (30 minutes). This is ignored if login information is stored in the Session scope.
applicationToken	Limits the login validity to a specific application as specified by a ColdFusion page's cfapplication tag. The default value is the current application name.
cookieDomain	Specifies the domain of the cookie used to mark a user as logged-in. You use cookieDomain if you have a clustered environment (for example, x.acme.com, x2.acme.com, and so on). This lets the cookie work for all the computers in the cluster.

Login identification scope and the applicationToken attribute

The login identification created by the cflogin tag is valid only for pages within the directory that contains the page that uses the cflogin tag and any of its subdirectories. Therefore, if a user requests a page in another directory tree, the current login credentials are not valid for accessing those pages. This security limitation lets you use the same user names and passwords for different sections of your application (for example, a UserFunctions tree and a SecurityFunctions tree) and enforce different roles to the users depending on the section.

ColdFusion uses the applicationToken value to generate a unique identifier that enforces this rule. The default applicationToken value is the current application name, as specified by a cfapplication tag or Application.cfc unitization code. In normal use, you do not need to specify an applicationToken value in the cflogin tag.

Specifying the Internet domain

Use the `cookieDomain` attribute to specify the domain of the cookie used to mark a user as logged-in. You use `cookieDomain` if you have a clustered environment (for example, www.acme.com, www2.acme.com, and so on). This lets the cookie work for all computers in the cluster. For example, to ensure that the cookie works for all servers in the acme.com domain, specify `cookieDomain=".acme.com"`. To specify a domain name, start the name with a period.

Caution: Before setting the cookie domain, consider the other applications or servers in the broader domain might have access to the cookie. For example, a clustered payroll application at payroll1.acme.com, payroll2.acme.com, and so on, might reveal sensitive information to the test computer at test.acme.com, if the cookie domain is broadly set to .acme.com.

Getting the user ID and password

The `cflogin` tag has a built-in cflogin structure that contains two variables, cflogin.username and cflogin.password, if the page is executing in response to any of the following:

- Submission of a login form that contains input fields with the names `j_username` and `j_password`.
- A request that uses HTTP Basic authentication and, therefore, includes an Authorization header with the user name and password.
- A message from the Macromedia Flash Remoting gatewayConnection object that has the `setCredentials` method set.
- A request that uses NTLM or Digest authentication. In this case, the user name and password are hashed using a one-way algorithm before they are put in the Authorization header; ColdFusion gets the user name from the web server and sets the cflogin.password value to the empty string.

You use the first three techniques with application authentication, and the last technique with web server authentication. The cflogin structure provides a consistent interface for determining the user's login ID and password, independent of the technique that you use for displaying the login form.

Caution: Login forms send the user name and password without encryption. Basic HTTP authentication sends the user name and password in a base64-encoded string with each request; this format can easily be converted back to plain text. Use these techniques only with https requests, or when you are not concerned about password security.

The following sections describe how you provide login information to your application for authentication

Using a login form to get user information

When you build an application that gets the User ID and password using a login form, the `cflogin` tag checks for the existence of a cflogin structure containing the user's login information. If the structure does not exist, it displays a login form, typically using a `cfinclude` tag on a login page; the following code shows this use.

In the Application.cfc `onRequestStart` method, or a ColdFusion page or CFC method called by the method, you have the following:

```
<cflogin>
  <cfif NOT IsDefined("cflogin")>
    <cfinclude template="loginform.cfm">
  </cfif>
   <cfabort>
   <cfelse>
  <!--- Code to authenticate the user based on the cflogin.user and
    cflogin.password values goes here. --->
  <!--- If User is authenticated, determine any roles and use a line like the
    following to log in the user. --->
    <cfloginuser name="#cflogin.name#" Password = "#cflogin.password#"
      roles="#loginQuery.Roles#">
</cflogin>
```

A simple login form looks like the following:

```
<cfform name="loginform" action="#CGI.script_name#?#CGI.query_string#"
    method="Post">
  <table>
     <tr>
      <td>username:</td>
      <td><cfinput type="text" name="j_username" required="yes"
        message="A username is required"></td>
     </tr>
     <tr>
      <td>password:</td>
      <td><cfinput type="password" name="j_password" required="yes"
        message="A password is required"></td>
     </tr>
  </table>
  <br>
  <input type="submit" value="Log In">
</cfform>
```

Using a browser dialog box to get user information

Application authentication does not require you to use a login form; you can rely on the browser to display its standard login dialog box, instead. To do so, your `cflogin` tag body returns an HTTP status 401 to the browser if the user is not logged in or if the login fails; that is, if it does not have a valid cflogin structure. The browser displays its login dialog box. When the user clicks the login button on the dialog box, the browser returns the login information as an HTTP Authorization header to ColdFusion, which puts the information in the `cflogin` tag's cflogin structure.

This technique has the advantage of simplicity; you do not need a login form and the user gets a familiar-looking login page. You must be careful of security issues, however. The browser sends the user name and password in a base64-encoded string, not just when the user logs in, but with each request. Use SSL (Secure Sockets Layer) for all page transactions to protect the user ID and password from unauthorized access.

Note: You must ensure that your web server is configured correctly to support browser-based login forms for this use. For example, in IIS 5, you must enable anonymous access and might have to disable Basic authentication and Integrated Windows authentication.

The following `cflogin` tag tells the browser to display a login form if the user has not logged in:

```
<cflogin>
  <cfif NOT IsDefined("cflogin")>
    <cfheader statuscode="401">
    <cfheader name="www-Authenticate" value="Basic
      realm=""MM Wizard #args.authtype# Authentication""">
  </cfif>
  <cfabort>
  <cfelse>
    <!--- code to authenticate the user based on the cflogin.user and
      cflogin.password values goes here. --->
</cflogin>
```

Logging in a user using Flash Remoting

If you are developing a Rich Internet Application with Macromedia Flash and Flash Remoting, your ColdFusion application does not need to be coded specially for a Flash login. The Flash Remoting gateway makes the user ID and password available to the `cflogin` tag in the `cflogin` structure.

In your Flash code, you use the ActionScript `SetCredentials` method to send login information to ColdFusion. Your Flash SWF file displays the user ID and password fields, and uses their contents in the `setCredentials` method, as follows:

```
if (inited == null)
{
  inited = true;
  NetServices.setDefaultGatewayUrl("http://localhost/flashservices/gateway");
  gatewayConnection = NetServices.createGatewayConnection();
  gatewayConnection.setCredentials(userID, password);
  myService = gatewayConnection.getService("securityTest.thecfc", this);
}
```

For more information on using Flash Remoting, see *Using Flash Remoting MX*.

Logging out users

After a user logs in, the ColdFusion user authorization and authentication information remains valid until any of the following happens:

- The application uses a `cflogout` tag to log out the user, usually in response to the user clicking a log-out link or button.
- If your application uses the Session scope for login information, the session ends.
- If your application does not use the Session scope for login information, the user does not request a new page for the `cflogin` tag `idleTimeout` period.
- If your application does not use Session scope for login information, or if you use J2EE-based session identification, the user closes all browser windows.

Logging a user out by using the `cflogout` tag does not close the user's session, but if you use session login storage, it does remove the login information (the Session.cfauthorization variable) from the Session scope. For more information on ending sessions, see "Ending a session" on page 356.

Caution: If you use web server-based authentication or any form authentication that uses a Basic HTTP Authorization header, the browser continues to send the authentication information to your application until the user closes the browser, or in some cases, all open browser windows. As a result, after the user logs out and your application uses the `cflogout` tag, until the browser closes, the cflogin structure in the `cflogin` tag will contain the logged-out user's UserID and password. If a user logs out and does not close the browser, another user might access pages with the first user's login.

Security scenarios

The following sections provide two detailed security scenarios. The first scenario uses the web server to perform the authentication against its user and password database. The second scenario uses ColdFusion for all authentication and authorization.

A web server authentication security scenario

An application that uses web server authentication might work as follows. The example in "Web server–based authentication user security example" on page 391 implements this scenario.

1. When the user requests a page from a particular directory on the server for the first time after starting the browser, the web server displays a login page and logs in the user. The web server handles all user authentication.
2. Because the user requested a ColdFusion page, the web server hands the request to ColdFusion.
3. When ColdFusion receives a request for a ColdFusion page, it instantiates the Application.cfc and runs `onRequestStart` method. If you use an Application.cfm page in place of the Application.cfc, it runs the contents of the Application.cfm page before it runs the requested page. The `onRequestStart` method or Application.cfm page contains a `cflogin` tag. ColdFusion executes the `cflogin` tag body if the user is not logged into ColdFusion. The user is logged in if the `cfloginuser` tag has run successfully for this application and the user has not been logged out.
4. Code in the `cflogin` tag body uses the user ID and password from the browser login, contained in the cflogin.name and cflogin.password variables, as follows. (With Digest or NTLM web server authentication, the cflogin.password variable is the empty string.)
    a. It checks the user's name against information it maintains about users and roles. In a simple case, the application might have two roles, one for users and one for administrators. The CFML assigns the Admin role to any user logged on with the user ID *Admin* and assigns the User role to all other users.
    b. It calls the `cfloginuser` tag with the user's ID, password, and roles, to identify the user to ColdFusion.
5. Application.cfc or the Application.cfm page completes processing, and ColdFusion processes the requested application page.

6. The application uses the `IsUserInRole` function to check whether the user belongs to a role before it runs protected code that must be available only to users in that role.
7. The application can use the `GetAuthUser` function to determine the user ID; for example, to display the ID for personalization. It can also use the ID as a database key to get user-specific data.

Caution: If you use web server–based authentication or any form authentication that uses a Basic HTTP Authorization header, the browser continues to send the authentication information to your application until the user closes the browser, or in some cases, all open browser windows. As a result, after the user logs out and your application uses the `cflogout` tag, until the browser closes, the cflogin structure in the `cflogin` tag will contain the logged-out user's UserID and password. If a user logs out and does not close the browser, another user might access pages with the first user's login.

An application authentication security scenario

An application that does its own authentication might work as follows. The example in "Application-based user security example" on page 394 implements this scenario.

1. Whenever ColdFusion receives a request for a ColdFusion page, it instantiates the Application.cfc and runs the `onRequestStart` method. If you use an Application.cfm page in place of Application.cfc, ColdFusion runs the contents of the Application.cfm page before it runs the requested page. The `onRequestStart` method or Application.cfm page contains the `cflogin` tag. ColdFusion executes the `cflogin` tag body if the user is not logged in. A user is logged in if the `cfloginuser` tag has run during the current session and the user had not been logged out by a `cflogout` tag.
2. Code in the `cflogin` tag body checks to see if it has received a user ID and password, normally from a login form.
3. If there is no user ID or password, the code in the `cflogin` tag body displays a login form that asks for the user's ID and password.

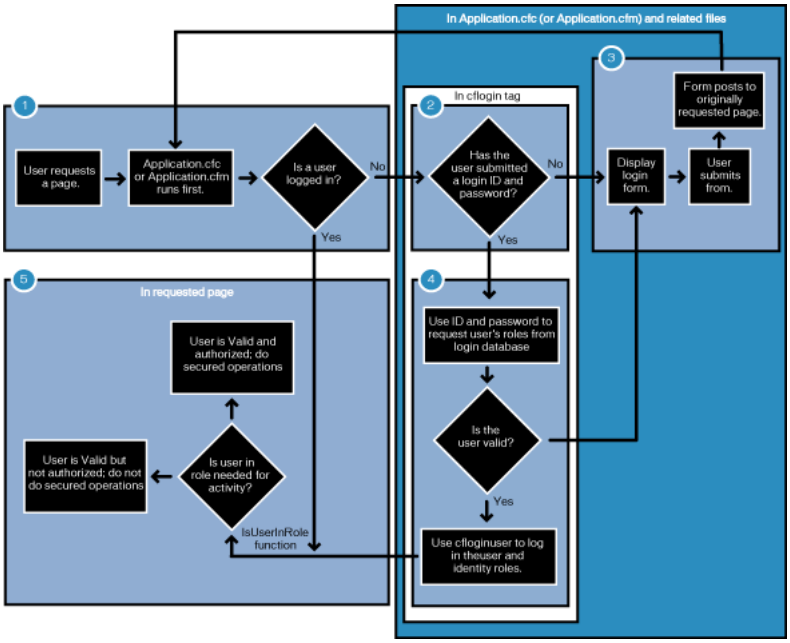
   The form posts the login information back to the originally requested page, and the `cflogin` tag in the `onRequestStart` method or the Application.cfm page runs again. This time, the `cflogin` tag body code checks the user name and password against a database, LDAP directory, or other policy store, to ensure that the user is valid and get the user's roles.
4. If the user name and password are valid, the `cflogin` tag body code calls the `cfloginuser` tag with the user's ID, password, and roles, to identify the user to ColdFusion.
5. When the user is logged in, application pages use the `IsUserInRole` function to check whether the user belongs to a role before they run protected code that must be available only to users in that role.

   The application can use the `GetAuthUser` function to determine the user ID; for example, to display the ID for personalization. It can also use the ID as a database key to get user-specific data.
6. Each application page displays a link to a logout form that uses the `cflogout` tag to log out the user. Typically, the logout link is in a page header that appears in all pages. The logout form can also be in the Application.cfc (for example, in the `onRequestStart` or `onRequestEnd` method) or on the Application.cfm page.

Although this scenario shows one method for implementing user security, it is only an example. For example, your application could require users to log in for only some pages, such as pages in a folder that contains administrative functions. When you design your user security implementation, remember the following:

- Code in the `cflogin` tag body executes only if there is no user logged in.
- With application authentication, you write the code that gets the identification from the user and tests this information against a secure credential store.
- After you have authenticated the user, you use the `cfloginuser` tag to log the user into ColdFusion.

The following figure shows this flow of control. For simplicity, it omits the log-out option.



Implementing user security

The following sections provide several examples of ways to implement security.

Using the Dreamweaver Login Wizard

ColdFusion MX installs a Login Wizard command in the Dreamweaver Commands menu that generates a skeleton set of pages for managing user authentication and authorization.

The wizard asks you to you select how to authenticate the login information. You can select one of the following options:

- Simple You specify a single user ID and password in the wizard. All users must enter this information to log in. You can use this option for testing, or you can use the generated files as a template where you can replace the authentication code with more complex code; for example, to verify the ID and password against a database.
- NT domain You specify an NT domain in the wizard, and the wizard generates code that queries the domain.
- LDAP You specify the LDAP server and port, the username and password required to access the login data, and the distinguished name to use to start the search for the username. The wizard generates the code to query the LDAP server with the user ID and password.

The wizard asks you to select one of the following options for displaying the request for login information:

- Browser Dialog Box
- ColdFusion Login Form

Structure code generated by the Login Wizard

The wizard generates or modifies the following files in the directory or site that you specify:

Application.cfc If this file does not exist, the wizard creates it with a single `onRequestStart` method; it does not specify an application name or any other methods. If the file exists, but does not have an `onRequestStart` method, it adds the method. If Application.cfc and the `onRequestStart` method exist, the wizard inserts the required code at the beginning of the method. The resulting `onRequestStart` method has a `cfinclude` tag that specifies mm_wizard_application_include.cfm; it also has a simple form with a logout button, which will display at the top of each page in the application.

Note: If the wizard creates the Application.cfc file, you should, at least, change the file to specify the application name. For more information on Application.cfc, see Chapter 13, "Designing and Optimizing a ColdFusion Application," on page 275.

mm_wizard_application_include.cfm The Login Wizard uses the information specified in the wizard fields to set several CFC method arguments. It then uses them to invoke the performlogin method of the master login CFC, mm_wizard.authenticate.

mm_wizard_authenticate.cfc This CFC contains all of the user authentication and login logic. The CFC consists of the following methods:

- The ntauth, ldapauth, and simpleauth authentication methods check the user's name and ID against the valid login information, and return information about whether the user is authenticated. For the details of how they authenticate the user and the specific return values, see the methods.
- The performLogin method is the master login method. It contains the `cflogin` tag, which displays the login form and calls the required authentication method. If the authentication method's return argument indicates a valid user, the method logs the user in.

- The logout method logs a user out. If you specified Browser Dialog Box as the login page type, it also calls the closeBrowser method to close the browser window. This behavior is necessary because the browser continues to send the old login credentials after the user logs out, and the `cflogin` tag will automatically use them and log the user in again.
- The closeBrowser method closes the browser window or tells the user to close the browser window to complete the logout, depending on the browser type.

mm_wizard_login.cfm This file contains a ColdFusion MX login form. The wizard generates this file for all options, but does not use it if you specify Browser Dialog login.

index.cfm or mm_wizard_index.cfm The wizard generates an index.cfm page if the directory does not have one; otherwise, creates an mm_wizard_index.cfm page. These pages let you test the generated login code before you implement your application, or without using any of your standard application pages. To test your login, open the index.cfm page in your browser.

Modifying the login code for your application

The Login Wizard creates a basic framework for authenticating a user. You must customize this framework to meet your application's needs. Typical security-related changes include the following:

- Providing user-specific role information in the `cflogin` tag
- Authenticating users against a database

Providing user-specific role information

The Login Wizard sets all users in a single role. In mm_wizard_authenticate.cfc, the performlogin method is hard-coded to set the role to "user." The authentication routines handle roles differently. (For the details, see the mm_wizard_authenticate.cfc code.) If your application uses roles for authorization, you must change the authentication method to get and return valid role information, and change the performlogin method to use the information in the `roles` attribute of its `cfloginuser` tag.

Authenticating users against a database

If you use a database to maintain user IDs and passwords, you can create your login framework by specifying simple authentication, and modify the code to use the database. The following instructions describe a simple way to change the code to use a database. They do not include all the cleanup work (particularly, removing the hard-coded user name and password), that you should do for a well-formatted application.

Replace the following code:

```
<cfif sUserName eq uUserName AND sPassword eq uPassword>
  <cfset retargs.authenticated="YES">
<cfelse>
  <cfset retargs.authenticated="NO">
</cfif>
<cfreturn retargs>
```

With code similar to the following:

```
<cfquery name="loginQuery" dataSource="#Application.DB#" >
  SELECT *
  FROM Users
  WHERE UserName = <cfqueryparam value="#uUserName#" CFSEQLType=
    "CF_SQL_VARCHAR"AND password = <cfqueryparam value="#uPassword#"
    CFSEQLType="CF_SQL_VARCHAR>"
</cfquery>

<cfif loginQuery.recordcount gt 0>
  <cfset retargs.authenticated="YES">
  <cfset retargs.roles=loginQuery.roles>
<cfelse>
  <cfset retargs.authenticated="NO">
</cfif>
<cfreturn retargs>
```

Note: For greater security, consider using a hashed password. Do not store the password directly in the database; instead, use the `hash` function to create a secure password fingerprint, and store it in the database. When the user provides a password, use the `Hash` function on the submitted string and compare it with the value in the database.

Web server–based authentication user security example

The example in this section shows how you might implement user security using web-server–based basic authentication and two roles, user and administrator.

This example has two ColdFusion pages:

- The Application.cfc page logs the user into the ColdFusion security system and assigns the user to specific roles based on the user's ID.

  This page also includes the one-button form and logic for logging out a user, which appears at the top of each page.
- The securitytest.cfm page is a sample application page. It displays the logged-in user's roles.

This simple example does not provide a user log-out interface. You can test the security behavior by adding your own pages to the same directory as the Application.cfc page.

Example: Application.cfc

The Application.cfc page consists of the following:

```
<cfcomponent>
<cfset This.name = "Orders">
<cffunction name="OnRequestStart">
  <cfargument name = "request" required="true"/>
  <cflogin>
    <cfif IsDefined("cflogin")>
      <cfif cflogin.name eq "admin">
        <cfset roles = "user,admin">
      <cfelse>
        <cfset roles = "user">
      </cfif>
      <cfloginuser name = "#cflogin.name#" password = "#cflogin.password#"
```

```
        roles = "#roles#" />
    <cfelse>
      <!--- This should never happen. --->
      <h4>Authentication data is missing.</h4>
      Try to reload the page or contact the site administrator.
    <cfabort>
    </cfif>
  </cflogin>
</cffunction>
</cfcomponent>
```

Reviewing the code

The Application.cfc `onRequestStart` method executes before the code in each ColdFusion page in an application. For more information on the Application.cfc page and when it is executed, see Chapter 13, "Designing and Optimizing a ColdFusion Application," on page 275.

The following table describes the CFML code in Application.cfc and its function:

Code	Description

`<cfcomponent>`
`<cfset This.name = "Orders">`
`<cffunction name="OnRequestStart">`
`<cfargument name = "request"`
`required="true"/>`
	Identifies the application and starts the `onRequestStart` method that runs at the starts of each request. The login information on this page only applies to this application.

`<cflogin>`
`<cfif IsDefined("cflogin")>`
`<cfif cflogin.name eq "admin">`
`<cfset roles = "user,admin">`
`<cfelse>`
`<cfset roles = "user">`
`</cfif>`
	Executes if there is no logged-in user.
	Makes sure the user is correctly logged in by the web server. (Otherwise, there would be no `cflogin` variable.)
	Sets a roles variable based on the user's ID.
	Assigns users named "admin" to the admin role.
	Assigns all other users to the users role.

`<cfloginuser name = "#cflogin.name#"`
`password = "#cflogin.password#"`
`roles = "#roles#" />`
	Logs the user into the ColdFusion security system and specifies the user's password, name, and roles. Gets the password and name directly from the cflogin structure.

`<cfelse>`
`<!--- This should never happen. --->`
`<h4>Authentication data is missing.</h4>`
`Try to reload the page or contact the site administrator.`
`<cfabort>`
	This code should never run, but if the user somehow got to this page without logging in to the web server, this message would display and ColdFusion would stop processing the request.

`</cfif>`
`</cflogin>`
`</cffunction>`
`</cfcomponent>`
	Ends the if/else block.
	Ends the `cflogin` tag body.
	Ends the `onRequestStart` method.
	Ends the Application component.

Example: securitytest.cfm

The securitytest.cfm page shows how any application page can use ColdFusion user authorization features. The web server ensures the existence of an authenticated user, and the Application.cfc page ensures that the user is assigned to roles the page content appears. The securitytest.cfm page uses the `IsUserInRole` and `GetAuthUser` functions to control the information that is displayed.

The securitytest.cfm page consists of the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <title>Basic authentication security test page</title>
</head>

<body>
<cfoutput>
  <h2>Welcome #GetAuthUser()#!</h2>
</cfoutput>

ALL Logged-in Users see this message.<br>
<br>
<cfscript>
  if (IsUserInRole("admin"))
    WriteOutput("Users in the admin role see this message.<br><br>");
  if (IsUserInRole("user"))
    WriteOutput("Everyone in the user role sees this message.<br><br>");
</cfscript>

</body>
</html>
```

Reviewing the code

The following table describes the securitytest.cfm page CFML code and its function:

Code	Description

`<cfoutput>`
`<h2>Welcome #GetAuthUser()#!</h2>`
`</cfoutput>`
	User is already logged in by Application.cfc. Displays a welcome message that includes the user's login ID.

`ALL Logged-in Users see this message. ` ` `	Displays this message in all cases. The page does not display until a user is logged in.
`<cfscript>` `  if (IsUserInRole("admin"))` `  WriteOutput("Users in the admin role` `  see this message.  ");` `  if (IsUserInRole("user"))` `  WriteOutput("Everyone in the user role` `  sees this message.  ");` `</cfscript>`	Tests whether the user belongs to each of the valid roles. If the user is in a role, displays a message with the role name. The user sees one message per role to which the user belongs.

Application-based user security example

The example in this section shows how you might implement user security by authenticating users and then allowing users to see or use only the resources that they are authorized to access.

This example has three ColdFusion pages:

- The Application.cfc page contains the authentication logic that checks whether a user is logged in, requests the login page if the user is not logged in, and authenticates the data from the login page. If the user is authenticated, it logs the user in.

  This page also includes the one-button form and logic for logging out a user, which appears at the top of each page.
- The loginform.cfm page displays the login form. The code on this page could also be included in Application.cfc.
- The securitytest.cfm page is a sample application page. It displays the logged-in user's roles.

You can test the security behavior by adding your own pages to the same directory as the Application.cfc page.

The example gets user information from the LoginInfo table of the cfdocexamples database that is installed with ColdFusion. You can replace this database with any database containing UserID, Password, and Roles fields. The sample database contains the following data:

UserID	Password	Roles
BobZ	Ads10	Employee,Sales
JaniceF	Qwer12	Contractor,Documentation
RandalQ	ImMe	Employee,Human Resources,Manager

Because spaces are meaningful in roles strings, you should not follow the comma separators in the Roles fields with spaces.

Example: Application.cfc

The Application.cfc page consists of the following:

```
<cfcomponent>
<cfset This.name = "Orders">
<cfset This.Sessionmanagement="True">
<cfset This.loginstorage="session">

<cffunction name="OnRequestStart">
  <cfargument name = "request" required="true"/>
  <cfif IsDefined("Form.logout")>
    <cflogout>
  </cfif>

  <cflogin>
    <cfif NOT IsDefined("cflogin")>
      <cfinclude template="loginform.cfm">
      <cfabort>
    <cfelse>
```

```
        <cfif cflogin.name IS "" OR cflogin.password IS "">
          <cfoutput>
            <h2>You must enter text in both the User Name and Password fields.
            </h2>
          </cfoutput>
          <cfinclude template="loginform.cfm">
          <cfabort>
        <cfelse>
          <cfquery name="loginQuery" dataSource="cfdocexamples">
          SELECT UserID, Roles
          FROM LoginInfo
          WHERE
            UserID = '#cflogin.name#'
            AND Password = '#cflogin.password#'
          </cfquery>
          <cfif loginQuery.Roles NEQ "">
            <cfloginuser name="#cflogin.name#" Password = "#cflogin.password#"
              roles="#loginQuery.Roles#">
          <cfelse>
            <cfoutput>
              <H2>Your login information is not valid.<br>
              Please Try again</H2>
            </cfoutput>
            <cfinclude template="loginform.cfm">
            <cfabort>
          </cfif>
        </cfif>
      </cfif>
    </cflogin>

    <cfif GetAuthUser() NEQ "">
      <cfoutput>
        <form action="securitytest.cfm" method="Post">
          <input type="submit" Name="Logout" value="Logout">
        </form>
      </cfoutput>
    </cfif>

</cffunction>
</cfcomponent>
```

Reviewing the code

The Application.cfc page executes before the code in each ColdFusion page in an application. For more information on the Application.cfc page and when it is executed, see Chapter 13, “Designing and Optimizing a ColdFusion Application,” on page 275.

The following table describes the CFML code in Application.cfc and its function:

Code	Description

`<cfcomponent>`
`<cfset This.name = "Orders">`
`<cfset This.Sessionmanagement="True">`
`<cffunction name="OnRequestStart">`
`<cfargument name = "request"`
`required="true"/>`
	Identifies the application, enables session management, and enables storing login information in the Session scope.
	Begins the definition of the `onRequestStart` method that runs at the starts of each request.

`<cfif IsDefined("Form.logout")>`
`<cflogout>`
`</cfif>`
	If the user just submitted the logout form, logs out the user. The following `cflogin` tag runs as a result.

`<cflogin>`
`<cfif NOT IsDefined("cflogin")>`
`<cfinclude template="loginform.cfm">`
`<cfabort>`
	Executes if there is no logged-in user.
	Tests to see if the user has submitted a login form. If not, uses `cfinclude` to display the form. The built-in `cflogin` variable exists and contains the user name and password only if the login form used `j_username` and `j_password` for the input fields.
	The `cfabort` tag prevents processing of any code that follows on this page.

`<cfelse>`
`<cfif cflogin.name IS "" OR`
`cflogin.password IS "">`
`<cfoutput>`
`<H2>You must enter text in both the`
`User Name and Password fields</H2>`
`</cfoutput>`
`<cfinclude template="loginform.cfm">`
`<cfabort>`
	Executes if the user submitted a login form.
	Tests to make sure that both name and password have data. If either variable is empty, displays a message, followed by the login form.
	The `cfabort` tag prevents processing of any code that follows on this page.

`<cfelse>`
`<cfquery name="loginQuery"`
`dataSource="cfdocexamples">`
`SELECT UserID, Roles`
`FROM LoginInfo`
`WHERE`
`UserID = '#cflogin.name#'`
`AND Password = '#cflogin.password#'`
`</cfquery>`
	Executes if the user submitted a login form and both fields contain data.
	Uses the cflogin structure's `name` and `password` entries to find the user record in the database and get the user's roles.

`<cfif loginQuery.Roles NEQ "">`
`<cfloginuser name="#cflogin.name#"`
`Password = "#cflogin.password#"`
`roles="#loginQuery.Roles#">`
	If the query returns data in the Roles field, logs in the user using the user's name and password and the Roles field from the database. In this application, every user must be in some role.

Code	Description
<cfelse> <cfoutput> <H2>Your login information is not valid. Please Try again</H2> </cfoutput> <cfinclude template="loginform.cfm"> <cfabort>	Executes if the query did not return a role. If the database is valid, this means there was no entry matching the user ID and password. Displays a message, followed by the login form. The cfabort tag prevents processing of any code that follows on this page.
</cfif> </cfif> </cfif> </cflogin>	Ends the loginquery.Roles test code. Ends the form entry empty value test. Ends the form entry existence test. Ends the cflogin tag body.
<cfif GetAuthUser() NEQ ""> <cfoutput> <form action="MyApp/index.cfm" method="Post"> <input type="submit" Name="Logout" value="Logout"> </form> </cfoutput> </cfif>	If a user is logged-in, displays the Logout button. If the user clicks the button, posts the form to the application's (theoretical) entry page, index.cfm. Application.cfc then logs out the user and displays the login form. If the user logs in again, ColdFusion displays index.cfm.
</cffunction> </cfcomponent>	Ends the onRequestStart method Ends the Application component.

Example: loginform.cfm

The loginform.cfm page consists of the following:

```
<H2>Please Log In</H2>
<cfoutput>
  <form action="#CGI.script_name#?#CGI.query_string#" method="Post">
    <table>
      <tr>
        <td>username:</td>
        <td><input type="text" name="j_username"></td>
      </tr>
      <tr>
        <td>password:</td>
        <td><input type="password" name="j_password"></td>
      </tr>
    </table>
    <br>
    <input type="submit" value="Log In">
  </form>
</cfoutput>
```

Reviewing the code

The following table describes the loginform.cfm page CFML code and its function:

Code	Description

<H2>Please Log In</H2>
<cfoutput>
  <form
  action="#CGI.script_name#?#CGI.
  query_string#"
  method="Post">
  <table>
  <tr>
  <td>username:</td>
  <td><input type="text"
  name="j_username"></td>
  </tr>
  <tr>
  <td>password:</td>
  <td><input type="password"
  name="j_password"></td>
  </tr>
  </table>
 	
  <input type="submit" value="Login">
  </form>
</cfoutput>
	Displays the login form.
	
	Constructs the form `action` attribute from CGI variables, with a ? character preceding the query string variable. This technique works because loginform.cfm is accessed by a `cfinclude` tag on Application.cfc, so the CGI variables are those for the originally requested page.
	
	The form requests a user ID and password and posts the user's input to the page specified by the newurl variable.
	Uses the field names j_username and j_password. ColdFusion automatically puts form fields with these values in the cflogin.name and cflogin.password variables inside the `cflogin` tag.

Example: securitytest.cfm

The securitytest.cfm page shows how any application page can use ColdFusion user authorization features. Application.cfc ensures the existence of an authenticated user before the page content appears. The securitytest.cfm page uses the `IsUserInRole` and `GetAuthUser` functions to control the information that is displayed.

The securitytest.cfm page consists of the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <title>Security test page</title>
</head>

<body>
<cfoutput>
  <h2>Welcome #GetAuthUser()#!</h2>
</cfoutput>

ALL Logged-in Users see this message.<br>
<br>
<cfscript>
  if (IsUserInRole("Human Resources"))
    WriteOutput("Human Resources members see this message.<br><br>");
  if (IsUserInRole("Documentation"))
    WriteOutput("Documentation members see this message.<br><br>");
  if (IsUserInRole("Sales"))
    WriteOutput("Sales members see this message.<br><br>");
```

```
  if (IsUserInRole("Manager"))
    WriteOutput("Managers see this message.<br><br>");
  if (IsUserInRole("Employee"))
    WriteOutput("Employees see this message.<br><br>");
  if (IsUserInRole("Contractor"))
    WriteOutput("Contractors see this message.<br><br>");
</cfscript>

</body>
</html>
```

Reviewing the code

The following table describes the securitytest.cfm page CFML code and its function:

Code	Description

`<cfoutput>`
`<h2>Welcome #GetAuthUser()#!</h2>`
`</cfoutput>`
	Displays a welcome message that includes the user's login ID.

`ALL Logged-in Users see this message. ` ` `	Displays this message in all cases. The page does not display until a user is logged in.
`<cfscript>` `  if (IsUserInRole("Human Resources"))` `  WriteOutput("Human Resources members see this message.  ");` `  if (IsUserInRole("Documentation"))` `  WriteOutput("Documentation members see this message.  ");` `  if (IsUserInRole("Sales"))` `  WriteOutput("Sales members see this message.  ");` `  if (IsUserInRole("Manager"))` `  WriteOutput("Managers see this message.  ");` `  if (IsUserInRole("Employee"))` `  WriteOutput("Employees see this message.  ");` `  if (IsUserInRole("Contractor"))` `  WriteOutput("Contractors see this message.  ");` `</cfscript>`	Tests whether the user belongs to each of the valid roles. If the user is in a role, displays a message with the role name. The user sees one message per role to which he or she belongs.

Using an LDAP directory for security information

LDAP directories are often used to store security information. The following example of a `cflogin` tag checks an LDAP directory to authenticate the user and retrieve the user's roles.

For more information on using LDAP directories with ColdFusion, see Chapter 23, "Managing LDAP Directories," on page 515.

```
<cfapplication name="Orders" sessionmanagement="Yes" loginstorage="Session">
<cflogin>
  <cfif isDefined("cflogin")>
    <!--- setting basic attributes --->
```

```
    <cfset LDAP_root = "o=mycompany.com">
    <cfset LDAP_server = "ldap.mycompany.com">
    <cfset LDAP_port = "389">

    <!--- Create the prefix and suffix parts of the user's DN. --->
    <cfset userPrefix = "cn=">
    <cfset userSuffix = ",ou=Users,o=mycompany.com">

    <!--- Concatenate the user's DN and use it to authenticate. --->
    <cfset LDAP_username = userPrefix&cflogin.name&userSuffix>

    <!--- This filter will look for groups for containing the user's ID. --->
    <cfset userfilter =
      "(&(objectClass=groupOfUniqueNames)(uniqueMember=#LDAP_username#))">

    <!--- Search for groups containing the user's dn.
      The groups represent the user's roles.
      NOTE: Your LDAP permissions must allow authenticated users to search.
      groups. --->
    <cftry>
      <cfldap action="QUERY"
        name="auth"
        attributes="cn"
        referral="yes"
        start="#LDAP_root#"
        scope="SUBTREE"
        server="#LDAP_server#"
        port="#LDAP_port#"
        filter="#userfilter#"
        username="#LDAP_username#"
        password="#cflogin.password#"
      >
      <cfcatch type="any">
        <cfif FindNoCase("Invalid credentials", cfcatch.detail)>
          <cfoutput>
            <script>alert("User ID or Password invalid for user:
              #cflogin.name#")</script>
          </cfoutput>
          <cfabort>
        <cfelse>
          <cfoutput>
            <script>alert("Unknown error for user: #cflogin.name#
              #cfcatch.detail#")</script>
          </cfoutput>
          <cfabort>
        </cfif>
      </cfcatch>
    </cftry>

<!--- If the LDAP query returned a record, the user is valid. --->
    <cfif auth.recordcount>
      <cfloginuser name="#cflogin.name#" password="#cflogin.password#"
        roles="#valueList(auth.cn)#">
    </cfif>
```

```
  </cfif>
</cflogin>
```

Reviewing the code

The following table describes the code and its function. Comments and some tab characters have been removed for brevity.

Code	Description

`<cflogin>`
`  <cfif isDefined("cflogin")>`
`  <!--- setting basic attributes --->`
`  <cfset LDAP_root = "o=mycompany.com">`
`  <cfset LDAP_server =`
`  "ldap.mycompany.com">`
`  <cfset LDAP_port = "389">`
	Starts the `cflogin` tag body. Sets several variables to the values used as attributes in the `cfldap` tag.

`  <cfset userPrefix = "cn=">`
`  <cfset userSuffix =`
`  ",ou=Users,o=mycompany.com">`
	Sets prefix and suffix values used to create a distinquished name (dn) for binding to the LDAP server.

`  <cfset LDAP_username =` `  userPrefix&cflogin.name&userSuffix>`	Creates the user's bind dn by concatenating the prefix and suffix with cflogin.name. This variable is used for authenticating the user to the LDAP server.

`  <cfset userfilter =`
`  "(&(objectClass=groupOfUniqueNames)`
`  (uniqueMember=#LDAP_username#))">`
	Sets the filter used to search the directory and retrieve the user's group memberships. The group membership represents the user's roles within the organization.

`<cftry>`
`  <cfldap action="QUERY"`
`  name="auth"`
`  attributes="cn"`
`  referral="yes"`
`  start="#LDAP_root#"`
`  scope="SUBTREE"`
`  server="#LDAP_server#"`
`  port="#LDAP_port#"`
`  filter="#userfilter#"`
`  username="#LDAP_username#"`
`  password="#cflogin.password#"`
`  >`
	In a `cftry` block, uses the user's concatenated dn to authenticate to the LDAP server and retrieve the common name (cn) attribute for groups to which the user is a member. If the authentication fails the LDAP server returns an error.
	
	Note: The LDAP permissions must allow an authenticated user to read and search groups in order for the query to return results.

Code	Description
`<cfcatch type="any">`	Catches any exceptions.
`<cfif FindNoCase("Invalid credentials", cfcatch.detail)>`	Tests to see if the error information includes the string "invalid credentials", which indicates that either the dn or password is invalid. If so, displays a dialog box with an error message indicating the problem.
`<cfoutput>`	
`<script>alert("User ID or Password invalid for user: #cflogin.name#")</script>`	
`</cfoutput>`	
`<cfabort>`	
`<cfelse>`	Otherwise, displays a general error message.
`<cfoutput>`	
`<script>alert("Unknown error for user: #cflogin.name# #cfcatch.detail#")</script>`	
`</cfoutput>`	
`<cfabort>`	If an error is caught, the `cfabort` tag ends processing of the request after displaying the error description.
`</cfif>`	
`</cfcatch>`	End of `cfcatch` and `cftry` blocks.
`</cftry>`	
`<cfif auth.recordcount>`	If the authorization query returns a valid record, logs in the user. Uses the `valueList` function to create a comma-separated list of the users retrieved group memberships, and passes them in the `cfloginuser roles` attribute.
`<cfloginuser name="#cflogin.name#" password="#cflogin.password#" roles="#valueList(auth.cn)#">`	
`</cfif>`	Ends the initial `isDefined("cflogin") cfif` block .
`</cfif>`	
`</cflogin>`	Ends the `cflogin` tag body

CHAPTER 17
Developing Globalized Applications

Macromedia ColdFusion MX lets you develop dynamic applications for the Internet. Many ColdFusion applications are accessed by users from different countries and geographical areas. One design detail that you must consider is the globalization of your application so that you can best serve customers in different areas.

This chapter contains information that you can use to develop applications that can be accessible to many different users.

Contents

Introduction to globalization 403

About character encodings 406

Locales 408

Processing a request in ColdFusion 410

Tags and functions for globalizing 413

Handling data in ColdFusion MX 416

Introduction to globalization

Globalization lets you create applications for all of your customers in all the languages that you support. In some cases, globalization can let you accept data input using a different character set than the one you used to implement your application. For example, you can create a website in English that lets customers submit form data in Japanese. Or, you can allow a request URL to contain parameter values entered in Korean.

Your application also can process data containing numeric values, dates, currencies, and times. Each of these types of data can be formatted differently for different countries and regions.

You can also develop applications in languages other than English. For example, you can develop your application in Japanese so that the default character encoding is Shift-JIS, your ColdFusion pages contain Japanese characters, and your interface displays in Japanese.

Globalizing your application requires that you perform one or more of the following actions:

- Accept input in more than one language.
- Process dates, times, currencies, and numbers formatted for multiple locales.
- Process data from a form, database, HTTP connection, e-mail message, or other input formatted in multiple character sets.
- Create ColdFusion pages containing text in languages other than English.

Defining globalization

You will probably find several different definitions for globalization. For this chapter, globalization is defined as an architectural process where you put as much application functionality as possible into a foundation that can be shared among multiple languages.

Globalization is composed of the following two parts:

Internationalization Developing language-neutral application functionality that can recognize, process, and respond to data regardless of its representation. That is, whatever the application can do in one language, it can also do in another. For example, think of copying and pasting text. A copy and paste operation should not be concerned with the language of the text it operates on. For a ColdFusion application, you might have processing logic that performs numeric calculations, queries a database, or performs other operations, independent of language.

Localization Taking shared, language-neutral functionality, and applying a locale-specific interface to it. Sometimes this interface is referred to as a *skin*. For example, you can develop a set of menus, buttons, and dialog boxes for a specific language, such as Japanese, that represents the language-specific interface. You then combine this interface with the language-neutral functionality of the underlying application. As part of localization, you create the functionality to handle input from customers in a language-specific manner and respond with appropriate responses for that language.

Importance of globalization in ColdFusion applications

The Internet has no country boundaries. Customers can access websites from anywhere in the world, at any time, or on any date. Unless you want to lock your customers into using a single language, such as English, to access your site, you should consider globalization issues.

One reason to globalize your applications is to avoid errors and confusion for your customers. For example, a date in the form 1/2/2003 is interpreted as January 2, 2003 in the United States, but as February 1, 2003 in European countries.

Another reason to globalize your applications is to display currencies in the correct format. Think of how your customers would feel when they find out the correct price for an item is 15,000 American dollars, not 15,000 Mexican pesos (about 1600 American dollars).

Your website can also accept customer feedback or some other form of text input. You might want to support that feedback in multiple languages using a variety of character sets.

How ColdFusion MX supports globalization

ColdFusion MX is implemented in Java. As a Java application, ColdFusion MX uses Java globalization features. For example, ColdFusion MX stores all strings internally using the Unicode character set. Because it uses Unicode, ColdFusion can represent any text data from any language.

In addition, ColdFusion MX includes many tags and functions designed to support globalizing your applications. You can use these tags and functions to set locales, convert date and currency formats, control the output encoding of ColdFusion pages, and perform other actions.

Character sets, character encodings, and locales

When you discuss globalization issues, two topics that you must consider are the character sets or character encodings recognized by the application and the locales for which the application must format data.

A *character set* is a collection of characters. For example, the Latin alphabet is the character set that you use to write English, and it includes all of the lower- and upper-case letters from A to Z. A character set for French includes the character set used by English, plus special characters such as "é," "à," and "ç."

The Japanese language uses three alphabets: Hiragana, Katakana, and Kanji. Hiragana and Katakana are phonetic alphabets that each contain 46 characters plus two accents. Kanji contains Chinese ideographs adapted to the Japanese language. The Japanese language uses a much larger character set than English because Japanese supports more than 10,000 different characters.

In order for a ColdFusion application to process text, the application must recognize the character set used by the text. The *character encoding* maps between a character set definition and the digital codes used to represent the data.

In general use, the terms character set (or charset) and character encoding are often used interchangeably, and most often a specific character encoding encodes one character set. However, this is not always true; for example, there are multiple encodings of the Unicode character set. For more information on character encodings, see "About character encodings" on page 406.

Note: ColdFusion uses the term *charset* to indicate character encoding in some attribute names, structure field keys, and function parameter names.

A *locale* identifies the exact language and cultural settings for a user. The locale controls how dates and currencies are formatted, how to display time, and how to display numeric data. For example, the locale English (US) determines that a currency value displays as:

$100,000.00

while a locale of Portuguese (Brazilian) displays the currency as:

R$ 100.000

In order to correctly display date, time, currency, and numeric data to your customers, you must know the customer's locale. For more information on locales, see "Locales" on page 408.

About character encodings

A *character encoding* maps each character in a character set to a numeric value that can be represented by a computer. These numbers can be represented by a single byte or multiple bytes. For example, the ASCII encoding uses seven bits to represent the Latin alphabet, punctuation, and control characters.

You use Japanese encodings, such as Shift-JIS, EUC-JP, and ISO-2022-JP, to represent Japanese text. These encodings can vary slightly, but they include a common set of approximately 10,000 characters used in Japanese.

The following terms apply to character encodings:

SBCS Single-byte character set; a character set encoded in one byte per character, such as ASCII or ISO 8859-1.

DBCS Double-byte character set; a method of encoding a character set in no more than two bytes, such as Shift-JIS. Many character encoding schemes that are referred to as double-byte, including Shift-JIS, allow mixing of single-byte and double-byte encoded characters. Others, such as UCS-2, use two bytes for all characters.

MBCS Multiple-byte character set; a character set encoded with a variable number of bytes per character, such as UTF-8.

The following table lists some common character encodings; however, there are many additional character encodings that browsers and web servers support:

Encoding	Type	Description
ASCII	SBCS	7-bit encoding used by English and Indonesian Bahasa languages
Latin-1 (ISO 8859-1)	SBCS	8-bit encoding used for many Western European languages
Shift_JIS	DBCS	16-bit Japanese encoding ***Note:*** You must use an underscore character (_), not a hyphen (-) in the name in CFML attributes.
EUC-KR	DBCS	16-bit Korean encoding
UCS-2	DBCS	Two-byte Unicode encoding
UTF-8	MBCS	Multibyte Unicode encoding. ASCII is 7-bit; non-ASCII characters used in European and many Middle Eastern languages are two-byte; and most Asian characters are three-byte

The World Wide Web Consortium maintains a list of all character encodings supported by the Internet. You can find this information at www.w3.org/International/O-charset.html.

Computers often must convert between character encodings. In particular, the character encodings most commonly used on the Internet are not used by Java or Windows. Character sets used on the Internet are typically single-byte or multiple-byte (including DBCS character sets that allow single-byte characters). These character sets are most efficient for transmitting data, because each character takes up the minimum necessary number of bytes. Currently, Latin characters are most frequently used on the web, and most character encodings used on the web represent those characters in a single byte.

Computers, however, process data most efficiently if each character occupies the same number of bytes. Therefore, Windows and Java both use double-byte encoding for internal processing.

The Java Unicode character encoding

ColdFusion MX uses the Java Unicode Standard for representing character data internally. This standard corresponds to UCS-2 encoding of the Unicode character set. The Unicode character set can represent many languages, including all major European and Asian character sets. Therefore, ColdFusion MX can receive, store, process, and present text from all languages supported by Unicode.

The Java Virtual Machine (JVM) that is used to processes ColdFusion pages converts between the character encoding used on a ColdFusion page or other source of information to UCS-2. The page or data encodings that ColdFusion supports depend on the specific JVM, but include most encodings used on the web. Similarly, the JVM converts between its internal UCS-2 representation and the character encoding used to send the response to the client.

By default, ColdFusion MX uses UTF-8 to represent text data sent to a browser. UTF-8 represents the Unicode character set using a variable-length encoding. ASCII characters are sent using a single byte. Most European and Middle Eastern characters are sent as two bytes, and Japanese, Korean, and Chinese characters are sent as three bytes. One advantage of UTF-8 is that it sends ASCII character set data in a form that can be recognized by systems designed to process only single-byte ASCII characters, while it is flexible enough to handle multiple-byte character representations.

While the default format of text data returned by ColdFusion is UTF-8, you can have ColdFusion return a page to any character set supported by Java. For example, you can return text using the Japanese language Shift-JIS character set. Similarly, ColdFusion can handle data that is in many different character sets. For more information, see “Determining the page encoding of server output” on page 412.

Character encoding conversion issues

Because different character encodings support different character sets, you can encounter errors if your application gets text in one encoding and presents it in another encoding. For example, the Windows Latin-1 character encoding, Windows-1252, includes characters with hexadecimal representations in the range 80-9F, while ISO 8859-1 does not include characters in that range. As a result, under the following circumstances, characters in the range 80-9F, such as the euro symbol (Ä), are not displayed properly:

- A file encoded in Windows-1252 includes characters in the range 80-9F.
- ColdFusion reads the file, specifying the Windows-1252 encoding in the `cffile` tag.
- ColdFusion displays the file contents, specifying ISO-8859 in the `cfcontent` tag.

Similar issues can arise if you convert between other character encodings; for example, if you read files encoded in the Japanese Windows default encoding and display them using Shift-JIS. To prevent these problems, ensure that the display encoding is the same as the input encoding.

Locales

A *locale* identifies the exact language and cultural settings to use for a user. The locale controls how to format the following:

- Dates
- Times
- Numbers
- Currency amounts

ColdFusion MX supports all locales supported by the JVM that it uses.

Note: Current JVM versions (through 1.4.2) do not support localized numbers such as Arabic-hindic numbers used in Arabic locales or hindic digits used in Hindi locales. ColdFusion uses Arabic numbers in all locales.

Locale names

ColdFusion MX 7 supports two formats for specifying locale names: the standard Java locale names and the ColdFusion naming convention that was required through ColdFusion MX 6.1.

- You can specify all locales using a name consisting of the following:
  - Two lowercase letters to identify the language; for example, en for English, or zh for Chinese.
  - Optionally, an underscore and a two uppercase letters to identify the regional variant of the language; for example, US for the United States, or HK for Hong Kong.

  For example, en_US represents United States English and es_MX represents Mexican Spanish. For a list of the Java locale identifiers supported in the Sun 1.4.2 JVM and their meanings, see http://java.sun.com/j2se/1.4.2/docs/guide/intl/locale.doc.html.
- Prior to ColdFusion MX 7, ColdFusion supported a limited set of locales, and used identifiers that consisted of the name of the language, followed, for most languages, by a regional identifier in parentheses, such as English (US) or German (Standard). ColdFusion MX continues to support these names; for a list, see `SetLocale` in *CFML Reference*.

The `Server.coldfusion.supportedlocales` variable is a comma-delimited list of the locale names that you can specify.

ColdFusion also includes a `GetLocaleDisplayName` function that returns a locale name in a format that is meaningful to users. It lets you display the locale using words in the user's language; for example, français (France).

Determining the locale

ColdFusion MX 7 determines the locale value as follows:

- By default, ColdFusion uses the JVM locale, and the default JVM locale is the operating system locale. You can set the JVM locale value explicitly in ColdFusion MX 7 in the JVM Arguments field on the Java and JVM Settings page in the ColdFusion MX Administrator; for example:

  ```
  -Duser.language=de -Duser.region=DE.
  ```

- A locale set using the `SetLocale` function persists for the current request or until it is reset by another `SetLocale` function in the request.
- If a request has multiple `SetLocale` functions, the current locale setting affects how locale-sensitive ColdFusion tags and functions (such as the functions that start with **LS**) format data. The last `SetLocale` function that ColdFusion processes before sending a response to the requestor (typically the client browser) determines the value of the response `Content-Language` HTTP header. The browser that requested the page displays the response according to the rules for the language specified by the `Content-Language` header.
- ColdFusion ignores any `SetLocale` functions that follow a `cfflush` tag.

Using the locale

The `SetLocale` function determines the default formats that ColdFusion uses to output date, time, number, and currency values. You use the `GetLocale` function to determine the current locale setting of ColdFusion, or you can use the `GetLocaleDisplayName` function to get the locale name in a format that is meaningful to users. If you have not made a call to `SetLocale`, `GetLocale` returns the locale of the JVM.

The current locale has two effects:

- When ColdFusion formats date, time, currency, or numeric output, it determines how to format the output. You can change the locale multiple times on a ColdFusion page to format information according to different locale conventions. This enables you to output a page that properly formats different currency values, for example.
- When ColdFusion returns a page to the client, it includes the HTTP `Content-Language` header. ColdFusion uses the last locale setting on the page for this information.

Note: In earlier versions of ColdFusion, the default locale was always English, not the operating system's locale. For the Japanese version of ColdFusion, the default was Japanese.

The following example uses the `LSCurrencyFormat` function to output the value 100,000 in monetary units for all the ColdFusion-supported locales. You can run this code to see how the locale affects the data returned to a browser.

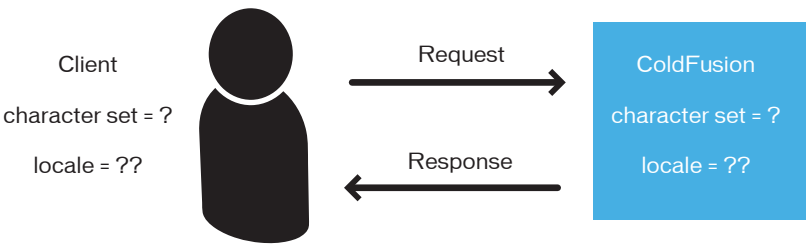
```
<p>LSCurrencyFormat returns a currency value using the locale convention.
<!--- loop through list of locales; show currency values for 100,000 units --->
<cfloop LIST = "#Server.Coldfusion.SupportedLocales#"
  index = "locale" delimiters = ",">
<cfset oldlocale = SetLocale(locale)>
  <cfoutput><p><b><I>#locale#</I></b><br>
    Local: #LSCurrencyFormat(100000, "local")#<br>
    International: #LSCurrencyFormat(100000, "international")#<br>
    None: #LSCurrencyFormat(100000, "none")#<br>
    <hr noshade>
  </cfoutput>
</cfloop>
```

This example uses the ColdFusion variable `Server.Coldfusion.SupportedLocales`, which contains a list of all supported ColdFusion locales.

Processing a request in ColdFusion

When ColdFusion receives an HTTP request for a ColdFusion page, ColdFusion resolves the request URL to a physical file path and reads the file contents to parse it. A ColdFusion page can be encoded in any character encoding supported by the JVM used by ColdFusion, but might need to be specified so that ColdFusion can identify it.

The following figure shows an example of a client making a request to ColdFusion:



The content of the ColdFusion page on the server can be static data (typically HTML and plain text not processed by ColdFusion), and dynamic content written in CFML. Static content is written directly to the response to the browser, and dynamic content is processed by ColdFusion.

The default language of a website might be different from that of the person connecting to it. For example, you could connect to an English website from a French computer. When ColdFusion generates a response, the response must be formatted in the way expected by the customer. This includes both the character set of the response and the locale.

The following sections describe how ColdFusion determines the character set of the files that it processes, and how it determines the character set and locale of its response to the client.

Determining the character encoding of a ColdFusion page

When a request for a ColdFusion page occurs, ColdFusion opens the page, processes the content, and returns the results back to the browser of the requestor. In order to process the ColdFusion page, though, ColdFusion has to interpret the page content.

One piece of information used by ColdFusion is the Byte Order Mark (BOM) in a ColdFusion page. The BOM is a special character at the beginning of a text stream that specifies the order of bytes in multibyte characters used by the page. The following table lists the common BOM values:

Encoding	BOM signature
UTF-8	EF BB BF
UTF-16 Big Endian	FE FF
UTF-16 Little Endian	FF FE

To insert a BOM character in a CFML page easily, your editor must support BOM characters. Many web page development tools support insertion of these characters, including Macromedia Dreamweaver MX, which automatically sets the BOM based on the Page Properties Document Encoding selection.

If your page does not contain a BOM, you can use the `cfprocessingdirective` tag to set the character encoding of the page. If you insert the `cfprocessingdirective` tag on a page that has a BOM, the information specified by the `cfprocessingdirective` tag must be the same as for the BOM; otherwise, ColdFusion issues an error.

The following procedure describes how ColdFusion recognizes the encoding format of a ColdFusion page.

To determine the page encoding (performed by ColdFusion):

1. Use the BOM, if specified on the page.

   Macromedia recommends that you use BOM characters in your files.
2. Use the `pageEncoding` attribute of the `cfprocessingdirective` tag, if specified. For detailed information on how to use this attribute, see the `cfprocessingdirective` tag in *CFML Reference*.
3. Default to the JVM default file character encoding. By default, this is the operating system default character encoding. To specify the JVM default file character encoding, use the `-Dfile.encoding= switch` in the JVM Arguments field of the ColdFusion MX Administrator Java and JVM Settings page.

Determining the page encoding of server output

Before ColdFusion can return a response to the client, it must determine the encoding to use for the data in the response. By default, ColdFusion returns character data using the Unicode UTF-8 format.

ColdFusion pages (.cfm pages) default to using the Unicode UTF-8 format for the response, even if you include the HTML `meta` tag in the page. Therefore, the following example will **not** modify the character set of the response:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
           "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type"
  content="text/html;
  charset="Shift_JIS">
</head>
...
```

In this example, the response will still use the UTF-8 character set. Use the `cfcontent` tag to set the output character set.

However, within a ColdFusion page you can use the `cfcontent` tag to override the default character encoding of the response. Use the `type` attribute of the `cfcontent` tag to specify the MIME type of the page output, including the character set, as follows:

```
<cfcontent type="text/html charset=EUC-JP">
```

Note: ColdFusion MX also provides attributes that let you specify the encoding of specific elements, such as HTTP requests, request headers, files, and mail messages. For more information, see "Tags and functions for controlling character encoding" on page 413 and "Handling data in ColdFusion MX" on page 416.

The rest of this chapter describes ColdFusion tags and functions that you use for globalization, and discusses specific globalization issues.

Tags and functions for globalizing

ColdFusion MX supplies many tags and functions that you can use to develop globalized applications.

Tags and functions for controlling character encoding

The following tags and functions let you specify the character encoding of text that ColdFusion generates and interprets:

Tag or function	Attribute or parameter	Use
`cfprocessingdirective`	pageencoding	Specifies the encoding of a ColdFusion page so ColdFusion can parse it. For more information, see "Determining the character encoding of a ColdFusion page" on page 411.
`cfcontent`	type	Specifies the encoding in which to return the results to the client browser. For more information, see "Determining the page encoding of server output" on page 412.
`cffile`	charset	Specifies how to encode data written to a file, or the encoding of a file being read. For more information, see "File data" on page 419.
`cfheader`	charset	Specifies the character encoding in which to encode the HTTP header value.
`cfhttp`	charset	Specifies the character encoding of the HTTP request.
`cfhttpparam`	mimeType	Specifies the MIME media type of a file; can positionally include the file's character encoding.
`cfmail`	charset	Specifies the character encoding of the mail message, including the headers.
`cfmailpart`	charset	Specifies the character encoding of one part of a multipart mail message.
`cfprocessingdirective`	pageEncoding	Identifies the character encoding of the contents of a page to be processed by ColdFusion MX..
`CharsetDecode`	encoding	Converts a string in the specified encoding to a binary object.
`CharsetEncode`	encoding	Converts a binary object to a string in the specified encoding.
`GetEncoding`		Returns the character encoding of text in the Form or URL scope.
`SetEncoding`	charset	Specifies the character encoding of text in the Form or URL scope. Used when the character set of the input to a form, or the character set of a URL, is not in UTF-8 encoding.

Tag or function	Attribute or parameter	Use
ToBase64	encoding	Specifies the character encoding of the string being converted to Base 64.
ToString	encoding	Returns a string encoded in the specified character encoding.
URLDecode	charset	Specifies the character encoding of the URL being decoded.
URLEncodedFormat	charset	Specifies the character encoding to use for the URL.

Functions for controlling and using locales

ColdFusion MX provides the following functions that let you specify and identify the locale and format text based on the locale:

Tag or function	Use
GetLocale	Returns the current locale setting.
GetLocaleDisplayName	Returns the name of a locale in the language of a specific locale. The default value is the current locale in the locale's language..
LSCurrencyFormat	Converts numbers into a string in a locale-specific currency format. For countries that use the euro, the result depends on the JVM version.
LSDateFormat	Converts the date part of a date/time value into a string in a locale-specific date format.
LSEuroCurrencyFormat	Converts a number into a string in a locale-specific currency format. Formats using the euro for all countries that use euro as the currency.
LSIsCurrency	Determines whether a string is a valid representation of a currency amount in the current locale.
LSIsDate	Determines whether a string is a valid representation of a date/time value in the current locale.
LSIsNumeric	Determines whether a string is a valid representation of a number in the current locale.
LSNumberFormat	Converts a number into a string in a locale-specific numeric format.
LSParseCurrency	Converts a string that is a currency amount in the current locale into a formatted number. For countries that use the euro, the result depends on the JVM version.
LSParseDateTime	Converts a string that is a valid date/time representation in the current locale into a date-time object.
LSParseEuroCurrency	Converts a string that is a currency amount in the current locale into a formatted number. Requires euro as the currency for all countries that use the euro.
LSParseNumber	Converts a string that is a valid numeric representation in the current locale into a formatted number.

Tag or function	Use
LSTimeFormat	Converts the time part of a date/time value into a string in a locale-specific date format.
SetLocale	Specifies the locale setting.

Note: Many functions that have names starting with LS have corresponding functions that do not have this prefix: DateFormat, IsDate, IsNumeric, NumberFormat, ParseDateTime, and TimeFormat. These function use English (US) locale rules.

If you do not precede calls to the LS functions with a call to the SetLocale function, they use the locale defined by the JVM, which typically is the locale of the operating system.

The following example uses the LSDateFormat function to display the current date in the format for each locale supported by ColdFusion MX:

```
<!--- This example shows LSDateFormat --->
<html>
<head>
<title>LSDateFormat Example</title>
</head>
<body>
<h3>LSDateFormat Example</h3>
<p>Format the date part of a date/time value using the locale convention.
<!--- loop through a list of locales; show date values for Now()--->
<cfloop list = "#Server.Coldfusion.SupportedLocales#"
  index = "locale" delimiters = ",">
  <cfset oldlocale = SetLocale(locale)>

  <cfoutput><p><B><I>#locale#</I></B><br>
    #LSDateFormat(Now(), "mmm-dd-yyyy")#<br>
    #LSDateFormat(Now(), "mmmm d, yyyy")#<br>
    #LSDateFormat(Now(), "mm/dd/yyyy")#<br>
    #LSDateFormat(Now(), "d-mmm-yyyy")#<br>
    #LSDateFormat(Now(), "ddd, mmmm dd, yyyy")#<br>
    #LSDateFormat(Now(), "d/m/yy")#<br>
    #LSDateFormat(Now())#<br>
    <hr noshade>
  </cfoutput>
</cfloop>
</body>
</html>
```

Additional globalization tags and functions

In addition to the tags and functions that are specifically for globalized applications, you might find the following useful when writing a globalized application:

- All string manipulation functions. For more information, see the String functions list in Chapter 3, "ColdFusion Functions" in *CFML Reference*.
- The GetTimeZoneInfo function, which returns the time zone of the operating system.

Handling data in ColdFusion MX

Many of the issues involved with globalizing applications deal with processing data from the various sources supported by ColdFusion MX, including the following:

- General character encoding issues
- Locale-specific content
- Input data from URLs and HTML forms
- File data
- Databases
- E-mail
- HTTP
- LDAP
- WDDX
- COM
- CORBA
- Searching and indexing

This section describes how to handle data from each of these sources.

General character encoding issues

Applications developed for earlier versions of ColdFusion that assumed that the character length of a string was the same as the byte length might produce errors in ColdFusion MX. The byte length of a string depends on the character encoding.

Locale-specific content

The following sections provide information on how to handle locale-specific content in pages that support multiple locales, and how to handle euro values.

Generating multi-locale content

In an application that supports users in multiple locales and produces output that is specific to multiple locales, you call the `SetLocale` function in every request to set the locale for that specific request. When processing has completed, the locale should be set back to its previous value. One useful technique is to save the user's desired locale in a Session variable once the user has selected it, and use the Session variable value to set the locale for each user request during the session.

Supporting the euro

The euro is the currency of many European countries, and ColdFusion supports the reading and writing of correctly formatted euro values. Unlike other supported currencies, the euro is not tied to any single country (or locale). The `LSCurrencyFormat` and `LSParseCurrency` functions rely on the underlying JVM for their operations, and the rules used for currencies depend on the JVM. For Sun JVMs, the 1.3 releases did not support euros and used the older country-specific currencies. The 1.4 releases use euros for all currencies that are in the euro zone as of 2002. If you are using a JVM that does not support the euro, use the `LSEuroCurrencyFormat` and `LSParseEuroCurrency` functions to format and parse euro values in locales that use euros as their currency.

Input data from URLs and HTML forms

A web application server receives character data from request URL parameters or as form data.

The HTTP 1.1 standard only allows US-ASCII characters (0-127) for the URL specification and for message headers. This requires a browser to encode the non-ASCII characters in the URL, both address and parameters, by escaping (URL encoding) the characters using the "%xx" hexadecimal format. URL encoding, however, does not determine how the URL is used in a web document. It only specifies how to encode the URL.

Form data uses the message headers to specify the encoding used by the request (Content headers) and the encoding used in the response (Accept headers). Content negotiation between the client and server uses this information.

This section contains suggestions on how you can handle both URL and form data entered in different character encodings.

Handling URL strings

URL requests to a server often contain name/value pairs as part of the request. For example, the following URL contains name/value pairs as part of the URL:

http://company.com/prod_page.cfm?name=Stephen;ID=7645

As discussed previously, URL characters entered using any character encoding other than US-ASCII are URL-encoded in a hexadecimal format. However, by default, a web server assumes that the characters of a URL string are single-byte characters.

One common method used to support non-ASCII characters within a URL is to include a name/value pair within the URL that defines the character encoding of the URL. For example, the following URL uses a parameter called *encoding* to define the character encoding of the URL parameters:

http://company.com/prod_page.cfm?name=Stephen;ID=7645;encoding=Latin-1

Within the prod_page.cfm page, you can check the value of the encoding parameter before processing any of the other name/value pairs. This guarantees that you will handle the parameters correctly.

You can also use the `SetEncoding` function to specify the character encoding of URL parameters. The `SetEncoding` function takes two parameters: the first specifies a variable scope and the second specifies the character encoding used by the scope. Since ColdFusion writes URL parameters to the URL scope, you specify "URL" as the scope parameter to the function.

For example, if the URL parameters were passed using Shift-JIS, you could access them as follows:

```
<cfscript>
     setEncoding("URL", "Shift_JIS");
     writeoutput(URL.name);
     writeoutput(URL.ID);
</cfscript>
```

Note: To specify the Shift-JIS character encoding, use the Shift_JIS attribute, with an underscore (_), not a hyphen (-).

Handling form data

The HTML `form` tag and the ColdFusion `cfform` tag let users enter text on a page, then submit that text to the server. The form tags are designed to work only with single-byte character data. Since ColdFusion uses two bytes per character when it stores strings, ColdFusion converts each byte of the form input into a two-byte representation.

However, if a user enters double-byte text into the form, the form interprets each byte as a single character, rather than recognize that each character is two bytes. This will corrupt the input text, as the following example shows:

1. A customer enters three double-byte characters in a form, represented by six bytes.
2. The form returns the six bytes to ColdFusion as six characters. ColdFusion converts them to a representation using two bytes per input byte for a total of twelve bytes.
3. Outputting these characters results in corrupt information displayed to the user.

To work around this issue, use the `SetEncoding` function to specify the character encoding of input form text. The `SetEncoding` function takes two parameters: the first specifies the variable scope and the second specifies the character encoding used by the scope. Since ColdFusion writes form parameters to the Form scope, you specify "Form" as the scope parameter to the function. If the input text is double-byte, ColdFusion preserves the two-byte representation of the text.

The following example specifies that the form data contains Korean characters:

```
<cfscript>
  setEncoding("FORM", "EUC-KR");
</cfscript>
<h1> Form Test Result </h1>
<strong>Form Values :</strong>

<cfset text = "String = #form.input1# , Length = #len(Trim(form.input1))#">
<cfoutput>#text#</cfoutput>
```

File data

You use the `cffile` tag to write to and read from text files. By default, the `cffile` tag assumes that the text that you are reading, writing, copying, moving, or appending is in the JVM default file character encoding, which is typically the system default character encoding. For `cffile action="Read"`, ColdFusion MX also checks for a byte order mark (BOM) at the start of the file; if there is one, it uses the character encoding that the BOM specifies.

Note: To specify the JVM default file character encoding, use the `-Dfile.encoding=` switch in the JVM Arguments field on the Java and JVM page in the ColdFusion MX Administrator.

Problems can arise if the file character encoding does not correspond to JVM character encoding, particularly if the number of bytes used for characters in one encoding does not match the number of bytes used for characters in the other encoding.

For example, assume that the JVM default file character encoding is ISO 8859-1, which uses a single byte for each character, and the file uses Shift-JIS, which uses a two-byte representation for many characters. When reading the file, the `cffile` tag treats each byte as an ISO 8859-1 character, and converts it into its corresponding two-byte Unicode representation. Because the characters are in Shift-JIS, the conversion corrupts the data, converting each two-byte Shift-JIS character into two Unicode characters.

To enable the `cffile` tag to correctly read and write text that is not encoded in the JVM default character encoding, you can pass the `charset` attribute to it. Specify as a value the character encoding of the data to read or write, as the following example shows:

```
<cffile action="read"
  charset="EUC-KR"
  file = "c:\web\message.txt"
  variable = "Message" >
```

Databases

ColdFusion applications access databases using drivers for each of the supported database types. The conversion of client native language data types to SQL data types is transparent and is done by the driver managers, database client, or server. For example, the character data (SQL CHAR, VARCHAR) you use with JDBC API is represented using Unicode-encoded strings.

Database administrators configure data sources and usually are required to specify the character encodings for character column data. Many of the major vendors, such as Oracle, Sybase, and Informix, support storing character data in many character encodings, including Unicode UTF-8 and UTF-16.

The database drivers supplied with ColdFusion MX correctly handle data conversions from the database native format to the ColdFusion Unicode format. You should not have to perform any additional processing to access databases. However, you should always check with your database administrator to determine how your database supports different character encodings.

E-mail

ColdFusion sends e-mail messages using the `cfmail`, `cfmailparam`, and `cfmailpart` tags.

By default, ColdFusion sends mail in UTF-8 encoding. You can specify a different default encoding on the Mail page in the ColdFusion MX Administrator, and you can use the `charset` attribute of the `cfmail` and `cfmailpart` tags to specify the character encoding for a specific mail message or part of a multipart mail message.

HTTP

ColdFusion MX supports HTTP communication using the `cfhttp` and `cfhttpparam` tags and the `GetHttpRequestData` function.

The `cfhttp` tag supports making HTTP requests. The `cfhttp` tag uses the Unicode UTF-8 encoding for passing data by default, and you can use the `charset` attribute to specify the character encoding. You can also use the `cfhttpparam` tag `mimeType` attribute to specify the MIME type and character set of a file.

LDAP

ColdFusion MX supports LDAP (Lightweight Directory Access Protocol) through the `cfldap` tag. LDAP uses the UTF-8 encoding format, so you can mix all retrieved data with other data and safely manipulated it. No extra processing is required to support LDAP.

WDDX

ColdFusion MX supports the `cfwddx` tag. ColdFusion stores WDDX (Web Distributed Data Exchange) data as UTF-8 encoding, so it automatically supports double-byte character encodings. You do not have to perform any special processing to handle double-byte characters with WDDX.

COM

ColdFusion MX supports COM through the `cfobject type="com"` tag. All string data used in COM interfaces is constructed using wide characters (wchars), which support double-byte characters. You do not have to perform any special processing to interface with COM objects.

CORBA

ColdFusion MX supports CORBA through the `cfobject type="corba"` tag. The CORBA 2.0 interface definition language (IDL) basic type “String” used the Latin-1 character encoding, which used the full 8-bits (256) to represent characters.

As long as you are using CORBA later than version 2.0, which includes support for the IDL types wchar and wstring, which map to Java types char and string respectively, you do not have to do anything to support double-byte characters.

However, if you are using a version of CORBA that does not support wchar and wstring, the server uses char and string data types, which assume a single-byte representation of text.

Searching and indexing

ColdFusion MX supports Verity search through the `cfindex`, `cfcollection`, and `cfsearch` tags. To support multilingual searching, the ColdFusion MX product CD-ROM includes the Verity language packs that you install to support different languages.

CHAPTER 18
Debugging and Troubleshooting Applications

Macromedia ColdFusion MX provides detailed debugging information to help you resolve problems with your application. This chapter describes how you configure ColdFusion MX to provide debugging information, how to understand the information it provides, and how to use the `cftrace` and `cftimer` tags to provide detailed information on code execution. It also provides additional information on tools for validating your code before you run it and techniques for troubleshooting particular problems.

Note: Macromedia Dreamweaver MX provides integrated tools for displaying and using ColdFusion debugging output. For information on using these tools, see the Dreamweaver MX online Help.

Contents

Configuring debugging in the ColdFusion MX Administrator 423

Using debugging information from browser pages 426

Controlling debugging information in CFML 434

Using the cftrace tag to trace execution 436

Using the cftimer tag to time blocks of code 440

Using the Code Compatibility Analyzer 442

Troubleshooting common problems 443

Configuring debugging in the ColdFusion MX Administrator

ColdFusion can provide important debugging information for every application page requested by a browser. The ColdFusion MX Administrator lets you specify which debugging information to make available and how to display it. The following sections briefly describe the Administrator settings. For more information, see the online Help for the Debugging pages.

Debugging Settings page

In the Administrator, the following options on the Debugging Settings page determine the information that ColdFusion displays in debugging output:

Option	Description
Enable Robust Exception Information	Enables the display of the following information when ColdFusion displays the exception error page. (Cleared by default.) • Path and URL of the page that caused the error • Line number and short snippet of the code where the error was identified • Any SQL statement and data source • Java stack trace
Enable Debugging	Enables debugging output. When this option is cleared, no debugging information is displayed, including all output of `cftrace` and `cftimer` calls. (Cleared by default.) You should disable debugging output on production servers. Doing so increases security by ensuring that users cannot see debugging information. It also improves server response times. You can also limit debugging output to specific IP addresses; for more information, see "Debugging IP addresses page" on page 426.
Select Debugging Output Format	Determines how to display debugging output: • The classic.cfm template (the default) displays information as plain HTML text at the bottom of the page. • The dockable.cfm template uses DHTML to display the debugging information using an expanding tree format in a separate window. This window can be either a floating pane or docked to the browser window. For more information on the dockable output format, see "Using the dockable.cfm output format" on page 433.
Report Execution Times	Lists ColdFusion pages that run as the result of an HTTP request and displays execution times, ColdFusion also highlights in red pages with processing times greater than the specified value, and you can select between a summary display or a more detailed, tree structured, display.
General Debug Information	Displays general information about the request: ColdFusion MX Version, Template, Time Stamp, User Locale, User Agent, User IP, and Host Name.
Database Activity	Displays debugging information about access to SQL data sources and stored procedures. (Selected by default.)
Exception information	Lists all ColdFusion exceptions raised in processing the request. (Selected by default.)
Tracing information	Displays an entry for each `cftrace` tag. When this option is cleared, the debugging output does not include tracing information, but the output page does include information for `cftrace` tags that specify `inline="Yes"`. (Selected by default.) For more information on using the `cftrace` tag, see"Using the cftrace tag to trace execution" on page 436.

Option	Description
Variables	Enables the display of ColdFusion variable values. When this option is cleared, disables display of all ColdFusion variables in the debugging output. (Selected by default.) When enabled, ColdFusion displays the values of variables in the selected scopes. You can select to display the contents of any of the ColdFusion scopes except Variables, Attributes, Caller, and ThisTag. To enhance security, Application, Server, and Request variable display is disabled by default,
Enable Performance Monitoring	Allows the standard NT Performance Monitor application to display information about a running ColdFusion application server.
Enable CFSTAT	Enables you to use of the `cfstat` command line utility to monitor real-time performance. This utility displays the same information that ColdFusion writes to the NT System Monitor, without using the System Monitor application. For information on the `cfstat` utility, see *Configuring and Administering ColdFusion MX*.

The following figure shows a sample debugging output using the classic output format:

Debugging Information

ColdFusion Server Evaluation 6,0,0,44589	
Template	/MYStuff/NeoDocs/debug/simplepage.cfm
Time Stamp	11-Apr-02 12:33 PM
Locale	English (US)
User Agent	Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 4.0; T312461)
Remote IP	127.0.0.1
Host Name	localhost

Execution Time (average time over 250 ms)

Total Time	Avg Time	Count	Template
60 ms	60 ms	1	C:\CFusionMX\wwwroot\MYStuff\NeoDocs\debug\simplepage.cfm
0 ms	0 ms	1	C:\CFusionMX\wwwroot\MYStuff\NeoDocs\debug\Application.cfm
70 ms			*STARTUP, PARSING, COMPILING, LOADING, & SHUTDOWN*

SQL Queries

```
myQuery (Datasource=CompanyInfo, Time=0ms, Records=1, Cached Query) in C @ 12:33:03.003

Select *
From Employee
Where Emp_ID=1
```

Debugging IP addresses page

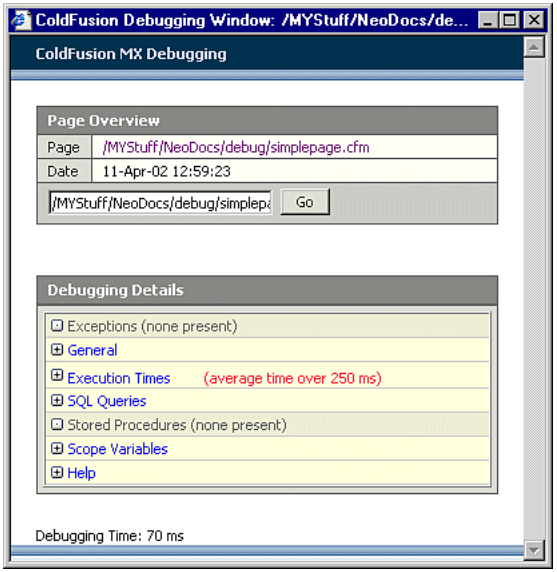
By default, when you enable debugging output, the output is visible only to local users (that is, via IP address 127.0.0.1). You can specify additional IP addresses whose users can see debugging output, or even disable output to local users. In the Administrator, use the Debugging IPs page to specify the addresses that can receive debugging messages.

Note: If you must enable debugging on a production server, for example to help locate the cause of a difficult problem, use the Debugging IP Addresses page to limit the output to your development systems and prevent clients from seeing the debugging information.

Using debugging information from browser pages

The ColdFusion debugging output that you configure in the Administrator displays whenever an HTML request completes. It represents the server conditions at the end of the request. For information on displaying debugging information while a request is processed, see "Using the cftrace tag to trace execution" on page 436.

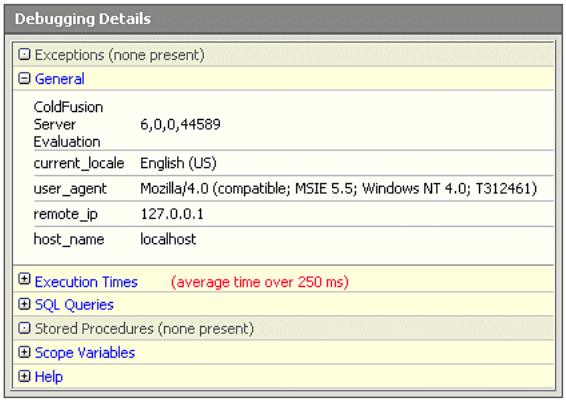
The following figure shows a sample collapsed debugging output using the dockable.cfm debugging output format. The next sections show each of the debugging sections and describe how you can use the information they display.



General debugging information

ColdFusion displays general debugging information. In the classic.cfm output format, the information is at the top of the debugging output. In the dockable.cfm output format, it looks like the following figure:





(In the classic.cfm output format, the section is first in the debugging output and has no heading.)

The general debugging information includes the following values. The table lists the names used in the classic output template view.

Name	Description
ColdFusion	The ColdFusion MX version.
Template	The requested template. (In the dockable.cfm format, this appears in the Page Overview section and is called Page.)
TimeStamp	The time the request was completed. (In the dockable.cfm format, this appears in the Page Overview section and is called Date.)
Locale	The locality and language that determines how information is processed, particularly the message language.
User Agent	The identity of the browser that made the HTTP request.
Remote IP	The IP address of the client system that made the HTTP request.
Host Name	The name of the host running the ColdFusion server that executed the request.

Execution Time

The Execution Time section displays the time required to process the request. It displays information about the time required to process all pages required for the request, including the Application.cfc, Application.cfm, and OnRequestEnd.cfm pages, if used, and any CFML custom tags, pages included by the `cfinclude` tag, and any ColdFusion component (CFC) pages.

Tip: To display execution time for a specific block of code, use the `cftimer` tag.

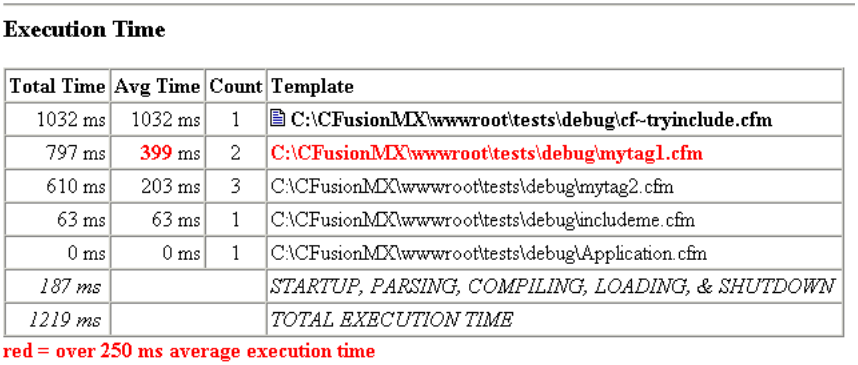
You can display the execution time in two formats:

- Summary
- Tree

Note: Execution tine decreases substantially between the first and second time you use a page after creating it or changing it. The first time ColdFusion uses a page it compiles the page into Java bytecode, which the server saves and loads into memory. Subsequent uses of unmodified pages do not require recompilation of the code, and therefore are substantially faster.

Summary execution time format

The summary format displays one entry for each ColdFusion page processed during the request. If a page is processed multiple times it appears only once in the summary. For example, if a custom tag gets called three time in a request, it appears only once in the output. In the classic.cfm output format, the summary format looks like the following figure:



Execution Time

Total Time	Avg Time	Count	Template
1032 ms	1032 ms	1	C:\CFusionMX\wwwroot\tests\debug\cf-tryinclude.cfm
797 ms	399 ms	2	C:\CFusionMX\wwwroot\tests\debug\mytag1.cfm
610 ms	203 ms	3	C:\CFusionMX\wwwroot\tests\debug\mytag2.cfm
63 ms	63 ms	1	C:\CFusionMX\wwwroot\tests\debug\includeme.cfm
0 ms	0 ms	1	C:\CFusionMX\wwwroot\tests\debug\Application.cfm
187 ms			*STARTUP, PARSING, COMPILING, LOADING, & SHUTDOWN*
1219 ms			*TOTAL EXECUTION TIME*

red = over 250 ms average execution time

The following table describes the display fields:

Column	Description
Total Time	The total time required to process all instances of the page *and all pages that it uses*. For example, if a request causes a page to be processed two times, and the page includes another page, the total time includes the time required to process both pages twice.
Avg Time	The average time for processing each instance of this page and the pages that it uses. The Avg Time multiplied by the Count equals the Total Time.

Column	Description
Count	The number of times the page is processed for the request.
Template	The path name of the page.

The page icon indicates the requested page.

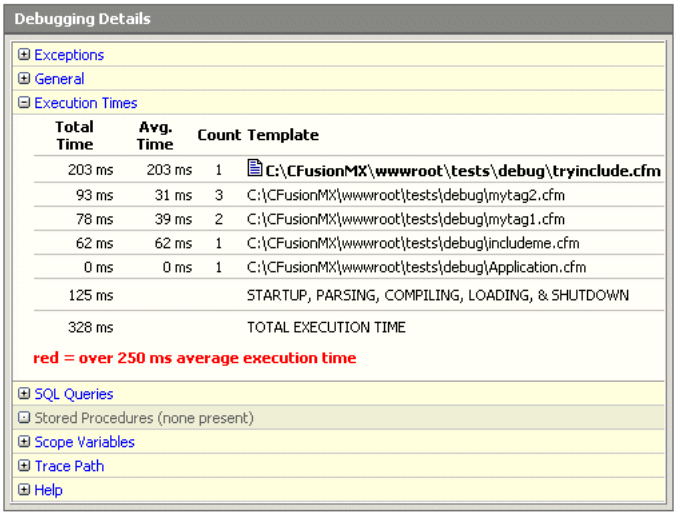
Any page with an average processing time that exceeds the highlight value that you set on the Debugging Settings page in the ColdFusion MX Administrator appears in red.

The next to last line of the output displays the time that ColdFusion took to parse, compile, and load pages, and to start and end page processing. This figure is not included in the individual page execution times. The last line shows the sum of all the time it took to process the request.

Tree execution time format

The tree execution time format is a hierarchical, detailed view of how ColdFusion processes each page. If a page includes or calls second page, the second page appears below and indented relative to the page that uses it. Each page appears once for each time it is used. Therefore, if a page gets called three times in processing a request, it appears three times in the tree. Therefore the tree view displays both processing times and an indication of the order of page processing.

The tree format looks as follows in the dockable.cfm output format:



As in the summary view, the execution times (in parentheses) show the times to process the listed page and all pages required to process the page, that is, all pages indented below the page in the tree.

By looking at this output in this figure you can determine the following information:

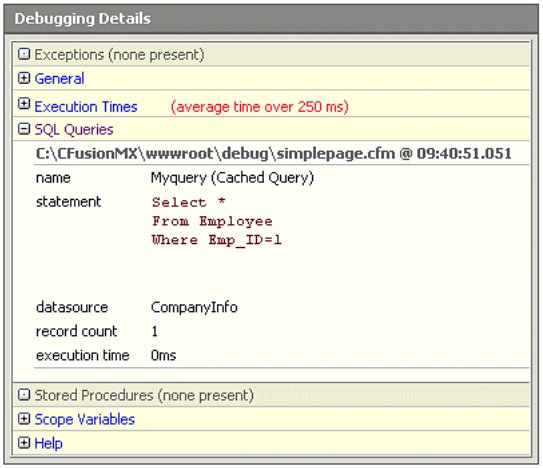
- ColdFusion took 0 ms to process an Application.cfm page as part of the request.
- The requested page was tryinclude.cfm. It took 203 ms to process this page and all pages required to execute it. The code directly on this page took 71 milliseconds (203 - 93 - 16 - 23) to process.
- The mytag2.cfm page was processed three times. All processing took 93 milliseconds, and the average processing time was 31 milliseconds. (This page does not call any other pages.)
- The mytag1.cfm page was processed two times. All processing took 78 milliseconds, and the average processing time was 39 milliseconds. This time included the time to process mytag2.cfm (this tag calls the mytag2 custom tag); therefore, the code directly on the page took an average of 8 milliseconds and a total of 16 milliseconds to process.
- The includeme.cfm page took about 62 ms to process. This processing time includes the time to process the mytag1.cfm, and therefore also the time to process mytag2.cfm once. Therefore the code directly on the page took 23 milliseconds (62-39) to process.
- ColdFusion took 125 ms for processing that was not associated with a specific page.
- The total processing time was 328 milliseconds, the sum of 125 + 203.

Database Activity

In the Administrator, when Database Activity is selected on the Debugging Settings page, the debugging output includes information about database access.

SQL Queries

The SQL Queries section provides information about tags that generate SQL queries or result in retrieving a cached database query: `cfquery`, `cfinsert`, `cfgridupdate`, and `cfupdate`. The section looks like the following figure in the dockable.cfm output format:



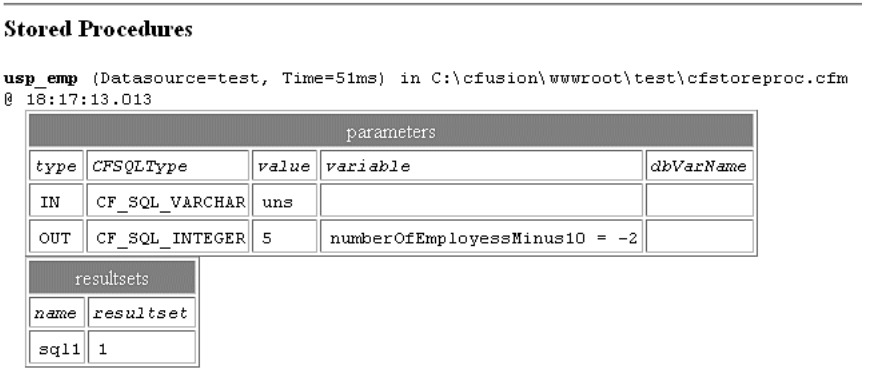
The output displays the following information:

- Page on which the query is located.
- The time when the query was made.
- Query name.
- An indicator if the result came from a cached query.
- SQL statement, including the results of processing any dynamic elements such as CFML variables and `cfqueryparam` tags. This information is particularly useful because it shows the results of all ColdFusion processing of the SQL statement.
- Data source name.
- Number of records returned; 0 indicates no match to the query.
- Query execution time.
- Any query parameters values from `cfqueryparam` tags.

Stored Procedures

The stored procedures section displays information about the results of using the `cfstoredproc` tag to execute a stored procedure in a database management system.

The Stored Procedures section looks as follows in the classic.cfm output format:



Stored Procedures

usp_emp (Datasource=test, Time=51ms) in C:\cfusion\wwwroot\test\cfstoreproc.cfm
@ 18:17:13.013

parameters				
type	*CFSQLType*	*value*	*variable*	*dbVarName*
IN	CF_SQL_VARCHAR	uns		
OUT	CF_SQL_INTEGER	5	numberOfEmployessMinus10 = -2	

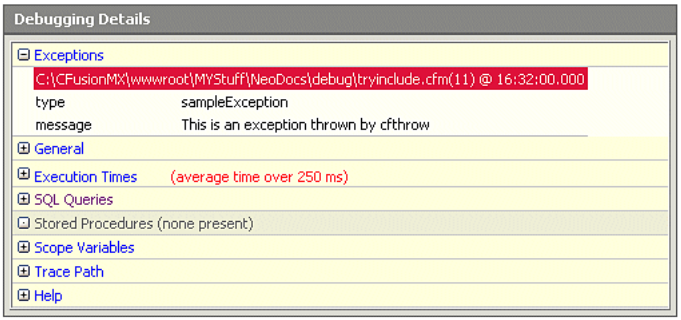
resultsets	
name	*resultset*
sql1	1

The output displays the following information:

- Stored procedure name
- Data source name
- Query execution time
- Page on which the query is located.
- The time when the query was made.
- A table displaying the procedure parameters sent and received, as specified in the `cfprocparam` tags, including the `ctype`, `CFSQLType`, `value` `variable`, and `dbVarName` attributes. The `variable` information for OUT and INOUT parameters includes the returned value.
- A table listing the procedure result sets returned, as specified in the `cfprocresult` tag.

Exceptions

In the Administrator, when Exception Information is selected on the Debugging Settings page, the debugging output includes a list of all ColdFusion exceptions raised in processing the application page. This section looks like the following figure when displaying information about an exception thrown by the `cfthrow` tag using the dockable.cfm output format:

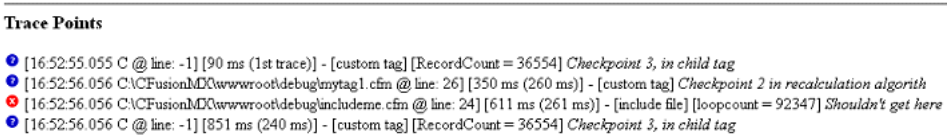


The exception information includes information about any application exceptions that are caught and handled by your application code or by ColdFusion MX.

Exceptions represent events that disrupt the normal flow of an application. You should catch and, whenever possible, recover from forseeable exceptions in your application, as described in Chapter 14, "Handling Errors," on page 307. However, you might also want to be alerted to caught exceptions when you are debugging your application. For example, if a file is missing, your application can catch the `cffile` exception and use a backup or default file instead. If you enable exception information in the debugging output, you can immediately see when this happens.

Trace points

In the Administrator, when Tracing Information is selected on the Debugging Settings page, the debugging output includes the results of all `cftrace` tags, including all tags that display their results in-line. Therefore, the debugging output contains a historical record of all trace points encountered in processing the request. This section looks like the following figure when you use the classic.cfm output format:



For more information on using the `cftrace` tag, see "Using the cftrace tag to trace execution" on page 436.

Scope variables

In the Administrator, when the Variables option and one or more variable scopes are selected on the Debugging Settings page, the debugging output displays the values of all variables in the selected scopes. The debugging output displays the values that result after all processing of the current page.

By displaying selected scope variables you can determine the effects of processing on persistent scope variables, such as application variables. This can help you locate problems that do not generate exceptions.

The Form, URL, and CGI scopes are useful for inspecting the state of a request. They let you inspect parameters that affect page behavior, as follows:

URL variables Identify the HTTP request parameters.

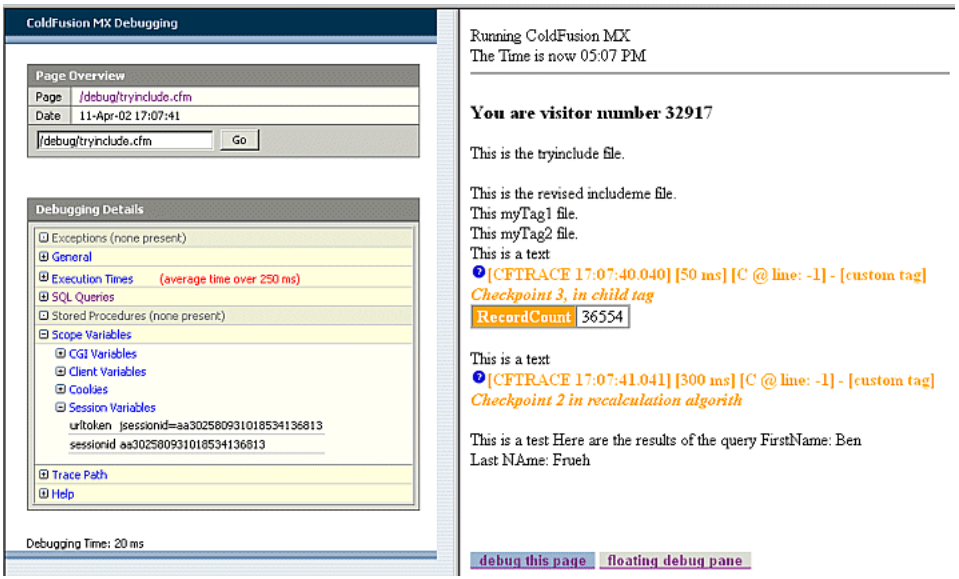
Form variables Identify the form fields posted to an action page.

CGI variables Provide a view of the server environment following the request.

Similarly, the Client, Session, Application, and Server scope variables show the global state of the application, and can be useful in tracing how each page affects the state of the ColdFusion persistent variables.

Using the dockable.cfm output format

The dockable.cfm output format has several features that are not included in the classic.cfm debugging display, as shown in the following figure of a docked debug pane:



Application page selections

ColdFusion displays two buttons at the bottom of each page, as described in the following table:

Button	Description
Debug This page	Tells ColdFusion to display the debugging information for the selected frame. Refreshes the debug pane if you select it for the current frame (or the application does not use frames).
Floating/Docked debug pane	Toggles the display between a floating window and a pane docked to the left of the selected frame.

Debug pane features

The debug pane has the following features:

- You can expand and collapse each debugging information category, such as Exceptions, by clicking on the plus or minus sign (+ or -) in front of each category heading. You can also expand and collapse each scope data type display in the Scoped Variables section.
- The top of the debug pane displays the URL of the application page being debugged (as identified by the cgi.script_name variable). Click this link to refresh the page and display the debugging information that results. (You can also refresh the page and debugging information by using your browser's Refresh button or key.)
- The debug pane also displays a box where you can enter a page pathname or URL. When you click the Go button, ColdFusion processes the page and the debug pane is updated with the debugging information for the new page.

Controlling debugging information in CFML

The following sections describe how you can use CFML tags and functions to display or hide debugging and tracing information.

Generating debugging information for an individual query

In the Administrator, the `cfquery` tag `debug` attribute overrides the Database Activity setting on the Debugging Settings page. The `debug` attribute has an effect *only* when debugging output is enabled on the Debugging Settings page, as follows:

- If Database Activity is selected in the Administrator, specify `debug="No"` to prevent ColdFusion from displaying the query's SQL and statistics in the debugging output.
- If Database Activity is not selected in the Administrator, specify `debug="Yes"` or `debug` to have ColdFusion display the query's SQL and statistics in the debugging output.

For example, if Database Activity is not selected in the Administrator, you can use the following code to show the query execution time, number of records returned, ColdFusion page, timestamp, and the SQL statement sent to the data source for this query only:

```
<cfquery name="TestQuery" datasource="cfdocexamples" debug>
  SELECT * FROM TestTable
</cfquery>
```

The `debug` attribute can be useful to disable query debugging information generated by queries in custom tags that you call frequently, so that you only see the debugging information for queries in pages that call the tags.

You can also view stored procedure-specific debugging information by specifying the `debug` attribute in the `cfstoredproc` tag.

Controlling debugging output with the cfsetting tag

Use the `cfsetting` tag `showDebugOutput` attribute to turn off debugging output for a specific page. The attribute controls debugging output *only* if the Debugging Settings page in the ColdFusion MX Administrator enables debugging output. The attribute's default value is Yes. The following tag suppresses all debugging output for the current page:

```
<cfsetting showDebugOutput="No">
```

You can put this tag in the initialization code of the Application.cfc file or on your Application.cfm page to suppress all debugging output for an application, and override it on specific pages by setting `showDebugOutput="Yes"` in `cfsetting` tags on those pages. Conversely, you can leave debugging on for the application, and use the `cfsetting showDebugOutput="No"` tag to suppress debugging on individual pages where the output could cause errors or confusion.

You can also use the `showDebugOutput` attribute to control debugging output if you do not have access to the ColdFusion MX Administrator, but only if the Administrator enables debugging.

Using the IsDebugMode function to run code selectively

The `IsDebugMode` function returns True if debugging is enabled. You can use this function in a `cfif` tag condition to selectively run code only when debugging output is enabled. The `IsDebugMode` function lets you tell ColdFusion to run *any* code in debug mode, so it provides more flexibility than the `cftrace` tag for processing and displaying information.

You can use the `IsDebugMode` function to selectively log information only when debugging is enabled. Because you control the log output, you have the flexibility of silently logging information without displaying trace information in the browser. For example, the following code logs the application page, the current time, and the values of two variables to the log file MyAppSilentTrace.log when debugging is enabled:

```
<cfquery name="MyDBQuery" datasource="cfdocexamples">
  SELECT *
  FROM Employee
</cfquery>
<cfif IsDebugMode()>
  <cflog file="MyAppSilentTrace" text="Page: #cgi.script_name#,
  completed query MyDBQuery; Query Execution time:
  #cfquery.ExecutionTime# Status: #Application.status#">
</cfif>
```

Tip: If you use `cfdump` tags frequently for debugging, put them in `<cfif IsDebugMode()>` tags; for example `<cfif IsDebugMode()><cfdump var=#myVar#></cfif>`. This way you ensure that if you leave any `cfdump` tags in production code, they are not displayed when you disable debugging output.

Using the cftrace tag to trace execution

The cftrace tag displays and logs debugging data about the state of your application at the time the cftrace tag executes. You use it to provide "snapshots" of specific information as your application runs.

About the cftrace tag

The cftrace tag provides the following information:

- A severity identifier specified by the cftrace tag type attribute
- A timestamp indicating when the cftrace tag executed
- The time elapsed between the start of processing the request and when the current cftrace tag executes.
- The time between any previous cftrace tag in the request and the current one. If this is the first cftrace tag processed for the request, the output indicates "1st trace". ColdFusion does not display this information in inline trace output, only the log and in the standard debugging output.
- The name of the page that called the cftrace tag
- The line on the page where the cftrace call is located
- A trace category specified by the category attribute
- A message specified by the text attribute
- The name and value, at the time the cftrace call executes, of a single variable specified by the var attribute

A typical cftrace tag might look like the following:

```
<cftrace category="UDF End" inline = "True" var = "MyStatus"
  text = "GetRecords UDF call has completed">
```

You can display the cftrace tag output in either or both of the following ways:

- **As a section in the debugging output** To display the trace information in the debugging output, in the Administrator, select Tracing Information on the Debugging Settings page.
- **In-line in your application page** When you specify the inline attribute in a cftrace tag, ColdFusion displays the trace output on the page at the cftrace tag location. (An inline cftrace tag does not display any output if it is inside a cfsilent tag block.)

The cftrace tag executes only if you select Enable Debugging on the ColdFusion MX Administrator Debugging Settings page. To display the trace results in the debugging output, you must also specify Tracing Information on the Debugging Settings page; otherwise, the trace information is logged and inline traces are displayed, but no trace information appears in the debugging output.

Note: When you use in-line trace tags, ColdFusion sends the page to the browser after all page processing is completed, but before it displays the debugging output from the debug template. As a result, if an error occurs after a trace tag but before the end of the page, ColdFusion might not display the trace for that tag.

An in-line trace messages look like the following:



The following table lists the displayed information:

Entry	Meaning
	Trace type (severity) specified in the `cftrace` call; in this case, Information.
[CFTRACE 13:21:11.011]	Time when the `cftrace` tag executed.
[501 ms]	Time taken for processing the current request to the point of the `cftrace` tag.
[C:\CusionMX\wwwroot\MYStuff\mydocs\tractest.cfm]	Path in the web server of the page that contains the `cftrace` tag.
@ line:14	The line number of the `cftrace` tag.
[UDF End]	Value of the `cftrace` tag `category` attribute.
GetRecords UDF call has completed	The `cftrace` tag `text` attribute with any variables replaced with their values.
MyStatus Success	Name and value of the variable specified by the `cftrace` tag `var` attribute.

ColdFusion logs all `cftrace` output to the file logs\cftrace.log in your ColdFusion installation directory.

A log file entry looks like the following:

```
"Information","web-29","04/01/02","13:21:11","MyApp","[501 ms (1st trace)]
  [C:\CFusionMX7\wwwroot\MYStuff\mydocs\tractest.cfm @ line: 14] - [UDF End]
  [MyStatus = Success] GetRecords UDF call has completed "
```

This entry is in standard ColdFusion log format, with comma-delimited fields inside double-quote characters. The information displayed in the trace output is in the last, message, field.

The following table lists the contents of the trace message and the log entries. For more information on the log file format, see "Logging errors with the cflog tag" on page 321.

Entry	Meaning
Information	The Severity specified in the `cftrace` call.
web-29	Server thread that executed the code.
04/01/02	Date the trace was logged.
13:21:11	Time the trace was logged.
MyApp	The application name, as specified in a `cfapplication` tag.

Entry	Meaning
501 ms (1st trace)]	The time ColdFusion took to process the current request up to the `cftrace` tag, This is the first `cftrace` tag processed in this request. If there had been a previous `cftrace` tag, the parentheses would contain the number of milliseconds between when the previous `cftrace` tag ran and when this tag ran.
[C:\CFusionMX7\wwwroot \MYStuff\mydocs\ tracetest.cfm @ line: 14]	Path of the page on which the trace tag is located and the line number of the `cftrace` tag on the page.
[UDF End]	Value of the `cftrace` tag `category` attribute.
[MyStatus = Success]	Name and value of the variable specified by the `cftrace` tag `var` attribute. If the variable is a complex data type, such as an array or structure, the log contains the variable value and the number of entries at the top level of the variable, such as the number of top-level structure keys.
GetRecords UDF call has completed	The `cftrace` tag `text` attribute with any variables replaced with their values.

Using tracing

As its name indicates, the `cftrace` tag is designed to help you trace the execution of your application. It can help you do any of several things:

- You can time the execution of a tag or code section. This capability is particularly useful for tags and operations that can take substantial processing time. Typical candidates include all ColdFusion tags that access external resources, including `cfquery`, `cfldap`, `cfftp`, `cffile`, and so on. To time execution of any tag or code block, call the `cftrace` tag before and after the code you want to time.
- You can display the values of internal variables, including data structures. For example, you can display the raw results of a database query.
- You can display an intermediate value of a variable. For example, you could use this tag to display the contents of a raw string value before you use string functions to select a substring or format it.
- You can display and log processing progress. For example, you can put a `cftrace` call at the head of pages in your application or before critical tags or calls to critical functions. (Doing this could result in massive log files in a complex application, so you should use this technique with care.)
- If a page has many nested `cfif` and `cfelseif` tags you can put `cftrace` tags in each conditional block to trace the execution flow. When you do this, you should use the condition variable in the message or `var` attribute.
- If you find that the ColdFusion server is hanging, and you suspect a particular block of code (or call to a cfx tag, COM object, or other third-party component), you can put a `cftrace` tag before and after the suspect code, to log entry and exit.

Calling the cftrace tag

The `cftrace` tag takes the following attributes. All attributes are optional.

Attribute	Purpose
abort	A Boolean value. If you specify True, ColdFusion stops processing the current request immediately after the tag. This attribute is the equivalent of placing a `cfabort` tag immediately after the `cftrace` tag. The default is False. If this attribute is True, the output of the `cftrace` call appears only in the cftrace.log file. The line in the file includes the text "[ABORTED]".
category	A text string specifying a user-defined trace type category. This attribute lets you identify or process multiple trace lines by categories. For example, you could sort entries in a log according to the category. The `category` attribute is designed to identify the general purpose of the trace point. For example, you might identify the point where a custom tag returns processing to the calling page with a "Custom Tag End" category. You can also use finer categories; for example, by identifying the specific custom tag name in the category. You can include simple ColdFusion variables, but not arrays, structures, or objects, in the category text by enclosing the variable name in number signs (#).
inline	A Boolean value. If you specify True, ColdFusion displays trace output in-line in the page. The default is False. The `inline` attribute lets you display the trace results at the place that the `cftrace` call is processed. This provides a visual cue directly in the ColdFusion page display. Trace output also appears in a section in the debugging information display.
text	A text message describing this trace point. You can include simple ColdFusion variables, but not arrays, structures, or objects, in the text output by enclosing the variable name in number signs (#).

Attribute	Purpose
type	A ColdFusion logging severity type. The inline trace display and dockable.cfm output format show a symbol for each type. The default debugging output shows the type name, which is also used in the log file. The type name must be one of the following:  Information (default)  Warning  Error  Fatal Information
var	The name of a single variable that you want displayed. This attribute can specify a simple variable, such as a string, or a complex variable, such as a structure name. Do not surround the variable name in number signs. Complex variables are displayed in inline output in `cfdump` format; the debugging display and log file report the number of elements in the complex variable, instead of any values. You can use this attribute to display an internal variable that the page does not normally show, or an intermediate value of a variable before the page processes it further. To display a function return value, put the function inside the message. Do not use the function in the `var` attribute, because the attribute cannot evaluate functions.

Note: If you specify inline trace output, and a `cftrace` tag is inside a `cfsilent` tag block, ColdFusion does not display the trace information in line, but does include it in the standard debugging display.

The following `cftrace` tag displays the information shown in the example output and log entry in the "About the cftrace tag" section:

```
<cftrace abort="False" category="UDF End" inline = "True" text = "GetRecords
  UDF
  call has completed" var = "MyStatus">
```

Using the cftimer tag to time blocks of code

The `cftimer` tag displays execution time for a specified section of CFML code.

Using timing

Use this tag to determine how long it takes for a block of code to execute. This is particularly useful when ColdFusion debugging output indicates excessive execution time, but does not pinpoint the long-running block of code.

To use this tag, you must enable debugging in the ColdFusion MX Administrator Debugging Settings page. In the Debugging Settings page, you must also specifically enable usage of the `cftimer` tag by checking the Timer Information check box.

Tip: If you enable debugging for the `cftimer` tag only and display timing information in an HTML comment, you can generate timing information without disturbing production users.

Calling the cftimer tag

You can control where the `cftimer` tag displays timing information, as follows:

- **Inline** Displays timing information following the `</cftimer>` tag.
- **Outline** Displays timing information at the beginning of the timed code and draws a box around the timed code. (This requires browser support for the HTML FIELDSET attribute.)
- **Comment** Displays timing information in an HTML comment in the format `<!--` *label*: *elapsed-time* `ms -->`. The default label is `cftimer`.
- **Debug** Displays timing information in the debugging output under the heading CFTimer Times.

The following example calls the `cftimer` tag multiple times, each time using a different `type` attribute:

```
<HTML>
<body>
<h1>CFTIMER test</h1>
<!--- type="inline" --->
     <cftimer label="Query and Loop Time Inline" type="inline">
     <cfquery name="empquery" datasource="cfdocexamples">
          select *
          from Employees
     </cfquery>

     <cfloop query="empquery">
       <cfoutput>#lastname#, #firstname#</cfoutput><br>
     </cfloop>
   </cftimer>
<hr><br>
<!--- type="outline" --->
     <cftimer label="Query and CFOUTPUT Time with Outline" type="outline">
     <cfquery name="coursequery" datasource="cfdocexamples">
          select *
          from CourseList
     </cfquery>

    <table border="1" width="100%">
     <cfoutput query="coursequery">
     <tr>
     <td>#Course_ID#</td>
     <td>#CorName#</td>
     <td>#CorLevel#</td>
     </tr>
     </cfoutput>
     </table>
   </cftimer>
<hr><br>
<!--- type="comment" --->
     <cftimer label="Query and CFOUTPUT Time in Comment" type="comment">
     <cfquery name="parkquery" datasource="cfdocexamples">
          select *
          from Parks
     </cfquery>
```

```
        <p>Select View &gt; Source to see timing information</p>
    <table border="1" width="100%">
    <cfoutput query="parkquery">
    <tr>
    <td>#Parkname#</td>
    </tr>
    </cfoutput>
    </table>
  </cftimer>

<hr><br>
<!--- type="debug" --->
    <cftimer label="Query and CFOUTPUT Time in Debug Output" type="debug">
    <cfquery name="deptquery" datasource="cfdocexamples">
        select *
        from Departments
    </cfquery>
        <p>Scroll down to CFTimer Times heading to see timing information</p>
    <table border="1" width="100%">
    <cfoutput query="deptquery">
    <tr>
    <td>#Dept_ID#</td>
    <td>#Dept_Name#</td>
    </tr>
    </cfoutput>
    </table>
  </cftimer>
</body>
```

Using the Code Compatibility Analyzer

The Code Compatibility Analyzer has two purposes:

- It can validate your application's CFML syntax. To do so, the analyzer runs the ColdFusion compiler on your pages, but does not execute the compiled code. It reports errors that the compiler encounters.
- It can identify places where ColdFusion MX might behave differently than previous versions. The analyzer identifies the following kinds of features:
  - **No longer supported** Their use results in errors. For example, ColdFusion now generates an error if you use the `cflog` tag with the `thread="Yes"` attribute.
  - **Deprecated** They are still available, but their use is not recommended and the they might not be available in future releases. Deprecated features might also behave differently now than in previous releases. For example, the `cfservlet` tag is deprecated.
  - **Modified behavior** They might behave differently than in previous versions. For example, the `StructKeyList` function no longer lists the structure key names in alphabetical order.

  The analyzer provides information about the incompatibility and its severity, and suggests a remedy where one is required.

You can run the Code Compatibility Analyzer from the ColdFusion MX Administrator. Select Code Analyzer from the list of Debugging & Logging pages.

Note: The CFML analyzer does not execute the pages that it checks. Therefore, it cannot detect invalid attribute combinations if the attribute values are provided dynamically at runtime.

For more information on using the Code Compatibility Analyzer, see *Migrating ColdFusion 5 Applications*.

Troubleshooting common problems

This section describes a few common problems that you might encounter and ways to resolve them.

For more information on troubleshooting ColdFusion, see the Macromedia ColdFusion Support Center Testing and Troubleshooting page at http://www.macromedia.com/support/coldfusion/troubleshoot.html. For common tuning and precautionary measurements that can help you prevent technical problems and improve application performance, see the ColdFusion tech tips article, TechNote number 13810. A link to the article is located near the top of the Testing and Troubleshooting page.

CFML syntax errors

Problem: You get an error message such as the following:

```
Encountered "function or tag name" at line 12, column 1.
Encountered "\"" at line 37, column 20.
Encountered "," at line 24, column 61.
Unable to scan the character '\"' which follows "" at line 38, column 53.
```

These errors typically indicate that you have unbalanced <, ", or # characters. One of the most common coding errors is to forget to close quoted code, number sign-delimited variable names, or opening tags. Make sure the code in the identified line and previous lines do not have missing characters.

The line number in the error message often does **not** identify the line that causes the error. Instead, it identifies the first line where the ColdFusion compiler encountered code that it could not handle as a result of the error.

Problem: You get an error message you do not understand.

Make sure all your CFML tags have matching end tags where appropriate. It is a common error to omit the end tag for the `cfquery`, `cfoutput`, `cftable`, or `cfif` tag.

As with the previous problem, the line number in the error message often does **not** identify the line that causes the error, but the first line where the ColdFusion compiler encounters code that it could not handle as a result of the error. Whenever you have an error message that does not appear to report a line with an error, check the code that precedes it for missing text.

Problem: Invalid attribute or value.

If you use an invalid attribute or attribute values, ColdFusion returns an error message. To prevent such syntax errors, use the CFML Code Analyzer. Also see "Using the cftrace tag to trace execution" on page 436.

Problem: You suspect that there are problems with the structure or contents of a complex data variable, such as a structure, array, query object, or WDDX-encoded variable.

Use the `cfdump` tag to generate a table-formatted display of the variable's structure and contents. For example, to dump a structure named `relatives`, use the following line. You must surround the variable name with number signs (#).

```
<cfdump var=#relatives#>
```

Data source access and queries

Problem: You cannot make a connection to the database.

You must create the data source before you can connect. Connection errors can include problems with the location of files, network connections, and database client library configuration.

Create data sources before you refer to them in your application source files. Verify that you can connect to the database by clicking the Verify button on the Data Sources page of the ColdFusion MX Administrator. If you are unable to make a simple connection from that page, you might need to consult your database administrator to help solve the problem.

Also, check the spelling of the data source name.

Problem: Queries take too long.

Copy and paste the query from the Queries section of the debugging output into your database's query analysis tool. Then retrieve and analyze the execution plan generated by the database server's query optimizer. (The method for doing this varies from dbms to dbms.) The most common cause of slow queries is the lack of a useful index to optimize the data retrieval. In general, avoid table scans (or "clustered index" scans) whenever possible.

HTTP/URL

Problem: ColdFusion MX cannot correctly decode the contents of your form submission.

The `method` attribute in forms sent to the ColdFusion server must be Post, for example:

```
<form action="test.cfm" method="Post">
```

Problem: The browser complains or does not send the full URL string when you include spaces in URL parameters.

Some browsers automatically replace spaces in URL parameters with the %20 escape sequence, but others might display an error or just send the URL string up to the first character (as does Netscape 4.7).

URL strings cannot have embedded spaces. Use a plus sign (+) or the standard HTTP space character escape sequence, (%20) wherever you want to include a space. ColdFusion correctly translates these elements into a space.

A common scenario in which this error occurs is when you dynamically generate your URL from database text fields that have embedded spaces. To avoid this problem, include only numeric values in the dynamically generated portion of URLs.

Or, you can use the `URLEncodedFormat` function, which automatically replaces spaces with %20 escape sequences. For more information on the `URLEncodedFormat` function, see *CFML Reference*.

PART IV
Accessing and Using Data

This part describes how to access and use sources of data, including SQL (Structured Query Language) databases, LDAP (Lightweight Directory Access Protocol) directories, and Verity document collections. It provides an introduction to the SQL language, describes how to query and update SQL data sources, and how to use record sets and the ColdFusion query of queries mechanism to manipulate record sets. It also describes how to access and use LDAP directories, and how to index and search collections of documents and data sources using the Verity search engine.

The following chapters are included:

Chapter 19: Introduction to Databases and SQL 447
Chapter 20: Accessing and Retrieving Data 465
Chapter 21: Updating Your Database. 475
Chapter 22: Using Query of Queries 489
Chapter 23: Managing LDAP Directories 515
Chapter 24: Building a Search Interface 547
Chapter 25: Using Verity Search Expressions 581

CHAPTER 19
Introduction to Databases and SQL

Macromedia ColdFusion MX lets you create dynamic applications to access and modify data stored in a database. You do not need a thorough knowledge of databases to develop ColdFusion applications, but you must know some basic concepts and techniques. This chapter contains an overview of many important database and SQL concepts.

This chapter does not contain a complete description of database theory and SQL syntax. Each database server (such as SQL Server, Oracle, or DB2) has unique capabilities and properties. For more information, see the documentation that ships with your database server.

Contents

What is a database? 447

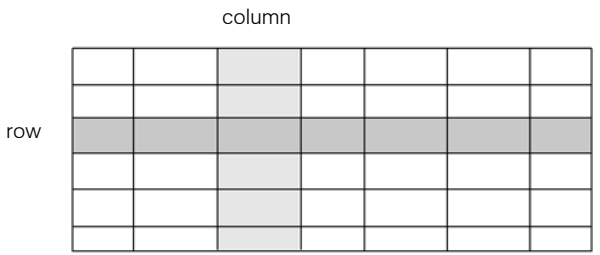
Using SQL. 452

Writing queries using an editor 460

What is a database?

A *database* defines a structure for storing information. Databases are typically organized into *tables*, which are collections of related items. You can think of a table as a grid of columns and rows. ColdFusion MX works primarily with relational databases, such as Oracle, DB2, and SQL Server.

The following figure shows the basic layout of a database table:



A *column* defines one piece of data stored in all rows of the table. A *row* contains one item from each column in the table.

For example, a table might contain the ID, name, title, and other information for individuals employed by a company. Each row, called a data *record*, corresponds to one employee. The value of a column within a record is referred to as a record *field*.

The following figure shows an example table, named employees, containing information about company employees:

EmpID	LastName	FirstName	Title	DeptID	Email	Phone
4	Smith	John	Engineer	3	jsmith	x5833

employees table

The record for employee 4 contains the following field values:

- LastName field is “Smith”
- FirstName field is “John”
- Title field is “Engineer”

This example uses the EmpID field as the table’s *primary key* field. The primary key contains a unique identifier to maintain each record's unique identity. Primary keys field can include an employee ID, part number, or customer number. Typically, you specify which column contains the primary key when you create a database table.

To access the table to read or modify table data, you use the SQL programming language. For example, the following SQL statement returns all rows from the table where the department ID is 3:

```
SELECT * FROM employees WHERE DEPTID=3
```

Note: In this chapter, SQL keywords and syntax are always represented by uppercase letters. Table and column names use mixed uppercase and lowercase letters.

Using multiple database tables

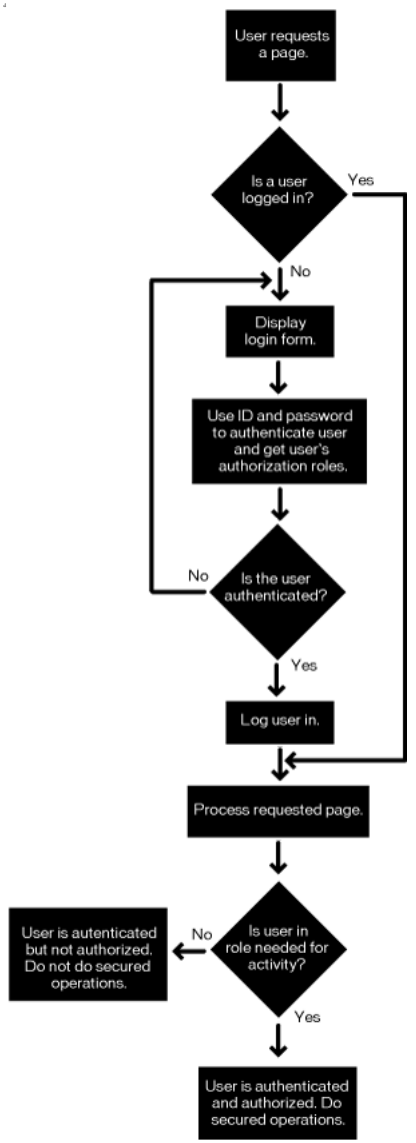
In many database designs, information is distributed to multiple tables. The following figure shows two tables, one for employee information and one for employee addresses:

EmpID	LastName	FirstName	Title	DeptID	Email	Phone
1	Jones	Joe	Engineer	3	jjones	x5844
2	Davis	Ken	Manager	4	kdavis	x5854
3	Baker	Mary	Engineer	3	mbaker	x5876
4	Smith	John	Engineer	3	jsmith	x5833
5	Morris	Jane	Manager	3	jmorris	x5812

employees table

EmpID	Street	City	State	Zip
1	4 Main St.	Newton	MA	02158
2	10 Oak Dr.	Newton	MA	02161
3	15 Main St.	Newton	MA	02158
4	56 Maple Ln.	Newton	MA	02160
5	25 Elm St.	Newton	MA	02160

addresses table



In this example, each table contains a column named EmpID. This column associates a row of the employees table with a row in the addresses table.

For example, to obtain all information about an employee, you request a row from the employees table and the row from the addresses table with the same value for EmpID.

One advantage of using multiple tables is that you can add tables containing new information without modifying the structure of your existing tables. For example, to add payroll information, you add a new table to the database where the first column contains the employee's ID and the columns contain current salary, previous salary, bonus payment, and 401(k) percent.

Also, an access to a small table is more efficient than an access to a large table. Therefore, if you update the street address of an employee, you update only the addresses table, without having to access any other table in the database.

Database permissions

In many database environments, a database administrator defines the access privileges for users accessing the database, usually through username and password. When a person attempts to connect to a database, the database ensures that the username and password are valid and then imposes access requirements on the user.

Privileges can restrict user access so that a user can do the following:

- Read data.
- Read data and add rows .
- Read data, add rows, modify existing tables.

In ColdFusion MX, you use the ColdFusion MX Administrator to define database connections, called *data sources*. As part of defining these connections, you specify the username and password used by ColdFusion to connect to the database. The database can then control access based on this username and password.

For more information on creating a data source, see *Configuring and Administering ColdFusion MX*.

Commits, rollbacks, and transactions

Before you access data stored in a database, it is important to understand several database concepts, including:

- Commit
- Rollback
- Transactions

A database *commit* occurs when you make a permanent change to a database. For example, when you write a new row to a database, the write does not occur until the database commits the change.

Rollback is the process of undoing a change to a database. For example, if you write a new row to a table, you can rollback the write up to the point where you commit the write. After the commit, you can no longer rollback the write.

Most databases support transactions where a transaction consists of one or more SQL statements. Within a transaction, your SQL statements can read, modify, and write a database. You end a transaction by either committing all your changes within the transaction or rolling back all of them.

Transactions can be useful when you have multiple writes to a database and want to make sure all writes occurred without error before committing them. In this case, you wrap all writes within a single transaction and check for errors after each write. If any write causes an error, you rollback all of them. If all writes occur successfully, you commit the transaction.

A bank might use a transaction to encapsulate a transfer from one account to another. For example, if you transfer money from your savings account to your checking account, you do not want the bank to debit the balance of your savings account unless it also credits your checking account. If the update to the checking account fails, the bank can rollback the debit of the savings account as part of the transaction.

ColdFusion MX includes the `cftransaction` tag that lets you implement database transactions for controlling rollback and commit. For more information, see *CFML Reference*.

Database design guidelines

From this basic description, the following database design rules emerge:

- Each record should contain a unique identifier as the primary key such as an employee ID, a part number, or a customer number. The primary key is typically the column used to maintain each record's unique identity among the tables in a relational database. Databases allow you to use multiple columns for the primary key.
- When you define a column, you define a SQL data type for the column, such as allowing only numeric values to be entered in the salary column.
- Assessing user needs and incorporating those needs in the database design is essential to a successful implementation. A well-designed database accommodates the changing data needs within an organization.

The best way to familiarize yourself with the capabilities of your database product or database management system (DBMS) is to review the product documentation.

Using SQL

This section introduces SQL, describes basic SQL syntax, and contains examples of SQL statements. It provides enough information for you to begin using ColdFusion MX. However, this section does not contain an exhaustive description of the entire SQL programming language. For complete SQL information, see the SQL reference that ships with your database.

A *query* is a request to a database. The query can ask for information from the database, write new data to the database, update existing information in the database, or delete records from the database.

Structured Query Language (SQL) is an ANSI/ISO standard programming language for writing database queries. All databases supported by ColdFusion support SQL, and all ColdFusion tags that access a database allow you to pass SQL statements to the tag.

SQL example

The most commonly used SQL statement in ColdFusion is the SELECT statement. The SELECT statement reads data from a database and returns it to ColdFusion. For example, the following SQL statement reads all the records from the employees table:

```
SELECT * FROM employees
```

You interpret this statement as "Select all rows from the table employees" where the wildcard symbol (*) corresponds to all columns.

Tip: If you are using Macromedia Dreamweaver MX 2004 or Macromedia HomeSite+, you can use the built-in query builder to build SQL statements graphically by selecting the tables and records to retrieve. For more information, see "Writing queries using an editor" on page 460.

In many cases, you do not want all rows from a table, but only a subset of rows. The next example returns all rows from the employees table, where the value of the DeptID column for the row is 3:

```
SELECT * FROM employees WHERE DeptID=3
```

You interpret this statement as "Select all rows from the table employees where the DeptID is 3".

SQL also lets you specify the table columns to return. For example, instead of returning all columns in the table, you can return a subset of columns:

```
SELECT LastName, FirstName FROM employees WHERE DeptID=3
```

You interpret this statement as "Select the columns FirstName and LastName from the table employees where the DeptID is 3".

In addition to with reading data from a table, you can write data to a table using the SQL INSERT statement. The following statement adds a new row to the employees table:

```
INSERT INTO employees(EmpID, LastName, Firstname)
  VALUES(51, 'Doe', 'John')
```

Basic SQL syntax elements

The following sections briefly describe the main SQL command elements.

Statements

A SQL statement always begins with a SQL verb. The following keywords identify commonly used SQL verbs:

Keyword	Description
SELECT	Retrieves the specified records.
INSERT	Adds a new row.
UPDATE	Changes values in the specified rows.
DELETE	Removes the specified rows.

Statement clauses

Use the following keywords to refine SQL statements:

Keyword	Description
FROM	Names the data tables for the operation.
WHERE	Sets one or more conditions for the operation.
ORDER BY	Sorts the result set in the specified order.
GROUP BY	Groups the result set by the specified select list items.

Operators

The following basic operators specify conditions and perform logical and numeric functions:

Operator	Description
AND	Both conditions must be met
OR	At least one condition must be met
NOT	Exclude the condition following
LIKE	Matches with a pattern
IN	Matches with a list of values
BETWEEN	Matches with a range of values
=	Equal to
<>	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
+	Addition
-	Subtraction
/	Division
*	Multiplication

Case sensitivity with databases

ColdFusion MX is a case-insensitive programming environment. *Case insensitivity* means the following statements are equivalent:

```
<cfset foo="bar">
<CFSET FOO="BAR">
<CfSet FOO="bar">
```

However, many databases, especially UNIX databases, are case-sensitive. *Case sensitivity* means that you must match exactly the case of all column and table names in SQL queries.

For example, the following queries are not equivalent in a case-sensitive database:

```
SELECT LastName FROM EMPLOYEES
SELECT LASTNAME FROM employees
```

In a case-sensitive database, employees and EMPLOYEES are two different tables.

For information on how your database handles case, see the product documentation.

SQL notes and considerations

When writing SQL in ColdFusion, keep the following guidelines in mind:

- There is a lot more to SQL than what is covered here. It is a good idea to purchase one or several SQL guides for reference.
- The data source, columns, and tables that you reference must exist in order to perform a successful query.
- Some DBMS vendors use nonstandard SQL syntax (known as a dialect) in their products. ColdFusion does not validate the SQL; it is passed on to the database for validation, so you are free to use any syntax that is supported by your database. Check your DBMS documentation for nonstandard SQL usage.

Reading data from a database

You use the SQL SELECT statement to read data from a database. The SQL statement has the following general syntax:

```
SELECT column_names
  FROM table_names
  [ WHERE search_condition ]
  [ GROUP BY group_expression ] [HAVING condition]
  [ ORDER BY order_condition [ ASC | DESC ] ]
```

The statements in square brackets are optional.

Note: There are additional options to SELECT depending on your database. For a complete syntax description for SELECT, see the product documentation.

This section describes options to the SELECT statement.

Results of a SELECT statement

When the database processes a SELECT statement, it returns a *record set* containing the requested data. The format of a record set is a table with rows and columns. For example, if you write the following query:

```
SELECT * FROM employees WHERE DeptID=3
```

The query returns the following table:

EmpID	LastName	FirstName	Title	DeptID	Email	Phone
				3		
				3		
				3		
				3		
				3		

Since the data returned to ColdFusion MX by a SELECT statement is in the form of a database table, ColdFusion MX lets you write a SQL query on the returned results. This functionality is called *query of queries*. For more information on query of queries, see Chapter 20, “Accessing and Retrieving Data,” on page 465.

The next example uses a SELECT statement to return only a specific set of columns from a table:

```
SELECT LastName, FirstName FROM employees WHERE DeptID=3
```

The query returns the following table:

LastName	FirstName

Filtering results

The SELECT statement lets you filter the results of a query to return only those records that meet specific criteria. For example, if you want to access all database records for employees in department 3, you use the following query:

```
SELECT * FROM employees WHERE DeptID=3
```

You can combine multiple conditions using the WHERE clause. For example, the following example uses two conditions:

```
SELECT * FROM employees WHERE DeptID=3 AND Title='Engineer'
```

Sorting results

By default, a database does not sort the records returned from a SQL query. In fact, you cannot guarantee that the records returned from the same query are returned in the same order each time you run the query.

However, if you require records in a specific order, you can write your SQL statement to sort the records returned from the database. To do so, you include an ORDER BY clause in the SQL statement.

For example, the following SQL statement returns the records of the table ordered by the LastName column:

```
SELECT * FROM employees ORDER BY LastName
```

You can combine multiple fields in the ORDER BY clause to perform additional sorting:

```
SELECT * FROM employees ORDER BY DepartmentID, LastName
```

This statement returns row ordered by department, then by last name within the department.

Returning a subset of columns

You might want only a subset of columns returned from a database table, as in the following example, which returns only the FirstName, LastName, and Phone columns. This example is useful if you are building a web page that shows the phone numbers for all employees.

```
SELECT FirstName, LastName, Phone FROM employees
```

However, this query does not to return the table rows in alphabetical order. You can include an ORDER clause in the SQL, as follows:

```
SELECT the FirstName, LastName, Phone
  FROM employees
  ORDER BY LastName, FirstName
```

Using column aliases

You might have column names that you do not want to retain in the results of your SQL statement. For example, your database is set up with a column that uses a reserved word in ColdFusion, such as EQ. In this case, you can rename the column as part of the query, as follows:

```
SELECT EmpID, LastName, EQ as MyEQ FROM employees
```

The results returned by this query contains columns named EmpID, LastName, and MyEQ.

Accessing multiple tables

In a database, you can have multiple tables containing related information. You can extract information from multiple tables as part of a query. In this case, you specify multiple table names in the SELECT statement, as follows:

```
SELECT LastName, FirstName, Street, City, State, Zip
  FROM employees, addresses
  WHERE employees.EmpID = addresses.EmpID
  ORDER BY LastName, FirstName
```

This SELECT statement uses the EmpID field to connect the two tables. This query prefixes the EmpID column with the table name. This is necessary because each table has a column named EmpID. You must prefix a column name with its table name if the column name appears in multiple tables.

In this case, you extract LastName and FirstName information from the employees table and Street, City, State, and Zip information from the addresses table. You can use output such as this is to generate mailing addresses for an employee newsletter.

The results of a SELECT statement that references multiple tables is a single result table containing a join of the information from corresponding rows. A *join* means information from two or more rows is combined to form a single row of the result. In this case, the resultant record set has the following structure:

LastName	FirstName	Street	City	State	Zip

What is interesting in this result is that even though you used the EmpID field to combine information from the two tables, you did not include that field in the output.

Modifying a database

You can use SQL to modify a database in the following ways:

- Inserting data into a database
- Updating data in a database
- Deleting data from a database
- Updating multiple tables

The following sections describe these modifications.

Inserting data into a database

You use SQL INSERT statement to write information to a database. A write adds a new row to a database table. The basic syntax of an INSERT statement is as follows:

```
INSERT INTO table_name(column_names)
  VALUES(value_list)
```

where:

- *column_names* specifies a comma-separated list of columns.
- *value_list* specifies a comma-separated list of values. The order of values has to correspond to the order that you specified column names.

Note: There are additional options to INSERT depending on your database. For a complete syntax description for INSERT, see the product documentation.

For example, the following SQL statement adds a new row to the employees table:

```
INSERT INTO employees(EmpID, LastName, Firstname)
  VALUES(51, 'Smith', 'John')
```

This statement creates a new row in the employees table and sets the values of the EmpID, LastName, and FirstName fields of the row. The remaining fields in the row are set to Null. *Null* means the field does not contain a value.

When you, or your database administrator, creates a table, you can set properties on the table and the columns of the table. One of the properties you can set for a column is whether the field supports Null values. If a field supports Nulls, you can omit the field from the INSERT statement. The database automatically sets the field to Null when you insert a new row.

However, if the field does not support Nulls, you must specify a value for the field as part of the INSERT statement; otherwise, the database issues an error.

The LastName and FirstName values in the query are contained within single-quotation marks. This is necessary because the table columns are defined to contain character strings. Numeric data does not require the quotation marks.

Updating data in a database

Use the UPDATE statement in SQL to update the values of a table row. Update lets you update the fields of a specific row or all rows in the table. The UPDATE statement has the following syntax:

```
UPDATE table_name
  SET column_name1=value1, ... , column_nameN=valueN
  [ WHERE search_condition ]
```

Note: There are additional options to UPDATE depending on your database. For a complete syntax description for UPDATE, see the product documentation.

You should not attempt to update a record's primary key field. Your database typically enforces this restriction.

The UPDATE statement uses the optional WHERE clause, much like the SELECT statement, to determine which table rows to modify. The following UPDATE statement updates the e-mail address of John Smith:

```
UPDATE employees SET Email='jsmith@mycompany.com' WHERE EmpID = 51
```

Be careful using UPDATE. If you omit the WHERE clause to execute the following statement:

```
UPDATE employees SET Email = 'jsmith@mycompany.com'
```

you update the Email field for all rows in the table.

Deleting data from a database

The DELETE statement removes rows from a table. The DELETE statement has the following syntax:

```
DELETE FROM table_name
  [ WHERE search_condition ]
```

Note: There are additional options to DELETE depending on your database. For a complete syntax description for DELETE, see the product documentation.

You can remove all rows from a table using a statement in the form:

```
DELETE FROM employees
```

Typically, you specify a WHERE clause to the DELETE statement to delete specific rows of the table. For example, the following statement deletes John Smith from the table:

```
DELETE FROM employees WHERE EmpID=51
```

Updating multiple tables

The examples in this section all describe how to modify a single database table. However, you might have a database that uses multiple tables to represent information.

One way to update multiple tables is to use one INSERT statement per table and to wrap all INSERT statements within a database transaction. A transaction contains one or more SQL statements that can be rolled back or committed as a unit. If any single statement in the transaction fails, you can roll back the entire transaction, cancelling any previous writes that occurred within the transaction. You can use the same technique for selects, updates, and deletes. The following example uses the `cftransaction` tag to wrap multiple SQL statements:

```
<cftransaction>

<cfquery name="qInsEmp" datasource="cfdocexamples">
INSERT INTO Employees (FirstName,LastName,EMail,Phone,Department)
VALUES ('Simon', 'Horwith', 'SHORWITH','(202)-797-6570','Research and
  Development')
</cfquery>

<cfquery name="qGetID" datasource="cfdocexamples">
SELECT MAX(Emp_ID) AS New_Employee
FROM Employees
</cfquery>

</cftransaction>
```

Writing queries using an editor

Dreamweaver MX 2004 and HomeSite+ provide a graphical user interface (GUI) for writing and executing queries. A GUI is useful for developing and testing your queries before you insert them into a ColdFusion application.

This section contains a brief description of these GUIs. For more information, see the documentation on your specific tool.

Writing queries using Dreamweaver MX 2004

This section describes how to define a query using the Dreamweaver MX 2004 Recordset dialog box, which allows you to create a record set without having to manually enter SQL statements. Defining a record set using this method can be as easy as selecting a database connection and table from the pop-up menus.

To define a record set without writing SQL:

1. In the Dreamweaver Document window, open the page that will use the record set.
2. To open the Data Bindings panel, select Window > Data Bindings.

3. In the Data Bindings panel, click the Plus (+) button and choose Recordset (Query) from the pop-up menu.

   The Simple Recordset dialog box appears:



4. Complete the dialog box.
5. Click the Test button to execute the query and ensure that it retrieves the information you intended.

   If you defined a filter that uses parameters input by users, the Test button displays the Test Value dialog box. Enter a value in the Test Value text box and click OK. If an instance of the record set is successfully created, a table displaying the data from your record set appears.
6. Click OK to add the record set to the list of available content sources in the Data bindings panel.

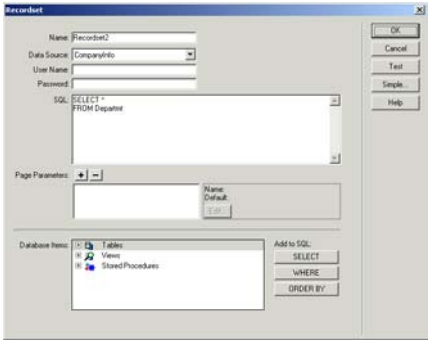
If you prefer to write your own SQL statements, or need to create more complex queries then the Simple Recordset dialog box allows, you can define record sets using the Advanced Recordset dialog box

Creating an advanced record set by writing SQL:

1. In the Dreamweaver Document window, open the page that will use the record set.
2. Select Windows > Data Bindings to display the Data Bindings panel.
3. In the Data Bindings panel, click the Plus (+) button and select Recordset (Query) from the pop-up menu.

   If the Simple Recordset dialog box appears, switch to the Advanced Recordset dialog box by clicking the Advanced button.

   The Advanced Recordset dialog box appears:



4. Complete the dialog box.
5. Click the Test button to execute the query and ensure that it retrieves the information you intended.

   If you defined a filter that uses parameters input by users, the Test button displays the Test Value dialog box. Enter a value in the Test Value text field and click OK. If an instance of the record set is successfully created, a table displaying the data from your record set appears.
6. Click OK to add the record set to the list of available content sources in the Data Bindings panel.

Writing queries using Macromedia HomeSite+

Macromedia HomeSite+ includes the combined features of HomeSite 5 and ColdFusion Studio 5, with additional support for new ColdFusion MX tags. HomeSite+ supports SQL Builder for writing queries.

SQL Builder is a powerful visual tool for building, testing, and saving SQL statements for use in application data queries. You can copy completed SQL code blocks directly into your ColdFusion applications.

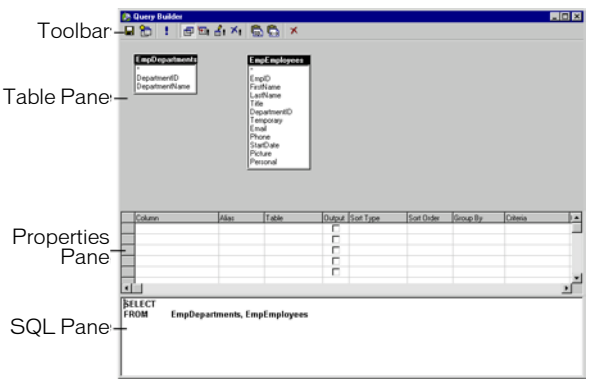
To open SQL Builder:

Do one of the following:

- Select Tools > SQL Builder from the HomeSite+ menu, select an RDS server, select a database from the drop-down list, and click New Query.
- In the Database tab, select an RDS server, right-click a database name or a table, and select New Query.
- Open the `cfquery` tag editor, select an RDS server, and click New Query.

The SQL Builder interface

The following figure shows the SQL Builder interface:



The SQL Builder is divided into the following four sections:

Section	Use
Toolbar	Contains buttons for SQL keywords and commands.
Table pane	Provides a view of the tables in your query and allows you to create joins between tables.
Properties pane	Allows you to set the properties of the query, such as the columns that are being selected or the columns that are being updated.
SQL pane	Shows you the SQL statement as it is built. The SQL pane does not support reverse editing, so any changes you make in this pane will not be made to the query in the Properties pane or the Table pane.

Writing SQL statements

SQL Builder opens a SELECT statement by default, since this is the most common type of query. SQL Builder supports the following four types of SQL statements:

- Select (default)
- Insert
- Update
- Delete

CHAPTER 20
Accessing and Retrieving Data

This chapter describes how to retrieve data from a database and work with query data. This chapter also shows how to use the `cfquery` tag to query a data source, and use the `cfoutput` tag to output the query results to a web page. This chapter also describes how to use the `cfqueryparam` tag to help reduce security risks.

Contents

Working with dynamic data 465

Retrieving data 466

Outputting query data 469

Getting information about query results 471

Enhancing security with cfqueryparam 472

Working with dynamic data

A web application page is different from a static web page because it can publish data dynamically. This can involve querying databases, connecting to LDAP or mail servers, and leveraging COM, DCOM, CORBA, or Java objects to retrieve, update, insert, and delete data at runtime—as your users interact with pages in their browsers.

For ColdFusion developers, the term data source can refer to a number of different types of structured content accessible locally or across a network. You can query websites, LDAP servers, POP mail servers, and documents in a variety of formats. Most commonly though, a database drives your applications, and for this discussion a *data source* means the entry point from ColdFusion to a database.

In this chapter, you build a query to retrieve data from the cfdocexamples data source. In Windows, this data source connects to a Microsoft Access database (cfdocexamples.mdb). In UNIX, this data source connects to a PointBase database. In subsequent chapters in this manual, you insert and update data in this database.

To query a database, you must use:

- ColdFusion data sources
- The `cfquery` tag
- SQL commands

Retrieving data

You can query databases to retrieve data at runtime. The retrieved data, called the *record set*, is stored on that page as a query object. A *query object* is a special entity that contains the record set values, plus RecordCount, CurrentRow, ColumnList, SQL, Cached, and SQLParameter query variables. You specify the query object's name in the `name` attribute of the `cfquery` tag. The query object is often called simply *the query*.

The following is a simple `cfquery` tag:

```
<cfquery name = "GetSals" datasource = "cfdocexamples">
  SELECT * FROM Employee
  ORDER BY LastName
</cfquery>
```

Note: The terms "record set" and "query object" are often used synonymously when discussing record sets for queries. For more information, see Chapter 22, "Using Query of Queries," on page 489.

When retrieving data from a database, perform the following tasks:

- To tell ColdFusion how to connect to a database, use the `cfquery` tag on a page.
- To specify the data that you want to retrieve from the database, write SQL commands inside the `cfquery` block.
- Reference the query object and use its data values in any tag that presents data, such as `cfoutput`, `cfgrid`, `cftable`, `cfgraph`, or `cftree`.

The cfquery tag

The `cfquery` tag is one of the most frequently used CFML tags. You use it to retrieve and reference the data returned from a query. When ColdFusion encounters a `cfquery` tag on a page, it does the following:

- Connects to the specified data source.
- Performs SQL commands that are enclosed within the block.
- Returns result set values to the page in a query object.

The cfquery tag syntax

The following code shows the syntax for the `cfquery` tag:

```
<cfquery name="EmpList" datasource="cfdocexamples">
  SQL code...
</cfquery>
```

In this example, the query code tells ColdFusion to do the following:

- Connect to the cfdocexamples data source (the cfdocexamples.mdb database).
- Execute SQL code that you specify.
- Store the retrieved data in the query object EmpList.

When creating queries to retrieve data, keep the following guidelines in mind:

- You must use opening `<cfquery>` and ending `</cfquery>` tags, because the `cfquery` tag is a block tag.
- Enter the query `name` and `datasource` attributes within the opening `cfquery` tag.
- To tell the database what to process during the query, place SQL statements inside the `cfquery` block.
- When referencing text literals in SQL, use single-quotation marks ('). For example, `SELECT * FROM mytable WHERE FirstName='Jacob'` selects every record from mytable in which the first name is Jacob.
- Surround attribute values with double quotation marks ("attrib_value").
- Make sure that a data source exists in the ColdFusion MX Administrator before you reference it in a `cfquery` tag.
- Columns and tables that you refer to in your SQL statement must exist, otherwise the query will fail.
- Reference the query data by naming the query in one of the presentation tags, such as `cfoutput`, `cfgrid`, `cftable`, `cfgraph`, or `cftree`.
- When ColdFusion returns database columns, it removes table and owner prefixes. For example, if you query Employee.Emp_ID in the query, the Employee, is removed and returns as Emp_ID. You can use an alias to handle duplicate column names; for more information, see Chapter 22, "Using Query of Queries," on page 489.
- You cannot use SQL reserved words, such as MIN, MAX, COUNT, in a SQL statement. Because reserved words are database-dependent, see your database's documentation for a list of reserved words.

Building queries

As discussed earlier in this chapter, you build queries using the `cfquery` tag and SQL.

Note: This and many subsequent procedures use the cfdocexamples data source that connects to the cfdocexamples.mdb database. This data source is installed by default. For information on adding or configuring a data source, see *Configuring and Administering ColdFusion MX*.

To query the table:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>Employee List</title>
</head>
<body>
<h1>Employee List</h1>
<cfquery name="EmpList" datasource="cfdocexamples">
  SELECT FirstName, LastName, Salary, Contract
  FROM Employee
</cfquery>
</body>
</html>
```

   Note: Macromedia recommends that you create structured, reusable code by putting queries in ColdFusion components; however, for simplicity, the examples in this chapter include the query in the body of the ColdFusion page. For more information about using ColdFusion components, see Chapter 10, "Building and Using ColdFusion Components," on page 201.

2. Save the page as emplist.cfm in the myapps directory under your *web_root* directory. For example, the default path on a Windows computer would be:

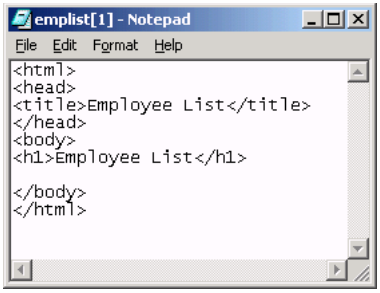
   C:\CFusionMX7\wwwroot\myapps\

3. Enter the following URL in your web browser:

   http://localhost/myapps/emplist.cfm

   Only the header appears.

4. View the source in the browser:



   ColdFusion creates the EmpList data set, but only HTML and text return to the browser. When you view the page's source, you see only HTML tags and the heading "Employee List." To display the data set on the page, you must code tags and variables to output the data.

Reviewing the code

The query you just created retrieves data from the cfdocexamples database. The following table describes the highlighted code and its function:

Code	Description
`<cfquery name="EmpList" datasource="cfdocexamples">`	Queries the database specified in the cfdocexamples data source.
`SELECT FirstName, LastName, Salary, Contract FROM Employee`	Gets information from the FirstName, LastName, Salary, and Contract fields in the Employee table.
`</cfquery>`	Ends the `cfquery` block.

Outputting query data

After you define a query, you can use the `cfoutput` tag with the `query` attribute to output data from the record set. When you use the `query` attribute, keep the following in mind:

- ColdFusion loops through all the code contained within the `cfoutput` block, once for each row in the record set returned from the database.
- You must reference specific column names within the `cfoutput` block to output the data to the page.
- You can place text, CFML tags, and HTML tags inside or surrounding the `cfoutput` block to format the data on the page.
- Although you do not have to specify the query name when you refer to a query column, you should use the query name as a prefix for best practices reasons. For example, if you specify the Emplist query in your `cfoutput` tag, you can refer to the Firstname column in the Emplist query as Firstname. However, using the query name as a prefix—Emplist.Firstname— is preferred, and is in the following procedure.

The `cfoutput` tag accepts a variety of optional attributes but, ordinarily, you use the `query` attribute to define the name of an existing query.

To output query data on your page:

1. Edit emplist.cfm so that it appears as follows:

```
<html>
<head>
<title>Employee List</title>
</head>
<body>
<h1>Employee List</h1>
<cfquery name="EmpList" datasource="cfdocexamples">
  SELECT FirstName, LastName, Salary, Contract
  FROM Employee
</cfquery>
<cfoutput query="EmpList">
#EmpList.FirstName#, #EmpList.LastName#, #EmpList.Salary#,
  #EmpList.Contract#<br>
```

```
</cfoutput>
</body>
</html>
```

2. Save the file and view it in your web browser:

   A list of employees appears in the browser, with each line displaying one row of data.

   Note: You might need to refresh your browser to see your changes.

You created a ColdFusion application page that retrieves and displays data from a database. At present, the output is raw and needs formatting. For more information, see "Introduction to Retrieving and Formatting Data" on page 609.

Reviewing the code

The results of the query appear on the page. The following table describes the highlighted code and its function:

Code	Description
`<cfoutput query="EmpList">`	Displays information retrieved in the EmpList query.

`#EmpList.FirstName#,`
`#EmpList.LastName#,`
`#EmpList.Salary#,`
`#EmpList.Contract#`
	Displays the value of the `FirstName`, `LastName`, `Salary`, and `Contract` fields of each record, separated by commas and spaces.

` `	Inserts a line break (go to the next line) after each record.
`</cfoutput>`	Ends the `cfoutput` block.

Query output notes and considerations

When outputting query results, keep the following guidelines in mind:

- A `cfquery` must retrieve data before the `cfoutput` tag can display its results. Although you can include both on the same page, Macromedia recommends that you put queries in ColdFusion components and output the results on a separate page. For more information, see Chapter 10, "Building and Using ColdFusion Components," on page 201.
- To output data from all the records of a query, specify the query name by using the `query` attribute in the `cfoutput` tag.
- Columns must exist and be retrieved to the application to output their values.
- Inside a `cfoutput` block that uses a `cfquery` attribute, you can prefix the query variables with the name of the query; for example, `Emplist.FirstName`.
- As with other attributes, surround the `query` attribute value with double-quotation marks (").
- As with any variables that you reference for output, surround column names with number signs (#) to tell ColdFusion to output the column's current values.
- Add a `<br>` tag to the end of the variable references so that ColdFusion starts a new line for each row that the query returns.

Getting information about query results

Each time you query a database with the `cfquery` tag, you get the data (the record set) and the query variables; together these comprise the query object. The following table describes the query variables, which are sometimes referred to as query properties:

Variable	Description
RecordCount	The total number of records returned by the query.
ColumnList	A comma-delimited list of the query columns, in alphabetical order.
SQL	The SQL statement executed.
Cached	Whether the query was cached.
SQLParameters	Ordered array of `cfqueryparam` values.
ExecutionTime	Cumulative time required to process the query, in milliseconds.

In your CFML code, you can use these variables as if they were columns in a database table. Use the `result` attribute to specify the name of the structure that ColdFusion populates with these variables. You then use that structure name to refer to the query variables as illustrated in the following example:

To output information about the query on your page:

1. Edit emplist.cfm so that it appears as follows:

```
<cfset Emp_ID = 1>
<cfquery name="EmpList" datasource="cfdocexamples" result="tmpResult">
  SELECT FirstName, LastName, Salary, Contract
  FROM Employee
  WHERE Emp_ID = <cfqueryPARAM value = "#Emp_ID#"
           CFSQLType = "CF_SQL_INTEGER">
</cfquery>
<cfoutput query="EmpList">
  #EmpList.FirstName#, #EmpList.LastName#, #EmpList.Salary#,
  #EmpList.Contract#<br>
</cfoutput> <br>
<cfoutput>
  The query returned #tmpResult.RecordCount# records.<br>
  The query columns are:  #tmpResult.ColumnList#.<br>
  The SQL is #tmpResult.SQL#.<br>
  Whether the query was cached: #tmpResult.Cached#.<br>
  Query execution time: #tmpResult.ExecutionTime#.<br>
</cfoutput>
<cfdump var="#tmpResult.SQLParameters#">
```

2. Save the file and view it in your web browser:

   The number of employees now appears below the list of employees. You might have to refresh your browser and scroll to see the RecordCount output.

Reviewing the code

You now display the number of records retrieved in the query. The following table describes the code and its function:

Code	Description
`<cfoutput>`	Displays what follows.
`The query returned`	Displays the text "The query returned".
`#EmpList.RecordCount#`	Displays the number of records retrieved in the EmpList query.
`records.`	Displays the text "records."
`</cfoutput>`	Ends the `cfoutput` block.

Query variable notes and considerations

When using query variables, keep the following guidelines in mind:

- Reference the query variable within a `cfoutput` block so that ColdFusion outputs the query variable value to the page.
- Surround the query variable reference with number signs (#) so that ColdFusion knows to replace the variable name with its current value.
- Do not use the `cfoutput` tag `query` attribute when you output the `RecordCount` or `ColumnList` property. If you do, you get one copy of the output for each row. Instead, prefix the variable with the name of the query.

Enhancing security with cfqueryparam

Some DBMSs let you send multiple SQL statements in a single query. However, hackers might try to modify URL or form variables in a dynamic query by appending malicious SQL statements to existing parameters. Be aware that there are potential security risks when you pass parameters in a query string. This can happen in many development environments, including ColdFusion, ASP, and CGI. Using the `cfqueryparam` tag can reduce this risk.

About query string parameters

When you let a query string pass a parameter, ensure that only the expected information is passed. The following ColdFusion query contains a WHERE clause, which selects only database entries that match the last name specified in the LastName field of a form:

```
<cfquery name="GetEmployees" datasource="cfdocexamples">
  SELECT FirstName, LastName, Salary
  FROM Employee
  WHERE LastName='#Form.LastName#'
</cfquery>
```

Someone could call this page with the following malicious URL:

http://myserver/page.cfm?Emp_ID=7%20DELETE%20FROM%20Employee

The result is that ColdFusion tries to execute the following query:

```
<cfquery name="GetEmployees" datasource="cfdocexamples">
  SELECT * FROM Employee
  WHERE Emp_ID = 7 DELETE FROM Employee
</cfquery>
```

In addition to an expected integer for the Emp_ID column, this query also passes malicious string code in the form of a SQL statement. If this query successfully executes, it deletes all rows from the Employee table—something you definitely do not want to enable by this method. To prevent such actions, you must evaluate the contents of query string parameters.

Using cfqueryparam

You can use the `cfqueryparam` tag to evaluate query string parameters and pass a ColdFusion variable within a SQL statement. This tag evaluates variable values before they reach the database. You specify the data type of the corresponding database column in the `cfsqltype` attribute of the `cfqueryparam` tag. In the following example, because the Emp_ID column in the cfdocexamples data source is an integer, you specify a `cfsqltype` of `cf_sql_integer`:

```
<cfquery name="EmpList" datasource="cfdocexamples">
  SELECT * FROM Employee
  WHERE Emp_ID = <cfqueryparam value = "#Emp_ID#"
                 cfsqltype = "cf_sql_integer">
</cfquery>
```

The `cfqueryparam` tag checks that the value of Emp_ID is an integer data type. If anything else in the query string is not an integer, such as a SQL statement to delete a table, the `cfquery` tag does not execute. Instead, the `cfqueryparam` tag returns the following error message:

Invalid data '7 DELETE FROM Employee' for CFSQLTYPE 'CF_SQL_INTEGER'.

Using cfqueryparam with strings

When passing a variable that contains a string to a query, specify a `cfsqltype` value of `cf_sql_char`, and specify the `maxLength` attribute, as in the following example:

```
<cfquery name = "getFirst" dataSource = "cfdocexamples">
  SELECT * FROM employees
  WHERE LastName = <cfqueryparam value = "#LastName#"
                   cfsqltype = "cf_sql_char" maxLength = "17">
</cfquery>
```

In this case, `cfqueryparam` performs the following checks:

- It ensures that LastName contains a string.
- It ensures that the string is 17 characters or less.
- It escapes the string with single-quotation marks so that it appears as a single value to the database. Even if a hacker passes a bad URL, it appears as follows:

  WHERE LastName = 'Smith DELETE FROM MyCustomerTable'.

Using cfSqlType

The following table lists the available SQL types against which you can evaluate the `value` attribute of the `cfqueryparam` tag:

BIGINT	BIT	CHAR	DATE
DECIMAL	DOUBLE	FLOAT	IDSTAMP
INTEGER	LONGVARCHAR	MONEY	MONEY4
NUMERIC	REAL	REFCURSOR	SMALLINT
TIME	TIMESTAMP	TINYINT	VARCHAR

Note: Specifying the `cfsqltype` attribute causes the DBMS to use bind variables, which can greatly enhance performance.

CHAPTER 21
Updating Your Database

This chapter describes how to use ColdFusion to insert, update, and delete information in a database.

Contents

About updating your database 475
Inserting data 475
Updating data 480
Deleting data 487

About updating your database

ColdFusion was originally developed as a way to readily interact with databases. You can quickly insert, update, and delete the contents of your database by using ColdFusion forms, which are typically a pair of pages. One page displays the form with which your end user will enter values; the other page performs the action (insert, update or delete).

Depending on the extent and type of data manipulation, you can use CFML with or without SQL commands. If you use SQL commands, ColdFusion requires a minimal amount of SQL knowledge.

Inserting data

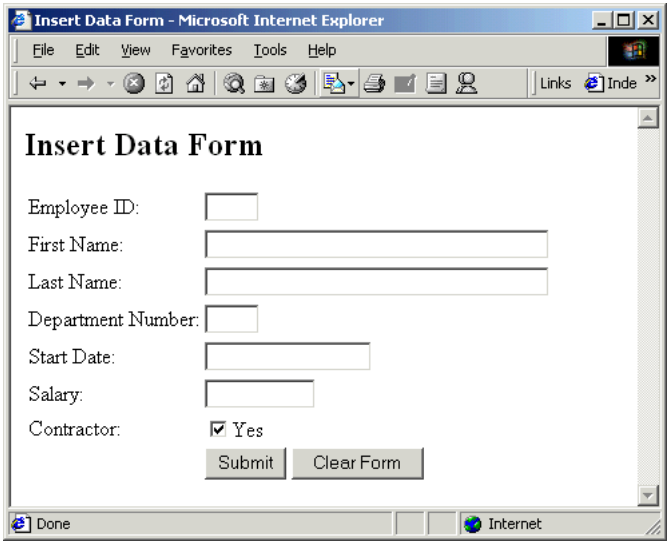
You usually use two application pages to insert data into a database:

- An insert form
- An insert action page

You can create an insert form with standard HTML form tags or with `cfform` tags (see "Creating custom forms with the cfform tag" on page 631). When the user submits the form, form variables are passed to a ColdFusion action page that performs an insert operation (and whatever else is called for) on the specified data source. The insert action page can contain either a `cfinsert` tag or a `cfquery` tag with a SQL INSERT statement. The insert action page should also contain a confirmation message for the end user.

Creating an HTML insert form

The following procedure creates a form using standard HTML tags. The form looks like the following in your web browser:



To create an insert form:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>Insert Data Form</title>
</head>

<body>
<h2>Insert Data Form</h2>

<table>
<!--- begin html form;
put action page in the "action" attribute of the form tag --->
<form action="insert_action.cfm" method="post">
<tr>
  <td>Employee ID:</td>
  <td><input type="text" name="Emp_ID" size="4" maxlength="4"></td>
</tr>
<tr>
  <td>First Name:</td>
  <td><input type="Text" name="FirstName" size="35" maxlength="50"></td>
</tr>
<tr>
  <td>Last Name:</td>
  <td><input type="Text" name="LastName" size="35" maxlength="50"></td>
</tr>
<tr>
  <td>Department Number:</td>
```

```
  <td><input type="Text" name="Dept_ID" size="4" maxlength="4"></td>
</tr>
<tr>
  <td>Start Date:</td>
  <td><input type="Text" name="StartDate" size="16" maxlength="16"></td>
</tr>
<tr>
  <td>Salary:</td>
  <td><input type="Text" name="Salary" size="10" maxlength="10"></td>
</tr>
<tr>
  <td>Contractor:</td>
  <td><input type="checkbox" name="Contract" value="Yes" checked>Yes</td>
</tr>
<tr>
  <td> </td>
  <td><input type="Submit" value="Submit"> <input type="Reset"
value="Clear Form"></td>
</tr>
</form>
<!--- end html form --->
</table>

</body>
</html>
```

2. Save the file as insert_form.cfm in the myapps directory under your *web_root* and view it in your web browser.

Note: The form will not work until you write an action page for it. For more information, see "Creating an action page to insert data" on page 477.

Data entry form notes and considerations

If you use the `cfinsert` tag in the action page to insert the data into the database, you should follow these rules for creating the form page:

- You only need to create HTML form fields for the database columns into which you will insert data.
- By default, `cfinsert` inserts all of the form's fields into the database columns with the same names. For example, it puts the Form.Emp_ID value in the database Emp_ID column. The tag ignores form fields that lack corresponding database column names.

Note: You can also use the `formfields` attribute of the `cfinsert` tag to specify which fields to insert; for example, `formfields="prod_ID,Emp_ID,status"`.

Creating an action page to insert data

You can use the `cfinsert` tag or the `cfquery` tag to create an action page that inserts data into a database.

Creating an insert action page with cfinsert

The `cfinsert` tag is the easiest way to handle simple inserts from either a `cfform` or an HTML form. This tag inserts data from all the form fields with names that match database field names.

To create an insert action page with cfinsert:

1. Create a ColdFusion page with the following content:

```
<html>
<head> <title>Input form</title> </head>

<body>
<!--- If the Contractor check box is clear,
  set the value of the Form.Contract to "No" --->
<cfif not isdefined("Form.Contract")>
   <cfset Form.Contract = "No">
</cfif>

<!--- Insert the new record --->
<cfinsert datasource="cfdocexamples" tablename="Employee">

<h1>Employee Added</h1>
<cfoutput>You have added #Form.FirstName# #Form.Lastname# to the
       employee database.
</cfoutput>

</body>
</html>
```

2. Save the page as insert_action.cfm.
3. View insert_form.cfm in your web browser and enter values.

   Note: You might want to compare views of the Employee table in the cfdocexamples data source before and after inserting values in the form.

4. Click Submit.

   ColdFusion inserts your values into the Employee table and displays a confirmation message.

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfif not isdefined("Form.Contract")>`
`  <cfset Form.Contract = "No">`
`</cfif>`
	Sets the value of Form.Contract to No if it is not defined. If the Contractor check box is unchecked, no value is passed to the action page; however, the database field must have some value.

Code	Description
`<cfinsert datasource="cfdocexamples" tablename="Employee">`	Creates a new row in the Employee table of the cfdocexamples database. Inserts data from the form into the database fields with the same names as the form fields.
`<cfoutput>You have added #Form.FirstName# #Form.Lastname# to the employee database. </cfoutput>`	Informs the user that values were inserted into the database.

Note: If you use form variables in `cfinsert` or `cfupdate` tags, ColdFusion automatically validates any form data it sends to numeric, date, or time database columns. You can use the hidden field validation functions for these fields to display a custom error message. For more information, see Chapter 26, "Introduction to Retrieving and Formatting Data," on page 609.

Creating an insert action page with cfquery

For more complex inserts from a form submittal, you can use a SQL INSERT statement in a `cfquery` tag instead of using a `cfinsert` tag. The SQL INSERT statement is more flexible because you can insert information selectively or use functions within the statement.

The following procedure assumes that you have created the insert_action.cfm page, as described in "Creating an insert action page with cfinsert" on page 478.

To create an insert action page with cfquery:

1. In insert_action.cfm, replace the `cfinsert` tag with the following highlighted `cfquery` code:

```
<html>
<head>
  <title>Input form</title>
</head>

<body>
<!--- If the Contractor check box is clear),
  set the value of the Form.Contract to "No" --->
<cfif not isdefined("Form.Contract")>
   <cfset Form.Contract = "No">
</cfif>

<!--- Insert the new record --->
<cfquery name="AddEmployee" datasource="cfdocexamples">
  INSERT INTO Employee
  VALUES (#Form.Emp_ID#, '#Form.FirstName#',
  '#Form.LastName#', #Form.Dept_ID#,
  '#Form.StartDate#', #Form.Salary#, '#Form.Contract#')
</cfquery>

<h1>Employee Added</h1>
<cfoutput>You have added #Form.FirstName# #Form.Lastname# to the
       employee database.
</cfoutput>

</body>
</html>
```

2. Save the page.
3. View insert_form.cfm in your web browser and enter values.
4. Click Submit.

   ColdFusion inserts your values into the Employee table and displays a confirmation message.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description

`<cfquery name="AddEmployee" datasource="cfdocexamples">`
`INSERT INTO Employee`
`VALUES (#Form.Emp_ID#,`
`'#Form.FirstName#',`
`'#Form.LastName#',`
`#Form.Dept_ID#,`
`'#Form.StartDate#',`
`#Form.Salary#,`
`'#Form.Contract#')`
`</cfquery>`
	Inserts a new row into the Employee table of the cfdocexamples database. Specifies each form field to be added.
	Because you are inserting data into all database fields in the same left-to-right order as in the database, you do not have to specify the database field names in the query.
	Because #From.Emp_ID#, #Form.Dept_ID#, and #Form.Salary# are numeric, they do not need to be enclosed in quotation marks.

Inserting into specific fields

The preceding example inserts data into all the fields of a table (the Employee table has seven fields). There might be times when you do not want users to add data into all fields. To insert data into specific fields, the SQL statement in the `cfquery` must specify the field names following both INSERT INTO and VALUES. For example, the following `cfquery` omits salary and start date information from the update. Database values for these fields are 0 and NULL, respectively, according to the database's design.

```
<cfquery name="AddEmployee" datasource="cfdocexamples">
  INSERT INTO Employee
    (Emp_ID,FirstName,LastName,
    Dept_ID,Contract)
  VALUES
    (#Form.Emp_ID#,'#Form.FirstName#','#Form.LastName#',
    #Form.Dept_ID#,'#Form.Contract#')
</cfquery>
```

Updating data

You usually use the following two application pages to update data in a database:

- An update form
- An update action page

You can create an update form with `cfform` tags or HTML form tags. The update form calls an update action page, which can contain either a `cfupdate` tag or a `cfquery` tag with a SQL UPDATE statement. The update action page should also contain a confirmation message for the end user.

Creating an update form

The following are the key differences between an update form and an insert form:

- An update form contains a reference to the primary key of the record that is being updated.

  A *primary key* is a field(s) in a database table that uniquely identifies each record. For example, in a table of employee names and addresses, only the Emp_ID is unique to each record.
- An update form is usually populated with existing record data.

The easiest way to designate the primary key in an update form is to include a hidden input field with the value of the primary key for the record you want to update. The hidden field indicates to ColdFusion which record to update.

To create an update form:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>Update Form</title>
</head>

<body>
<cfquery name="GetRecordtoUpdate"
  datasource="cfdocexamples">
  SELECT * FROM Employee
  WHERE Emp_ID = #URL.Emp_ID#
</cfquery>


<cfoutput query="GetRecordtoUpdate">
<table>
<form action="update_action.cfm" method="Post">
  <input type="Hidden" name="Emp_ID"
    value="#Emp_ID#"><br>
<tr>
  <td>First Name:</td>
  <td><input type="text" name="FirstName" value="#FirstName#"></td>
</tr>
<tr>
  <td>Last Name:</td>
  <td><input type="text" name="LastName" value="#LastName#"></td>
</tr>
<tr>
  <td>Department Number:</td>
  <td><input type="text" name="Dept_ID" value="#Dept_ID#"></td>
</tr>
<tr>
  <td>Start Date:</td>
  <td><input type="text" name="StartDate" value="#StartDate#"></td>
</tr>
<tr>
  <td>Salary:</td>
  <td><input type="text" name="Salary" value="#Salary#"></td>
</tr>
```

```
<tr>
  <td>Contractor:</td>
  <td><cfif #Contract# IS "Yes">
    <input type="checkbox" name="Contract" checked>Yes
  <cfelse>
    <input type="checkbox" name="Contract">Yes
  </cfif></td>
</tr>
<tr>
  <td> </td>
  <td><input type="Submit" value="Update Information"></td>
</tr>
</form>
</table>
</cfoutput>

</body>
</html>
```

2. Save the file as update_form.cfm.
3. View update_form.cfm in your web browser by specifying the page URL and an Employee ID; for example, enter the following:

   http://localhost/myapps/update_form.cfm?Emp_ID=3

   Note: Although you can view an employee's information, you must code an action page before you can update the database. For more information, see "Creating an action page to update data" on page 483.

Reviewing the code

The following table describes the code and its function:

Code	Description
`<cfquery name="GetRecordtoUpdate" datasource="cfdocexamples"> SELECT * FROM Employee WHERE Emp_ID = #URL.Emp_ID# </cfquery>`	Queries the cfdocexamples data source and returns records in which the employee ID matches what was entered in the URL that called this page.
`<cfoutput query="GetRecordtoUpdate"> ... </cfoutput>`	Makes available as variables the results of the GetRecordtoUpdate query in the form created in subsequent lines.
`<form action="update_action.cfm" method="Post"> ... </form>`	Creates a form whose variables will be processed on the update_action.cfm action page.
`<input type="Hidden" name="Emp_ID" value="#Emp_ID#"> `	Uses a hidden input field to pass the Emp_ID (primary key) value to the action page.

Code	Description
First Name: <input type="text" name="FirstName" value="#FirstName#"> Last Name: <input type="text" name="LastName" value="#LastName#"> Department Number: <input type="text" name="Dept_ID" value="#Dept_ID#"> Start Date: <input type="text" name="StartDate" value="#StartDate#"> Salary: <input type="text" name="Salary" value="#Salary#"> 	Populates the fields of the update form. This example does not use ColdFusion formatting functions. As a result, start dates look like 1985-03-12 00:00:00 and salaries do not have dollar signs or commas. The user can replace the information in any field using any valid input format for the data.
Contractor: <cfif #Contract# IS "Yes"> <input type="checkbox" name="Contract" checked>Yes <cfelse> <input type="checkbox" name="Contract"> Yes </cfif> <input type="Submit" value="Update Information"> </form> </cfoutput>	The Contract field requires special treatment because a check box displays and sets its value. The `cfif` structure puts a check mark in the check box if the Contract field value is Yes, and leaves the box empty otherwise.

Creating an action page to update data

You can create an action page to update data with either the `cfupdate` tag or `cfquery` with the UPDATE statement.

Creating an update action page with cfupdate

The `cfupdate` tag is the easiest way to handle simple updates from a front-end form. The `cfupdate` tag has an almost identical syntax to the `cfinsert` tag.

To use the `cfupdate` tag, you must include the primary key field(s) in your form submittal. The `cfupdate` tag automatically detects the primary key field(s) in the table that you are updating and looks for them in the submitted form fields. ColdFusion uses the primary key field(s) to select the record to update (therefore, you cannot update the primary key value itself). It then uses the remaining form fields that you submit to update the corresponding fields in the record. Your form only needs to have fields for the database fields that you want to change.

To create an update page with cfupdate:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Update Employee</title>
</head>
<body>
<cfif not isdefined("Form.Contract")>
```

```
  <cfset form.contract = "No">
<cfelse>
  <cfset form.contract = "Yes">
</cfif>

<cfupdate datasource="cfdocexamples"
  tablename="Employee">

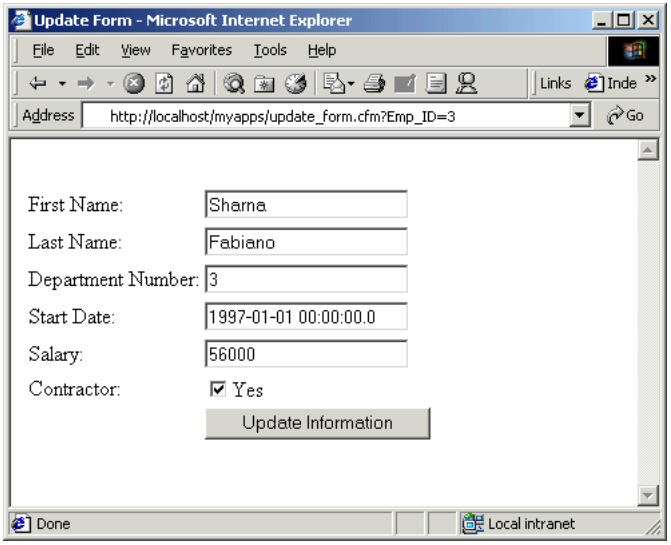
<h1>Employee Updated</h1>
<cfoutput>
You have updated the information for #Form.FirstName#
#Form.LastName# in the employee database.
</cfoutput>

</body>
</html>
```

2. Save the page as update_action.cfm.
3. View update_form.cfm in your web browser by specifying the page URL and an Employee ID; for example, enter the following:

   http://localhost/myapps/update_form.cfm?Emp_ID=3

   The current information for that record appears:



4. Enter new values in any of the fields, and click Update Information.

   ColdFusion updates the record in the Employee table with your new values and displays a confirmation message.

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfif not isdefined("Form.Contract")>`
`  <cfset Form.contract = "No">`
`<cfelse>`
`  <cfset form.contract = "Yes">`
`</cfif>`
	Sets the value of Form.Contract to No if it is not defined, or to Yes if it is defined. If the Contractor check box is unchecked, no value is passed to the action page; however, the database field must have some value.

`<cfupdate datasource="cfdocexamples"` `  tablename="Employee">`	Updates the record in the database that matches the primary key on the form (Emp_ID). Updates all fields in the record with names that match the names of form controls.

`<cfoutput>`
`You have updated the information for`
`  #Form.FirstName# #Form.LastName#`
`  in the employee database.`
`</cfoutput>`
	Informs the user that the change was made successfully.

Creating an update action page with cfquery

For more complicated updates, you can use a SQL UPDATE statement in a `cfquery` tag instead of a `cfupdate` tag. The SQL UPDATE statement is more flexible for complicated updates.

The following procedure assumes that you have created the update_action.cfm page as described in "Creating an update action page with cfupdate" on page 483.

To create an update page with cfquery:

1. In update_action.cfm, replace the `cfupdate` tag with the following highlighted `cfquery` code:

```
<html>
<head>
  <title>Update Employee</title>
</head>
<body>
<cfif not isdefined("Form.Contract")>
  <cfset form.contract = "No">
<cfelse>
  <cfset form.contract = "Yes">
</cfif>

<!--- cfquery requires date formatting when retrieving from
Access. Use the left function when setting StartDate to trim
the ".0" from the date when it first appears from the
Access database --->
 <cfquery name="UpdateEmployee" datasource="cfdocexamples">
  UPDATE Employee
  SET FirstName = '#Form.Firstname#',
    LastName = '#Form.LastName#',
    Dept_ID = #Form.Dept_ID#,
    StartDate = '#left(Form.StartDate,19)#',
    Salary = #Form.Salary#
```

```
  WHERE Emp_ID = #Form.Emp_ID#
</cfquery>

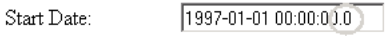
<h1>Employee Updated</h1>
<cfoutput>
You have updated the information for
#Form.FirstName# #Form.LastName#
in the employee database.
</cfoutput>
</body>
</html>
```

2. Save the page.
3. View update_form.cfm in your web browser by specifying the page URL and an Employee ID; for example, enter the following:

   http://localhost/myapps/update_form.cfm?Emp_ID=3
4. Enter new values in any of the fields, and click Update Information.

   ColdFusion updates the record in the Employee table with your new values and displays a confirmation message.

When the `cfquery` tag retrieves date information from a Microsoft Access database, it displays the date and time with tenths of seconds, as follows:



This example uses the `Left` function to trim the two final characters. The cfdocexamples data source connects to company.mdb.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description

`<cfquery name="UpdateEmployee"`
`  datasource="cfdocexamples">`
`  UPDATE Employee`
`  SET FirstName = '#Form.Firstname#',`
`  LastName = '#Form.LastName#',`
`  Dept_ID = #Form.Dept_ID#,`
`  StartDate =`
`  '#left(Form.StartDate,19)#',`
`  Salary = #Form.Salary#`
`  WHERE Emp_ID = #Form.Emp_ID#`
`</cfquery>`
	Updates the specified columns in the record in the Employee table of the cfdocexamples database that matches the primary key (Emp_ID). Because #Form.Dept_ID#, #Form.Salary#, and #Form.Emp_ID# are numeric, they do not need to be enclosed in quotation marks.
	Because of the way `cfquery` gets and displays dates from Access databases, you use the `Left` function to trim the returned value.

Deleting data

You use a `cfquery` tag with a SQL DELETE statement to delete data from a database. ColdFusion has no `cfdelete` tag.

Deleting a single record

To delete a single record, use the table's primary key in the WHERE condition of a SQL DELETE statement. In the following procedure, Emp_ID is the primary key, so the SQL Delete statement is as follows:

```
DELETE FROM Employee WHERE Emp_ID = #Form.Emp_ID#
```

You often want to see the data before you delete it. The following procedure displays the data to be deleted by reusing the form page used to insert and update data. Any data that you enter in the form before submitting it is not used, so you can use a table to display the record to be deleted instead.

To delete one record from a database:

1. In update_form.cfm, change the title to "Delete Form" and the text on the submit button to "Delete Record".
2. Change the `form` tag so that it appears as follows:

   ```
   <form action="delete_action.cfm" method="Post">
   ```

3. Save the modified file as delete_form.cfm.
4. Create a ColdFusion page with the following content:

   ```
   <html>
   <head>
   <title>Delete Employee Record</title>
   </head>
   <body>

   <cfquery name="DeleteEmployee"
     datasource="cfdocexamples">
       DELETE FROM Employee
       WHERE Emp_ID = #Form.Emp_ID#
   </cfquery>

   <h1>The employee record has been deleted.</h1>
   <cfoutput>
   You have deleted #Form.FirstName# #Form.LastName# from the
     employee database.
   </cfoutput>
   </body>
   </html>
   ```

5. Save the page as delete_action.cfm.
6. View delete_form.cfm in your web browser by specifying the page URL and an Employee ID; for example, enter the following:

   http://localhost/myapps/delete_form.cfm?Emp_ID=3.Click Delete Record

   ColdFusion deletes the record in the Employee table and displays a confirmation message.

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfquery name="DeleteEmployee"`
`  datasource="cfdocexamples">`
`  DELETE FROM Employee`
`  WHERE Emp_ID = #Form.Emp_ID#`
`</cfquery>`
	Deletes the record in the database whose Emp_ID column matches the Emp_ID (hidden) field on the form. Since the Emp_ID is the table's primary key, only one record is deleted.

`<cfoutput>`
`You have deleted #Form.FirstName#`
`  #Form.LastName# from the`
`  employee database.`
`</cfoutput>`
	Informs the user that the record was deleted.

Deleting multiple records

You can use a SQL condition to delete several records. The following example deletes the records for everyone in the Sales department (which has Dept_ID number 4) from the Employee table:

```
DELETE FROM Employee
  WHERE Dept_ID = 4
```

To delete all the records from the Employee table, use the following code:

```
DELETE FROM Employee
```

Caution: Deleting records from a database is not reversible. Use DELETE statements carefully.

CHAPTER 22
Using Query of Queries

A query that retrieves data from a record set is called a *Query of Queries*. After you generate a record set, you can interact with its results as if they were database tables by using Query of Queries. This chapter describes the benefits and procedures for this feature.

Contents

About record sets 489

About Query of Queries. 491

Query of Queries user guide 497

About record sets

Query of Queries is based on manipulating the record set, which you can create using the `cfquery` tag and other ways.

When you execute a database query, ColdFusion retrieves the data in a *record set*. In addition to presenting record set data to the user, you can manipulate this record set to improve your application's performance.

Because a record set contains rows (records) and columns (fields), you can think of it as a virtual database table, or as a spreadsheet. For example, the `cfpop` tag retrieves a record set in which each row is a message and each column is a message component, such as To, From, and Subject.

Creating a record set

You can perform a Query of Queries on any ColdFusion tag or function that generates a record set, including the following:

- `cfcollection`
- `cfdirectory`
- `cfftp`
- `cfhttp`
- `cfindex`
- `cfldap`

- cfmail
- cfpop
- cfprocresult
- cfquery (against a database or against another Query of Queries)
- cfsearch
- cfstoredproc
- cfwddx
- The QueryNew function

Creating a record set with a function

In addition to creating a record set by using a cfquery or other CFML tags, you can create it with the QueryNew() function.

To create a record set with the QueryNew() function:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>The queryNew function</title>
</head>
<body>
<h2>QueryNew Example</h2>

<!--- create a query, specify data types for each column --->
<cfset qInstruments = queryNew("name, instrument, years_playing",
  "CF_SQL_VARCHAR, CF_SQL_VARCHAR, CF_SQL_INTEGER")>

<!--- add rows --->
<cfset newrow = queryaddrow(qInstruments, 3)>

<!--- set values in cells --->
<cfset temp = querysetcell(qInstruments, "name", "Thor", 1)>
<cfset temp = querysetcell(qInstruments, "instrument", "hammer", 1)>
<cfset temp = querysetcell(qInstruments, "years_playing", "1000", 1)>

<cfset temp = querysetcell(qInstruments, "name", "Bjorn", 2)>
<cfset temp = querysetcell(qInstruments, "instrument", "sitar", 2)>
<cfset temp = querysetcell(qInstruments, "years_playing", "24", 2)>

<cfset temp = querysetcell(qInstruments, "name", "Raoul", 3)>
<cfset temp = querysetcell(qInstruments, "instrument", "flute", 3)>
<cfset temp = querysetcell(qInstruments, "years_playing", "12", 3)>

<!--- output the query --->
<cfoutput query="qInstruments">
  <pre>#name##instrument# #years_playing#</pre>
</cfoutput>

<h3>Individual record retrieval:</h3>
<cfoutput>
```

```
<p>#qInstruments.name[2]# has played #qInstruments.instrument[2]# for
  #qInstruments.years_playing[2]# years.</p>
</cfoutput>

</body>
</html>
```

2. Save the page as queryNew.cfm in the myapps directory under the *web_root* directory.
3. Display queryNew.cfm in your browser

About Query of Queries

After you have created a record set with a tag or function, you can retrieve data from the record set in one or more dependent queries. A query that retrieves data from a record set is called a Query of Queries. A typical use of a Query of Queries is to retrieve an entire table into memory with one query, and then access the table data (the record set) with subsequent sorting or filtering queries. In essence, you query the record set as if it were a database table.

Note: Because you can generate a record set in ways other than using the `cfquery` tag, the term *In Memory Query* is sometimes used instead of *Query of Queries*.

Benefits of Query of Queries

Performing a Query of Queries has many benefits, including the following:

- If you need to access the same tables multiple times, you greatly reduce access time, because the data is already in memory (in the record set).

  A Query of Queries is ideal for tables of 5,000 to 50,000 rows, and is limited only by the memory of the ColdFusion MX host computer.
- You can perform joins and union operations on results from different data sources.

  For example, you can perform a union operation on queries from different databases to eliminate duplicates for a mailing list.
- You can efficiently manipulate cached query results in different ways. You can query a database once, and then use the results to generate several different summary tables.

  For example, if you need to summarize the total salary by department, by skill, and by job, you can make one query to the database and use its results in three separate queries to generate the summaries.
- You can obtain drill-down, master-detail information for which you do not access the database for the details.

  For example, you can select information about departments and employees in a query, and cache the results. You can then display the employees' names. When users select an employee, the application displays the employee's details by selecting information from the cached query, without accessing the database.
- You can use a Query of Queries in report definitions to generate subreport data. For more information, see "Using subreports" on page 800.

Performing a Query of Queries

There are four steps to perform a Query of Queries.

To perform a Query of Queries:

1. Generate a record set through a *master query*.

   You can write a master query using a tag or function that creates a record set. For more information, see "Creating a record set" on page 489.

2. Write a *detail query*—a `cfquery` tag that specifies `dbtype="query"`.

3. In the detail query, write a SQL statement that retrieves the relevant records. Specify the names of one or more existing queries as the table names in your SQL code. Do not specify a `datasource` attribute.

4. If the database content does not change rapidly, use the `cachedwithin` attribute of the master query to cache the query results between page requests. This way, ColdFusion accesses the database on the first page request, and does not query the database again until the specified time expires. You must use the `CreateTimeSpan` function to specify the `cachedwithin` attribute value (in days, hours, minutes, seconds format).

The detail query generates a new query result set, identified by the value of the `name` attribute of the detail query. The following example illustrates the use of a master query and a single detail query that extracts information from the master.

To use the results of a query in a query:

1. Create a ColdFusion page with the following content:

```
<h1>Employee List</h1>
<!--- LastNameSearch (normally generated interactively) --->
<cfset LastNameSearch="Doe">

<!--- Master Query --->
<cfquery datasource="cfdocexamples" name="master"
  cachedwithin=#CreateTimeSpan(0,1,0,0)#>
  SELECT * from Employee
</cfquery>

<!--- Detail Query (dbtype=query, no data source) --->
<cfquery dbtype="query" name="detail">
  SELECT Emp_ID, FirstName, LastName
  FROM master
  WHERE LastName=<cfqueryparam value="#LastNameSearch#"
cfsqltype="cf_sql_char" maxLength="20"></cfquery>

<!--- output the detail query results --->
<p>Output using a query of query:</p>
<cfoutput query=detail>
  #Emp_ID#: #FirstName# #LastName#<br>
</cfoutput>

<p>Columns in the master query:</p>
<cfoutput>
  #master.columnlist#<br>
```

```
</cfoutput>

<p>Columns in the detail query:</p>
<cfoutput>
  #detail.columnlist#<br>
</cfoutput>
```

2. Save the page as query_of_query.cfm in the myapps directory under the *web_root*.
3. Display query_of_query.cfm in your browser

Reviewing the code

The master query retrieves the entire Employee table from the cfdocexamples data source. The detail query selects only the three columns to display for employees with the specified last name. The following table describes the code and its function:

Code	Description
`cfset LastNameSearch="Doe"`	Sets the last name to use in the detail query. In a complete application, this information comes from user interaction.
`<cfquery datasource="cfdocexamples" name="master" cachedwithin=#CreateTimeSpan(0,1,0,0)#> SELECT * from Employee </cfquery>`	Queries the cfdocexamples data source and selects all data in the Employees table. Caches the query data between requests to this page, and does not query the database if the cached data is less than an hour old.
`<cfquery dbtype="query" name="detail"> SELECT Emp_ID, FirstName, LastName FROM master WHERE LastName=<cfqueryparam value="#LastNameSearch#" cfsqltype="cf_sql_char" maxLength="20"></cfquery>`	Uses the master query as the source of the data in a new query, named detail. This new query selects only entries that match the last name specified by the `LastNameSearch` variable. The query also selects only three columns of data: employee ID, first name, and last name. The query uses the `cfqueryparam` tag to prevent passing erroneous or harmful code.
`<cfoutput query=detail> #Emp_ID#: #FirstName# #LastName#   </cfoutput>`	Uses the detail query to display the list of employee IDs, first names, and last names.
`<cfoutput> #master.columnlist#  </cfoutput>`	Lists all the columns returned by the master query.
`<cfoutput> #detail.columnlist#  </cfoutput>`	Lists all the columns returned by the detail query.

Displaying record set data incrementally

If your database is large, you can limit the number of rows displayed at one time. The following example shows how to use the `currentRow` query variable of a Query of Queries to do this. For more information on query variables, see "Getting information about query results" on page 471.

To display record set data incrementally:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>QoQ with incremental row return</title>
</head>

<body>
<h3>QoQ with incremental row return</h3>
<!--- define startrow and maxrows to facilitate 'next N' style browsing --->
<cfparam name = "MaxRows" default = "5">
<cfparam name = "StartRow" default = "1">

<!--- master query: retrieve all info from Employee table --->
<cfquery name = "GetSals" datasource = "cfdocexamples">
  SELECT * FROM Employee
  ORDER BY LastName
</cfquery>

<!--- detail query: select 3 fields from the master query --->
<cfquery name = "GetSals2" dbtype = "query">
  SELECT   FirstName, LastName, Salary
  FROM     GetSals
  ORDER BY LastName
</cfquery>

<!--- build table to display output --->
<table cellpadding = 1 cellspacing = 1>
  <tr>
    <td bgcolor = f0f0f0>
    <b><i> </i></b>
    </td>

    <td bgcolor = f0f0f0>
    <b><i>FirstName</i></b>
    </td>

    <td bgcolor = f0f0f0>
    <b><i>LastName</i></b>
    </td>

    <td bgcolor = f0f0f0>
    <b><i>Salary</i></b>
    </td>
  </tr>

<!--- Output the query and define the startrow and maxrows
     parameters. Use the query variable currentRow to
     keep track of the row you are displaying. --->
  <cfoutput query = "GetSals2" startrow = "#StartRow#" maxrows =
  "#MaxRows#">
  <tr>
    <td valign = top bgcolor = ffffed>
      <b>#GetSals2.currentRow#</b>
```

```
    </td>

    <td valign = top>
      <font size = "-1">#FirstName#</font>
    </td>

    <td valign = top>
      <font size = "-1">#LastName#</font>
    </td>

    <td valign = top>
      <font size = "-1">#LSCurrencyFormat(Salary)#</font>
    </td>
  </tr>
  </cfoutput>
<!--- If the total number of records is less than or equal to
  the total number of rows, provide a link to the same page, with the
  StartRow value incremented by MaxRows (5, in this example) --->
  <tr>
    <td colspan = 4>
    <cfif (startrow + maxrows) lte getsals2.recordcount>
      <a href="qoq_next_row.cfm?startrow=<cfoutput>#Evaluate(StartRow +
  MaxRows)#</cfoutput>">See next <cfoutput>#MaxRows#</cfoutput>
      rows</a>
    </cfif>
    </td>
  </tr>
</table>
</body>
</html>
```

2. Save the page as qoq_next_row.cfm in the myapps directory under the *web_root*.
3. Display qoq_next_row.cfm in your browser

Using the cfdump tag with query results

As you debug your CFML code, you can use the `cfdump` tag to quickly display the contents of your query. This tag has the following format:

```
<cfdump var="#query_name#">
```

For more information on the `cfdump` tag, see *CFML Reference*.

Using Query of Queries with non-SQL record sets

A Query of Queries can operate on any CFML tag or function that returns a record set; you are not limited to operating on `cfquery` results. You can perform queries on non-SQL record sets, such as a `cfdirectory` tag, a `cfsearch` tag, a `cfldap` tag, and so on.

The following example shows how a Query of Queries interacts with the record set of a Verity search. This example assumes that you have a valid Verity collection, called bbb, which contains documents with a target word, film, or its variants (films, filmed, filming). Change the name of the collection and the search criteria to as appropriate for your Verity collection. For more information on Verity, see Chapter 24, “Building a Search Interface,” on page 547.

To use Query of Queries with a Verity record set:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>QoQ and Verity</title>
</head>

<body>
<!--- master query: retrieve all documents from the bbb collection
that contain 'film' (or its stemmed variants); change values for
collection and criteria as needed for your Verity collection --->
<cfsearch name = "quick"
  collection="bbb"
  type = "simple"
  criteria="film">

<h3>Master query dump:</h3>
<cfdump var="#quick#">

<!--- detail query: retrieve from the master query only those
documents with a score greater than a criterion (here,
0.7743) --->
<cfquery name="qoq" dbtype="query">
  SELECT * from quick
  WHERE quick.score > 0.7743
</cfquery>

<h3>Detail query dump:</h3>
<cfdump var="#qoq#">

</body>
</html>
```

2. Save the page as qoq_verity.cfm in the myapps directory under the *web_root*.
3. Display qoq_verity.cfm in your browser

The next example shows how a Query of Queries combines record sets from a `cfdirectory` tag, which is limited to retrieval of one file type per use.

To use Query of Queries to combine record sets:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>Images Folder</title>
</head>

<body>
<h2>Image Retrieval with QoQ</h2>
<!--- set the images directory --->
<cfset dir = ("C:\pix\")>

<!--- Retrieve all GIFs --->
<cfdirectory name="GetGIF"
```

```
  action="list"
  directory="#dir#"
  filter="*.gif">

<!--- Retrieve all JPGs --->
<cfdirectory name="GetJPG"
  action="list"
  directory="#dir#"
  filter="*.jpg">

<!--- Join the queries with a UNION in a QoQ (cfdirectory
      automatically returns the directory name as "Name") --->
<cfquery dbtype="query" name="GetBoth">
  SELECT * FROM GetGIF
  UNION
  SELECT * FROM GetJPG
  ORDER BY Name
</cfquery>

<!--- Display output in a linked, ordered list --->
<cfoutput>
  <p>The <strong>#dir#</strong> directory contains #GetBoth.RecordCount#
  images:<br>
  <ol>
    <cfloop query="GetBoth">
      <li><a href="../images/#Name#">#GetBoth.Name#</a><br>
    </cfloop>
  </ol>
</cfoutput>

</body>
</html>
```

2. Save the page as qoq_cfdirectory.cfm in the myapps directory under the *web_root*.
3. Display qoq_cfdirectory.cfm in your browser

Query of Queries user guide

The following sections discuss Query of Queries functionality. If you know SQL or have interacted with databases, you might be familiar with some of these features.

Using dot notation

ColdFusion supports using dot notation in table names.

Example

If a structure named A contains a field named B, which contains a table named Products, you can refer to the table with dot notation, as follows:

```
SELECT tape_ID, length
FROM A.B.Products;
```

Using joins

A join operation uses a single SELECT statement to return a result set from multiple, related tables, typically those with a primary key - foreign key relationship. There are two SQL clauses that perform joins:

- **WHERE clause** ColdFusion supports joins through a WHERE clause.
- **INNER JOIN and OUTER JOIN** ColdFusion does not support joins through INNER JOIN or OUTER JOIN clauses.

Note: Query of Queries supports joins between two tables only.

Using unions

The UNION operator lets you combine the results of two or more SELECT expressions into a single record set. The original tables must have the same number of columns, and corresponding columns must be UNION-compatible data types. Columns are UNION-compatible data types if they meet one of the following conditions:

- The same data type; for example, both Tinyint
- Both Numeric; for example, Tinyint, Smallint, Integer, Bigint, Double, Float, Real, Decimal, or Numeric
- Both Characters; for example, Char, Varchar, or LongVarchar
- Both Dates; for example, Time, TimeStamp, or Date

Note: Query Of Queries does not support ODBC-formatted dates and times.

Syntax

```
select_expression = select_expression UNION [ALL] select_expression
```

Example

This example uses the following tables:

Table1	
Type(int)	Name(varchar)
1	Tennis
2	Baseball
3	Football

Table2	
ID(int)	Sport(varchar)
3	Football
4	Volleyball
5	PingPong

To combine Table1 and Table2, use a UNION statement, as follows:

```
SELECT * FROM Table1
UNION
SELECT * FROM Table2
```

The UNION statement produces the following result (UNION) table:

Result table	
Type(int)	Name(varchar)
1	Tennis
2	Baseball
3	Football
4	Volleyball
5	PingPong

Using aliases for column names

The column names of a UNION table are the column names in the result set of the first SELECT statement in the UNION operation; ColdFusion ignores the column names in the other SELECT statement. To change the column names of the result table, you can use an alias, as follows:

```
Select Type as SportType, Name as SportName from Table1
UNION
Select * from Table2
```

Duplicate rows and multiple tables

By default, the UNION operator removes duplicate rows from the result table. If you use the keyword ALL, then duplicates are included.

You can combine an unlimited number of tables using the UNION operator, for example:

```
Select * from Table1
UNION
Select * from Table2
UNION
Select * from Table3
...
```

Parentheses and evaluation order

By default, the Query of Queries SQL engine evaluates a statement containing UNION operators from left to right. You can use parentheses to change the order of evaluation. For example, the following two statements are different:

```
/* First statement. */
SELECT * FROM TableA
UNION ALL
(SELECT * FROM TableB
UNION
```

```
SELECT * FROM TableC
)

/* Second statement. */
(SELECT * FROM TableA
UNION ALL
SELECT * FROM TableB
)
UNION
SELECT * FROM TableC
```

In the first statement, there are no duplicates in the union between TableB and TableC. Then, in the union between that set and TableA, the ALL keyword includes the duplicates. In the second statement, duplicates are included in the union between TableA and TableB but are eliminated in the subsequent union with TableC. The ALL keyword has no effect on the final result of this expression.

Using other keywords with UNION

When you perform a UNION, the individual SELECT statements cannot have their own ORDER BY or COMPUTE clauses. You can only have one ORDER BY or COMPUTE clause after the last SELECT statement; this clause is applied to the final, combined result set. You can only specify GROUP BY and HAVING expressions in the individual SELECT statements.

Using conditional operators

ColdFusion lets you use the following conditional operators in your SQL statements:

- Test
- Null
- Comparison
- Between
- IN
- LIKE

Test conditional

This conditional tests whether a Boolean expression is True, False, or Unknown.

Syntax

```
cond_test ::= expression [IS [NOT] {TRUE | FALSE | UNKNOWN} ]
```

Example

```
SELECT _isValid FROM Chemicals
WHERE _isValid IS true;
```

Null conditional

This conditional tests whether an expression is null.

Syntax

```
null_cond ::= expression IS [NOT] NULL
```

Example

```
SELECT bloodVal FROM Standards
WHERE bloodVal IS NOT null;
```

Comparison conditional

This conditional lets you compare an expression against another expression of the same data type (Numeric, String, Date, or Boolean). You can use it to selectively retrieve only the relevant rows of a record set.

Syntax

```
comparison_cond ::= expression [> | >= | <> | != | < | <=] expression
```

Example

The following example uses a comparison conditional to retrieve only those dogs whose IQ is at least 150:

```
SELECT dog_name, dog_IQ
FROM Dogs
WHERE dog_IQ >= 150;
```

Between conditional

This conditional lets you compare an expression against another expression. You can use it to selectively retrieve only the relevant rows of a record set. Like the comparison conditional, the BETWEEN conditional makes a comparison; however, the between conditional makes a comparison against a range of values. Therefore, its syntax requires two values, which are inclusive, a minimum and a maximum. You must separate these values with the AND keyword.

Syntax

```
between_cond ::= expression [NOT] BETWEEN expression AND expression
```

Example

The following example uses a BETWEEN conditional to retrieve only those dogs whose IQ is between 150 and 165, inclusive:

```
SELECT dog_name, dog_IQ
FROM Dogs
WHERE dog_IQ BETWEEN 150 AND 165;
```

IN conditional

This conditional lets you specify a comma-delimited list of conditions to match. It is similar in function to the OR conditional. In addition to being more legible when working with long lists, the IN conditional can contain another SELECT statement.

Syntax

```
in_cond ::= expression [NOT] IN (expression_list)
```

Example

The following example uses the IN conditional to retrieve only those dogs who were born at either Ken's Kennels or Barb's Breeders:

```
SELECT dog_name, dog_IQ, Kennel_ID
FROM Dogs
WHERE kennel_ID IN ('Kens','Barbs');
```

LIKE conditional

This conditional lets you perform wildcard searches, in which you compare your data to search patterns. This strategy differs from other conditionals, such as BETWEEN or IN, because the LIKE conditional compares your data to a value that is partially unknown.

Syntax

```
like_cond ::= left_string_exp [NOT] LIKE right_string_exp [ESCAPE escape_char]
```

The left_string_exp can be either a constant string, or a column reference to a string column. The right_string_exp can be either a column reference to a string column, or a search pattern. A *search pattern* is a search condition that consists of literal text and at least one wildcard character. A *wildcard character* is a special character that represents an unknown part of a search pattern, and is interpreted as follows:

- The underscore (_) represents any single character.
- The percent sign (%) represents zero or more characters.
- Square brackets ([ ]) represents any character in the range.
- Square brackets with a caret [^] represent any character not in the range.
- All other characters represent themselves.

Note: Earlier versions of ColdFusion do not support bracketed ranges.

Examples

The following example uses the LIKE conditional to retrieve only those dogs of the breed Terrier, whether the dog is a Boston Terrier, Jack Russell Terrier, Scottish Terrier, and so on:

```
SELECT dog_name, dog_IQ, breed
FROM Dogs
WHERE breed LIKE '%Terrier';
```

The following examples are select statements that use bracketed ranges:

```
SELECT lname FROM Suspects WHERE lname LIKE 'A[^c]%';
SELECT lname FROM Suspects WHERE lname LIKE '[a-m]%';
SELECT lname FROM Suspects WHERE lname LIKE '%[]';
SELECT lname FROM Suspects WHERE lname LIKE 'A[%]%';
SELECT lname FROM Suspects WHERE lname LIKE 'A[^c-f]%';
```

Case sensitivity

Unlike the rest of ColdFusion, Query of Queries is case-sensitive. However, Query of Queries supports two string functions, `UPPER()` and `LOWER()`, which you can use to achieve case-insensitive matching.

Examples

The following example matches only 'Sylvester':

```
SELECT dog_name
FROM Dogs
WHERE dog_name LIKE 'Sylvester';
```

The following example is not case-sensitive; it uses the `LOWER()` function to treat 'Sylvester', 'sylvester', 'SYLVESTER', and so on as all lowercase, and matches them with the all lowercase string, 'sylvester':

```
SELECT dog_name
FROM Dogs
WHERE LOWER(dog_name) LIKE 'sylvester';
```

If you use a variable on the right side of the LIKE conditional and want to ensure that the comparison is not case-sensitive, use the `LCase` or `UCase` function to force the variable text to be all of one case, as in the following example:

```
WHERE LOWER(dog_name) LIKE '#LCase(FORM.SearchString)#';
```

Escaping wildcards

You can specify your own escape character using the conditional ESCAPE clause.

Example

The following example uses the ESCAPE clause to enable a search for a literal percent sign (%), which ColdFusion normally interprets as a wildcard character:

```
SELECT emp_discount
FROM Benefits
WHERE emp_discount LIKE '10\%'
ESCAPE '\';
```

Managing data types for columns

A Query of Queries requires that every column have metadata that defines the column's data type. All queries that ColdFusion MX creates have metadata. However, a query created with `QueryNew` function that omits the second parameter does not contain metadata. You use this optional second parameter to define the data type of each column in the query.

To specify column data types in the QueryNew function:

- Type a `QueryNew` function, specifying the column names in the first parameter and the data types in the second parameter, as the following example shows:

```
<cfset qInstruments = queryNew("name, instrument, years_playing",
  "CF_SQL_VARCHAR, CF_SQL_VARCHAR, CF_SQL_INTEGER")>
```

Tip: To see the metadata for a Query of Queries, use the `GetMetaData` function.

To specify the column data types in the QueryAddColumn function:

1. Create a query by specifying the QueryNew function with no parameters.

```
<!--- Make a query. --->
<cfset myQuery = QueryNew("")>
```

2. Add and populate a column with the `QueryAddColumn` function, specifying the data type in the third parameter:

```
<!--- Create an array. --->
<cfset FastFoodArray = ArrayNew(1)>
<cfset FastFoodArray[1] = "French Fries">
<cfset FastFoodArray[2] = "Hot Dogs">
<cfset FastFoodArray[3] = "Fried Clams">
<cfset FastFoodArray[4] = "Thick Shakes">
<!--- Use the array to add a column to the query. --->
<cfset nColumnNumber = QueryAddColumn(myQuery, "FastFood", "CF_SQL_VARCHAR",
    FastFoodArray)>
```

If you do not specify the data type, ColdFusion MX examines the first fifty rows of each column to determine the data type when performing conditional expressions.

In some cases, ColdFusion MX can guess a data type that is inappropriate for your application. In particular, if you use columns in a WHERE clause or other conditional expression, the data types must be compatible. If they are not compatible, you must use the CAST function to recast one of the columns to a compatible data type. For more information on casting, see "Using the CAST function" on page 504. For more information on data type compatibility, see "Understanding Query of Queries processing" on page 512.

Note: Specifying the data type in the `QueryNew` function helps you avoid compatibility issues.

Using the CAST function

In some cases, a column's data type may not be compatible with the processing you want to do. For example, query columns returned by the `cfhttp` tag are all of type CF_SQL_VARCHAR, even though the contents may be numeric. In this case, you can use the Query of Queries CAST function to convert a column value into an expression of the correct data type.

The syntax for the CAST function is as follows:

```
CAST ( expression AS castType )
```

Where *castType* is one of the following:

- BINARY
- BIGINIT
- BIT
- DATE
- DECIMAL
- DOUBLE
- INTEGER
- TIME
- TIMESTAMP
- VARCHAR

For example:

```
<cfhttp url="http://quote.yahoo.com/download/
  quotes.csv?Symbols=csco,jnpr&format=sc1l1&ext=.csv"
        method="GET"
        name="qStockItems"
        columns="Symbol,Change,LastTradedPrice"
        textqualifier="""
        delimiter=","
        firstrowasheaders="no">
<cfoutput>
  <cfdump var="#qStockItems#">
  <cfdump var="#qStockItems.getColumnNames()#">
</cfoutput>

<cfoutput>
  <cfloop index="i" from="1" to="#arrayLen(qStockItems.getColumnNames())#">
       #qStockItems.getMetaData().getColumnTypeName(javaCast("int",i))#<br/>
  </cfloop>
</cfoutput>
<cftry>
    <cfquery name="hello" dbtype="query">
        SELECT SUM(CAST(qStockItems.LastTradedPrice as INTEGER))
       AS SUMNOW from qStockItems
    </cfquery>
    <cfcatch>Error in Query of Queries</cfcatch>
</cftry>

<cfoutput>
  <cfdump var="#hello#">
</cfoutput>
```

Using aggregate functions

Aggregate functions operate on a set of data and return a single value. Use these functions for retrieving summary information from a table, as opposed to retrieving an entire table and then operating on the record set of the entire table.

Consider using aggregate functions to perform the following operations:

- To display the average of a column
- To count the number of rows for a column
- To find the earliest date in a column

Since not every relational database management system (RDBMS) supports all aggregate functions, refer to your database's documentation. The following table lists the aggregate functions that ColdFusion supports:

Function	Description
AVG()	Returns the average (mean) for a column.
COUNT()	Returns the number of rows in a column.
MAX()	Returns the largest value of a column.
MIN()	Returns the lowest value of a column.
SUM()	Returns the sum of values of a column.

Syntax

```
aggregate_func ::= <COUNT>(* | column_name) | AVG | SUM | MIN | MAX)
([ALL | DISTINCT] numeric_exp)
```

Example

The following example uses the `AVG()` function to retrieve the average IQ of all terriers:

```
SELECT dog_name, AVG(dog_IQ) AS avg_IQ
FROM Dogs
WHERE breed LIKE '%Terrier';
```

Arbitrary expressions in aggregate functions

ColdFusion supports aggregate functions of any arbitrary expression, as follows:

```
SELECT lorange, count(lorange+hirange)
FROM roysched
GROUP BY lorange;
```

Aggregate functions in arbitrary expressions

ColdFusion supports mathematical expressions that include aggregate functions, as follows:

```
SELECT MIN(lorange) + MAX(hirange)
FROM roysched
GROUP BY lorange;
```

Using group by and having expressions

ColdFusion supports the use of any arbitrary arithmetic expression, as long as it is referenced by an alias.

Examples

The following code is correct:

```
SELECT (lorange + hirange)/2 AS midrange,
COUNT(*)
FROM roysched
GROUP BY midrange;
```

The following code is correct:

```
SELECT (lorange+hirange)/2 AS x,
COUNT(*)
FROM roysched GROUP BY x
HAVING x > 10000;
```

The following code is not supported in Query of Queries:

```
SELECT (lorange + hirange)/2 AS midrange,
COUNT(*)
FROM roysched
GROUP BY (lorange + hirange)/2;
```

Using ORDER BY clauses

ColdFusion supports the ORDER BY clause to sort. Make sure that it is the last clause in your SELECT statement. You can sort by multiple columns, by relative column position, by nonselected columns. You can specify a descending sort direction with the DESC keyword (by default, most RDBMS sorts are ascending, which makes the ASC keyword unnecessary).

Syntax

```
order_by_column ::= ( <IDENTIFIER> | <INTEGER_LITERAL> ) [<ASC> | <DESC>]
```

Examples

The following example shows a simple sort using an ORDER BY clause:

```
SELECT acetylcholine_levels, dopamine_levels
FROM results
ORDER BY dopamine_levels
```

The following example shows a more complex sort; results are first sorted by ascending levels of dopamine, then by descending levels of acetylcholine. The ASC keyword is unnecessary, and is used only for legibility.

```
SELECT acetylcholine_levels, dopamine_levels
FROM results
ORDER BY 2 ASC, 1 DESC
```

Using aliases

ColdFusion supports the use of database column aliases. An *alias* is an alternate name for a database field or value. ColdFusion lets you reuse an alias in the same SQL statement.

One way to create an alias is to concatenate (append) two or more columns to generate a value. For example, you can concatenate a first name and a last name to create the value fullname. Because the new value does not exist in a database, you refer to it by its alias. The AS keyword assigns the alias in the SELECT statement.

Examples

ColdFusion supports alias substitutions in the ORDER BY, GROUP BY, and HAVING clauses.

Note: ColdFusion does not support aliases for table names.

```
SELECT FirstName + ' ' + LastName AS fullname
from Employee;
```

The following examples rely on these two master queries:

```
<cfquery name="employee" datasource="2pubs">
  SELECT * FROM employee
</cfquery>

<cfquery name="roysched" datasource="2pubs">
  SELECT * FROM roysched
</cfquery>
```

ORDER BY example

```
<cfquery name="order_by" dbtype="query">
  SELECT (job_id + job_lvl)/2 AS job_value
  FROM employee
  ORDER BY job_value
</cfquery>
```

GROUP BY example

```
<cfquery name="group_by" dbtype="query">
  SELECT lorange+hirange AS x, count(hirange)
  FROM roysched
  GROUP BY x
</cfquery>
```

HAVING example

```
<cfquery name="having" dbtype="query">
  SELECT (lorange+hirange)/2 AS x,
  COUNT(*)
  FROM roysched GROUP BY x
  HAVING x > 10000
</cfquery>
```

Handling null values

ColdFusion uses Boolean logic to handle conditional expressions. Proper handling of NULL values requires the use of ternary logic. The `IS [NOT] NULL` clause works correctly in ColdFusion. However the following expressions do not work properly when the column breed is NULL:

```
WHERE (breed > 'A')
WHERE NOT (breed > 'A')
```

The correct behavior should not include NULL breed columns in the result set of either expression. To avoid this limitation, you can add an explicit rule to the conditionals and rewrite them in the following forms:

```
WHERE breed IS NOT NULL AND (breed > 'A')
WHERE breed IS NOT NULL AND not (breed > 'A')
```

Concatenating strings

Query of Queries support two string concatenation operators: + and ||, as the following examples show:

```
LASTNAME + ', ' + FIRSTNAME
LASTNAME || ', ' || FIRSTNAME
```

Escaping reserved keywords

ColdFusion has a list of reserved keywords, which are typically part of the SQL language and are not normally used for names of columns or tables. To escape a reserved keyword for a column name or table name, enclose it in brackets.

Caution: Earlier versions of ColdFusion let you use some reserved keywords without escaping them.

Examples

ColdFusion supports the following SELECT statement examples:

```
SELECT [from] FROM parts;
SELECT [group].firstname FROM [group];
SELECT [group].[from] FROM [group];
```

ColdFusion does not support nested escapes, such as in the following example:

```
SELECT [[from]] FROM T;
```

The following table lists ColdFusion reserved keywords:

ABSOLUTE	ACTION	ADD	ALL	ALLOCATE
ALTER	AND	ANY	ARE	AS
ASC	ASSERTION	AT	AUTHORIZATION	AVG
BEGIN	BETWEEN	BIT	BIT_LENGTH	BOTH
BY	CASCADE	CASCADED	CASE	CAST

CATALOG	CHAR	CHARACTER	CHARACTER_ LENGTH	CHAR_LENGTH
CHECK	CLOSE	COALESCE	COLLATE	COLLATION
COLUMN	COMMIT	CONNECT	CONNECTION	CONSTRAINT
CONSTRAINTS	CONTINUE	CONVERT	CORRESPONDING	COUNT
CREATE	CROSS	CURRENT	CURRENT_DATE	CURRENT_TIME
CURRENT_ TIMESTAMP	CURRENT_ USER	CURSOR	DATE	DAY
DEALLOCATE	DEC	DECIMAL	DECLARE	DEFAULT
DEFERRABLE	DEFERRED	DELETE	DESC	DESCRIBE
DESCRIPTOR	DIAGNOSTICS	DISCONNECT	DISTINCT	DOMAIN
DOUBLE	DROP	ELSE	END	END-EXEC
ESCAPE	EXCEPT	EXCEPTION	EXEC	EXECUTE
EXISTS	EXTERNAL	EXTRACT	FALSE	FETCH
FIRST	FLOAT	FOR	FOREIGN	FOUND
FROM	FULL	GET	GLOBAL	GO
GOTO	GRANT	GROUP	HAVING	HOUR
IDENTITY	IMMEDIATE	IN	INDICATOR	INITIALLY
INNER	INPUT	INSENSITIVE	INSERT	INT
INTEGER	INTERSECT	INTERVAL	INTO	IS
ISOLATION	JOIN	KEY	LANGUAGE	LAST
LEADING	LEFT	LEVEL	LIKE	LOCAL
LOWER	MATCH	MAX	MIN	MINUTE
MODULE	MONTH	NAMES	NATIONAL	NATURAL
NCHAR	NEXT	NO	NOT	NULL
NULLIF	NUMERIC	OCTET_LENGTH	OF	ON
ONLY	OPEN	OPTION	OR	ORDER
OUTER	OUTPUT	OVERLAPS	PAD	PARTIAL
POSITION	PRECISION	PREPARE	PRESERVE	PRIMARY
PRIOR	PRIVILEGES	PROCEDURE	PUBLIC	READ
REAL	REFERENCES	RELATIVE	RESTRICT	REVOKE
RIGHT	ROLLBACK	ROWS	SCHEMA	SCROLL
SECOND	SECTION	SELECT	SESSION	SESSION_USER
SET	SMALLINT	SOME	SPACE	
SQL	SQLCODE	SQLERROR	SQLSTATE	SUBSTRING

SUM	SYSTEM USER	TABLE	TEMPORARY	THEN
TIME	TIMESTAMP	TIMEZONE HOUR	TIMEZONE_MINUTE	TO
TRAILING	TRANSACTION	TRANSLATE	TRANSLATION	TRIM
TRUE	UNION	UNIQUE	UNKNOWN	UPDATE
UPPER	USAGE	USER	USING	VALUE
VALUES	VARCHAR	VARYING	VIEW	WHEN
WHENEVER	WHERE	WITH	WORK	WRITE
YEAR	ZONE			

Using Queries of Queries with dates

If you create a query object with the `QueryNew` function and populate a column with date constants, ColdFusion MX stores the dates as a string inside the query object until a Query of Queries is applied to the query object. When ColdFusion applies a Query of Queries to the query object, it converts the string representations into date objects.

Query of Queries supports date constants in SQL and ODBC format, as follows:

- **SQL format** Dates, times, or timestamps in one of the following format:
  - **Date string** *yyyy-mm-dd*, for example, 1955-06-13.
  - **Time string** *hh:mm:ss[.[nnn]]*, for example, 14:34:30.75.
  - **Timestamp string** *yyyy-mm-dd hh:mm:ss[.[nnn]]*, for example, 1924-01-14 12:00:00.000.
- **ODBC format** Dates, times, or timestamps in one of the following format:
  - **Date string** *{d 'value'}*, for example, {d '2004-07-06'}.
  - **Time string** *{t 'value'}*, for example, {t '13:45:30'}.
  - **Timestamp string** *{ts 'value'}*, for example, {ts '2004-07-06 13:45:30'}.

Tip: If you want to convert the date to its original format, use the `DateFormat` function and apply the "mm/dd/yy" mask.

Understanding Query of Queries performance

Query of Queries performs very well on single-table query objects that were accessed directly from a database. This is because ColdFusion MX stores meta information for a query object accessed from a database.

When working with a query resulting in a SQL join, Query of Queries performs as follows:

- Query of Queries is very efficient for simple joins in which there is only one equality between two column references or constants; for example:

```
SELECT T1.a, b, c, d FROM T1, T2 WHERE T1.a = T2.a
```

- Query of Queries is less efficient for joins in which the predicate contains multiple expressions; for example:

```
SELECT T1.a, b, c, d FROM T1, T2
WHERE T1.a = T2.a AND T1.b + T1.c = T2.b + T2.c
```

Understanding Query of Queries processing

The following discussions describe how Query of Queries handles the following:

- Column comparisons
- Queries passed by reference
- Complex objects

Comparing columns with different data types

ColdFusion MX 7 includes enhancements that allow you to compare columns with different data types.

If one of the operands has a known column type (only constants have an unknown column type), Query of Queries tries to coerce the constant with an unknown type to the type of the operand with metadata. The pairs of allowed coercions are as follows:

- Binary, string
- Dates, string
- Numeric, bigdecimal
- Boolean, numeric

That is, ColdFusion MX can coerce between binary and string, but not between date and string.

If both operands have known data types, the types must be the same. The only exception is that ColdFusion MX can coerce among integer, float, and double.

If both operands are constants, ColdFusion MX tries to coerce the values, first to the most restrictive type, then to the least restrictive type.

- First to binary then to string.
- First to date then to string.
- First to boolean then to numeric.

Passing queries by reference

A Query of Queries is copied by reference from its related query; this means that ColdFusion MX does not create a new query when you create a Query of Queries. It also means that changes to a Query of Queries, such as ordering, modifying, and deleting data, are also applied to the base query object.

If you do not want the original query to change, use the `Duplicate` function to create a copy and create the Query of Queries using the copied query.

Managing complex objects

You cannot use Query Of Queries on a record set that contains complex objects, such as arrays and structures.

Note: You can store a record set in a complex objects.

CHAPTER 23
Managing LDAP Directories

CFML applications use the `cfldap` tag to access and manage LDAP (Lightweight Directory Access Protocol) directories. This chapter provides information on how to use this tag to view, query, and update LDAP directories.

This chapter teaches you how to query and update an LDAP database. It does not assume that you are familiar with LDAP, and provides an introduction to LDAP directories and the LDAP protocol. However, it does assume that you have information on your LDAP database's structure and attributes, and it does not explain how to create an LDAP directory or manage a directory server. To learn more about LDAP and LDAP servers, see your LDAP server documentation and published books on LDAP.

The examples in this chapter use the Airius sample LDAP database that is supplied with the Netscape and iPlanet Directory Servers.

Contents

About LDAP 516

The LDAP information structure. 517

Using LDAP with ColdFusion 520

Querying an LDAP directory. 521

Updating an LDAP directory. 527

Advanced topics 538

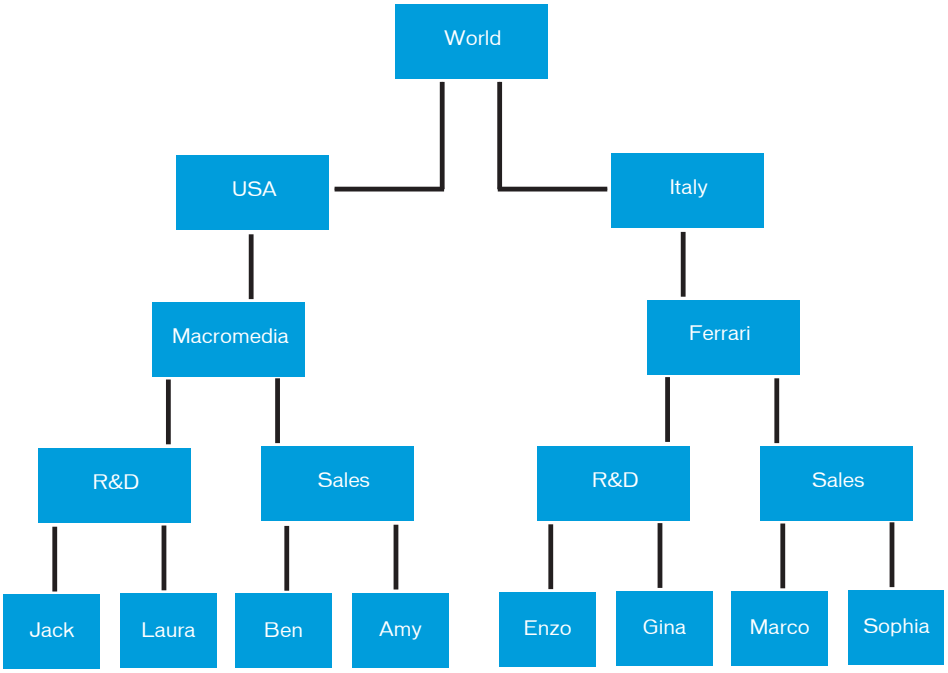
About LDAP

The LDAP protocol enables organizations to arrange and access directory information in a hierarchy. In this context, *directory* refers to a collection of information, such as a telephone directory, not a collection of files in a folder on a disk drive.

LDAP originated in the mid-1990s as a response to the need to access ISO X.500 directories from personal computers that had limited processing power. Since then, products such as iPlanet Server have been developed that are native LDAP directory servers. Several companies now provide LDAP access to their directory servers, including Novell NDS, Microsoft Active Directory Services (ADS), Lotus Domino, and Oracle.

An LDAP directory is typically a hierarchically structured database. Each layer in the hierarchy typically corresponds to a level of organizational structure.

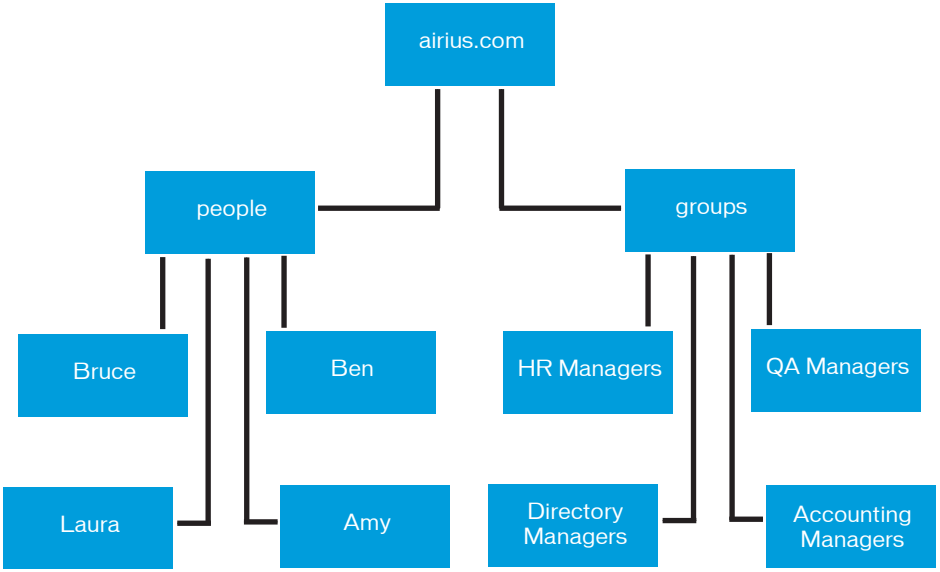
The following example shows a simple directory structure:



This example is fully symmetrical: all the entries at each layer are of the same type.

You can also structure an LDAP directory so that the layers under one entry contain different information from the layers under another entry.

The following figure show such an asymmetric directory:



In this directory structure, the second level of the tree divides the directory into two organizational units: people and groups. The third level contains entries with information that is specific to the organizational unit. Each person's entry includes a name, e-mail address, and telephone number. Each group's entry includes the names of group members.

This complexity and flexibility is a key to LDAP's usefulness. With it, you can represent any organizational structure.

LDAP offers performance advantages over conventional databases for accessing hierarchical, directory-like information that is read frequently and changed infrequently.

Although LDAP is often used for e-mail, address, telephone, or other organizational directories, it is not limited to these types of applications. For example, you can store ColdFusion MX Advanced Security information in an LDAP database.

The LDAP information structure

The following sections describe the LDAP information structure: the elements of an LDAP directory and how they are structured. These sections describe the following basic LDAP concepts:

- Entry
- Attribute
- Distinguished name (DN)
- Schema, including the object class and attribute type

Entry

The basic information object of LDAP is the *entry*. An entry is composed of one or more *attributes*. Entries are subject to content rules defined by the directory *schema* (see "Schema" on page 519).

Each node, not just the terminal nodes, of an LDAP directory is an entry. In the preceding figures, each item is an entry. For example, in the first diagram, both USA and Ferrari are entries. The USA entry's attributes could include a Language attribute, and the Ferrari entry could include an entry for the chief executive officer.

Attribute

An LDAP directory entry consists of one or more attributes. Attributes have *types* and *values*. The type determines the information that the values can contain. The type also specifies how the value is processed. For example, the type determines whether an attribute can have multiple values. The mail attribute type, which contains an e-mail address, is multivalued so you can store multiple e-mail addresses for one person.

Some commonly-used attribute types have short keyword type names. Often these correspond to longer type names, and the two names can be used interchangeably. The following table lists common attribute type keywords used in LDAP directories:

Keyword	Long name	Comment
c	CountryName	
st	stateOrProvinceName	
l	LocalityName	Typically, city, but can be any geographical unit
street	StreetAddress	
o	OrganizationName	
ou	OrganizationalUnitName	
cn	CommonName	Typically, first and last name
sn	SurName	
dc	domaincomponent	
mail	mail	E-mail address

For more information, see "Attribute type" on page 520.

Distinguished name (DN)

An entry's *distinguished name* uniquely identifies it in the directory. A DN is made up of *relative distinguished names* (RDNs). An RDN identifies the entry among the children of its parent entry. For example, in the first figure in "About LDAP", the RDN for the Ferrari entry is "o=Ferrari".

An entry's DN consists of an entry's RDN followed by the DN of its parent. In other words, it consists of the RDNs for the entry and each of the entry's parent entries, up to the root of the directory tree. The RDNs are separated by commas and optional spaces. For example, in the first figure, the DN for the Ferrari entry is "o=Ferrari, c=Italy".

As with file system pathnames and URLs, entering the correct LDAP name format is essential to successful search operations.

Note: The RDN is an attribute of a directory entry. The full DN is not. However, you can output the full DN by specifying "dn" in a query's `attributes` list. For more information, see `cfldap` in *CFML Reference*. ColdFusion always returns DNs with spaces after the commas.

A *multivalued RDN* is made up of more than one attribute-value pair. In multivalued RDNs, the attribute-value pairs are separated by plus signs (+). In the sample directories, individuals could have complex RDNs consisting of their common name and their e-mail address; for example, "cn=Robert Boyd + mail=rjboyd@macromedia.com".

Schema

The concepts of schemas and object classes are central to a thorough understanding of LDAP. Although detailed descriptions of them are beyond the scope of this chapter, the following sections provide enough information to use the `cfldap` tag effectively.

A directory *schema* is a set of rules that determines what can be stored in a directory. It defines, at a minimum, the following two basic directory characteristics:

- The object classes to which entries can belong
- The directory attribute types

Object class

Object classes enable LDAP to group related information. Frequently, an object class corresponds to a real object or concept, such as a country, person, room, or domain (in fact, these are all standard object type names). Each entry in an LDAP directory must belong to one or more object classes.

The following characteristics define an object class:

- The class name
- A unique object ID that identifies the class
- The attribute types that entries of the class must contain
- The attribute types that entries of the class can optionally contain
- (Optional) A *superior* class from which the class is derived

If an entry belongs to a class that derives from another class, the entry's objectclass attribute lists the lowest-level class and all the superior classes from which the lowest-level class derives.

When you add, modify, or delete a directory entry, you must treat the entry's object class as a possibly multivalued attribute. For example, when you add a new entry, you specify the object class in the `cfldap` tag `attributes` attribute. To retrieve an entry's object class names, specify "objectclass" in the list of query attributes. To retrieve entries that provide a specific type of information, you can use the object class name in the `cfldap` tag `filter` attribute.

Attribute type

A schema's attribute type specification defines the following properties:

- The attribute type name
- A unique object ID that identifies the attribute type
- (Optional) An indication of whether the type is single-valued or multivalued (the default is multivalued)
- The attribute syntax and matching rules (such as case sensitivity)

The attribute type definition can also determine limits on the range or size of values that the type represents, or provide an application-specific usage indicator. For standard attributes, a registered numeric ID specifies the syntax and matching rule information. For more information on attribute syntaxes, see ETF RFC 2252 at http://www.ietf.org/rfc/rfc2252.txt.

Operational attributes, such as creatorsName or modifyTimeStamp, are managed by the directory service and cannot be changed by user applications.

Using LDAP with ColdFusion

The `cfldap` tag extends the ColdFusion query capabilities to LDAP network directory services. The `cfldap` tag lets you use LDAP in many ways, such as the following:

- Create Internet White Pages so users can locate people and resources and get information about them.
- Provide a front end to manage and update directory entries.
- Build applications that incorporate data from directory queries in their processes.
- Integrate applications with existing organizational or corporate directory services.

The `cfldap` tag `action` attribute supports the following operations on LDAP directories:

Action	Description
query	Returns attribute values from a directory.
add	Adds an entry to a directory.
modify	Adds, deletes, or changes the value of an attribute in a directory entry.
delete	Deletes an entry from a directory.
modifyDN	Renames a directory entry (changes its distinguished name).

The following table lists the attributes that are required and optional for each action. For more information on each attribute, see the `cfldap` tag in *CFML Reference*.

Action	Required attributes	Optional attributes
query	server, name, start, attributes	port, username, password, timeout, secure, rebind, referral, scope, filter, sort, sortControl startRow, maxRows, separator, delimiter
add	server, dn, attributes	port, username, password, timeout, secure, rebind, referral, separator, delimiter
modify	server, dn, attributes	port, username, password, timeout, secure, rebind, referral, modifyType, separator, delimiter
modifyDN	server, dn, attributes	port, username, password, timeout, secure, rebind, referral
delete	server, dn	port, username, password, timeout, secure, rebind, referral

Querying an LDAP directory

The `cfldap` tag lets you search an LDAP directory. The tag returns a ColdFusion query object with the results, which you can use as you would any query result. When you query an LDAP directory, you specify the directory entry where the search starts and the attributes whose values to return. You can specify the search scope and attribute content filtering rules and use other attributes to further control the search.

Scope

The search *scope* sets the limits of a search. The default scope is the level below the distinguished name specified in the `start` attribute. This scope does not include the entry identified by the `start` attribute. For example, if the `start` attribute is "ou=support, o=macromedia" the level below support is searched. You can restrict a query to the level of the `start` entry, or extend it to the entire subtree below the `start` entry.

Search filter

The search filter syntax has the form *attribute operator value.* The default filter, objectclass=*, returns all entries in the scope.

The following table lists the filter operators:

Operator	Example	Matches
=*	(mail=*)	All entries that contain a mail attribute.
=	(o=macromedia)	Entries in which the organization name is macromedia.
~=	(sn~=Hansen)	Entries with a surname that approximates Hansen. The matching rules for approximate matches vary among directory vendors, but anything that "sounds like" the search string should be matched. In this example, the directory server might return entries with the surnames Hansen and Hanson.

Operator	Example	Matches
>=	(st>=ma)	The name “ma” and names appearing after “ma” in an alphabetical state attribute list.
<=	(st<=ma)	The name “ma” and names appearing before “ma” in an alphabetical state attribute list.
*	(o=macro*)	Organization names that start with “macro”.
	(o=*media)	Organization names that end with “media”.
	(o=mac*ia)	Organization names that start with “mac” and end with “ia”. You can use more than one * operator in a string; for example, m*ro*dia.
	(o=*med*)	Organization names that contain the string “med”, including the exact string match “med”.
&	(&(o=macromedia)(co=usa))	Entries in which the organization name is “macromedia” and the country is “usa”.
|	(|(o=macromedia)(sn=macromedia)(cn=macromedia))	Entries in which the organization name is “macromedia” or the surname is “macromedia”, or the common name is “macromedia”.
!	(!(STREET=*))	Entries that do not contain a StreetAddress attribute.

The Boolean operators & and | can operate on more than two attributes and precede all of the attributes on which they operate. You surround a filter with parentheses and use parentheses to group conditions.

If the pattern that you are matching contains an asterisk, left parenthesis, right parenthesis, backslash, or NUL character, you must use the following three-character escape sequence in place of the character:

Character	Escape sequence
*	\2A
(	\28
)	\29
\	\5C
NUL	\00

For example, to match the common name St*r Industries, use the filter (cn=St\2Ar Industries).

LDAP v3 supports an extensible match filter that permits server-specific matching rules. For more information on using extensible match filters, see your LDAP server documentation.

Searching and sorting notes

- To search for multiple values of a multivalued attribute type, use the & operator to combine expressions for each attribute value. For example, to search for an entry in which cn=Robert Jones and cn=Bobby Jones, specify the following filter:

  ```
  filter="(&(cn=Robert Jones)(cn=Bobby Jones))"
  ```

- You can use object classes as search filter attributes; for example, you can use the following search filter:

  ```
  filter="(objectclass=inetorgperson)"
  ```

- To specify how query results are sorted, use the `sort` field to identify the attribute(s) to sort. By default, ColdFusion returns sorted results in case-sensitive ascending order. To specify descending order, case-insensitive sorting, or both, use the `sortControl` attribute.
- ColdFusion requests the LDAP server to do the sorting. This can have the following effects:
  - The sort order might differ between ColdFusion MX and previous versions.
  - If you specify sorting and the LDAP server does not support sorting, ColdFusion generates an error. To sort results from servers that do not support sorting, use a query of queries on the results.
- If you use filter operators to construct sophisticated search criteria, performance might degrade if the LDAP server is slow to process the synchronous search routines that `cfldap` supports. You can use the `cfldap` tag `timeout` and `maxRows` attributes to control the apparent performance of pages that perform queries, by limiting the number of entries and by exiting the query if the server does not respond in a specified time.

Getting all the attributes of an entry

Typically, you do not use a query that gets all the attributes in an entry. Such a query would return attributes that are used only by the directory server. However, you can get all the attributes by specifying attributes="*" in your query.

If you do this, ColdFusion returns the results in a structure in which each element contains a single attribute name-value pair. The tag does *not* return a query object. ColdFusion does this because LDAP directory entries, unlike the rows in a relational table, vary depending on their object class.

For example, the following code retrieves the contents of the Airius directory:

```
<cfldap name="GetList"
  server=#myServer#
  action="query"
  attributes="*"
  scope="subtree"
  start="o=airius.com"
  sort="sn,cn">
```

This tag returns entries for all the people in the organization and entries for all the groups. The group entries have a different object class, and therefore different attributes from the person entries. If ColdFusion returned both types of entries in one query object, some rows would have only the group-specific attribute values and the other rows would have only person-specific attribute values. Instead, ColdFusion returns a structure in which each attribute is an entry.

Example: querying an LDAP directory

The following example uses the `cfldap` tag to get information about the people in the Airius corporation's Santa Clara office. Users can enter all or part of a person's name and get a list of matching names with their departments, e-mail addresses, and telephone numbers.

This example uses the sample Airius corporate directory that is distributed with the Netscape Directory Server. If you do not have access to this directory, modify the code to work with your LDAP directory.

To query an LDAP directory:

1. Create a file that looks like the following:

```
<!--- This example shows the use of CFLDAP --->
<html>
<head> <title>cfldap Query Example</title> </head>

<h3>cfldap Query Example</h3>

<body>
<p>This tool queries the Airius.com database to locate all people in
  the company's Santa Clara office whose common names contain the
  text entered in the form.</p>

<p>Enter a full name, first name, last name, or name fragment.</p>

<form action="cfldap.cfm" method="POST">
  <input type="text" name="name"><br><br>
  <input type="submit" value="Search">
</form>

<!--- make the LDAP query --->
<!-- Note that some search text is required.
  A search filter of cn=** would cause an error -->
<cfif (isdefined("form.name") AND (form.name IS NOT ""))>
  <cfldap
    server="ldap.airius.com"
    action="query"
    name="results"
    start="ou=People, o=Airius.com"
    scope="onelevel"
    filter="(&(cn=*#form.Name#*)(l=Santa Clara))"
    attributes="cn,sn,ou,mail,telephonenumber"
    sort="ou,sn"
    maxrows=100
    timeout=20000
  >
```

```
<!--- Display results --->
  <table border=0 cellspacing=2 cellpadding=2>
   <tr>
       <th colspan=4><cfoutput>#results.RecordCount# matches found</
  cfoutput>
       </th>
   </tr>
   <tr>
       <th>Name</th>
       <th>Department</th>
       <th>E-Mail</th>
       <th>Phone</th>
   </tr>
  <cfoutput query="results">
   <tr>
       <td>#cn#</td>
       <td>#listFirst(ou)#</td>
       <td><a href="mailto:#mail#">#mail#</a></td>
       <td>#telephonenumber#</td>
   </tr>
  </cfoutput>
  </table>
</cfif>

</body>
</html>
```

2. Change the `server` attribute from ldap.airius.com to the name of your installation of the Airius database.
3. Save the page as `cfldap.cfm` and run it in your browser.

Reviewing the code

The following table describes the code:

Code	Description
`<form action="cfldap.cfm" method="POST">` `<input type="text" name="name">` `  ` `<input type="submit" value="Search">` `</form>`	Uses a form to get the name or name fragment to search for.
`<cfif (isdefined("form.name")` `AND (form.name IS NOT ""))>`	Ensures that the user has submitted the form. This is necessary because the form page is also the action page. Ensures that the user entered search text.

Code	Description

<cfldap
  server="ldap.airius.com"
  action="query"
  name="results"
  start="ou=People, o=Airius.com"
  scope="onelevel"
 	
  filter="(&(cn=*#form.Name#*)
  (l=Santa Clara))"
 	
  attributes="cn,sn,ou,mail,
  telephonenumber"
 	
  sort="ou,sn"
 	
  maxrows=100
  timeout=20000
>
	Connects anonymously to LDAP server ldap.airius.com, query the directory, and return the results to a query object named results.
	Starts the query at the directory entry that has the distinguished name ou=People, o=Airius.com, and searches the directory level immediately below this entry.
	Requests records for entries that contain the location (l) attribute value "Santa Clara" and the entered text in the common name attribute.
	Gets the common name, surname, organizational unit, e-mail address, and telephone number for each entry.
	Sorts the results first by organization name, then by surname. Sorts in the default sorting order.
	Limit the request to 100 entries. If the server does not return the data in 20 seconds, generates an error indicating that LDAP timed out.

<table border=0 cellspacing=2
  cellpadding=2>
  <tr>
  <th colspan=4>
  <cfoutput>#results.RecordCount#
  matches found</cfoutput>
  </th>
  </tr>
  <tr>
  <th>Name</th>
  <th>Department</th>
  <th>E-Mail</th>
  <th>Phone</th>
  </tr>
  <cfoutput query="results">
  <tr>
  <td>#cn#</td>
  <td>#ListFirst(ou)#</td>
  <td>#mail#
  </td>
  <td>#telephonenumber#</td>
  </tr>
  </cfoutput>
  </table>
</cfif>
	Starts a table to display the output
	Displays the number of records returned.
	Displays the common name, department, e-mail address, and telephone number of each entry.
	Displays only the first entry in the list of organizational unit values. (For more information, see the description that follows this table.)

This search shows the use of a logical AND statement in a filter. It returns one attribute, the surname, that is used only for sorting the results.

In this query, the `ou` attribute value consists of two values in a comma-delimited list. One is the department name. The other is People. This is because the Airius database uses the `ou` attribute type twice:

- In the distinguished names, at the second level of the directory tree, where it differentiates between organizational units such as people, groups, and directory servers
- As the department identifier in each person's entry

Because the attribute values are returned in order from the person entry to the directory tree root, the `ListFirst` function extracts the person's department name.

Updating an LDAP directory

The `cfldap` tag lets you do the following to LDAP directory entries:

- Add
- Delete
- Add attributes
- Delete attributes
- Replace attributes
- Change the DN (rename the entry)

These actions let you manage LDAP directory contents remotely.

The following sections show how to build a ColdFusion page that lets you manage an LDAP directory:

- "Adding a directory entry" on page 528
- "Deleting a directory entry" on page 533
- "Updating a directory entry" on page 535

The form displays directory entries in a table and includes a button that lets you populate the form fields based on the unique user ID.

The example ColdFusion page does not add or delete entry attributes or change the DN. The sections "Adding and deleting attributes of a directory entry" on page 537 and "Changing a directory entry's DN" on page 538 describe these operations.

To keep the code short, this example has limitations that are not appropriate in a production application. In particular, it has the following limitations:

- If you enter an invalid user ID and click either the Update or the Delete button, ColdFusion generates a "No such object" error, because there is no directory entry to update or delete. Your application should verify that the ID exists in the directory before it tries to change or delete its entry.
- If you enter a valid user ID in an empty form and click Update, the application deletes all the attributes for the User. The application should ensure that all required attribute fields contain valid entries before updating the directory.

Adding a directory entry

When you add an entry to an LDAP directory, you specify the DN, all the required attributes, including the entry's object class, and any optional attributes. The following example builds a form that adds an entry to an LDAP directory.

To add an entry:

1. Create a file that looks like the following:

```
<!--- set the LDAP server ID, user name, and password as variables
  here so they can be changed in only one place --->
<cfset myServer="ldap.myco.com">
<cfset myUserName="cn=Directory Manager">
<cfset myPassword="password">

<!--- Initialize the values used in form fields to empty strings --->
<cfparam name="fullNameValue" default="">
<cfparam name="surnameValue" default="">
<cfparam name="emailValue" default="">
<cfparam name="phoneValue" default="">
<cfparam name="uidValue" default="">

<!---When the form is submitted, add the LDAP entry --->
<cfif isdefined("Form.action") AND Trim(Form.uid) IS NOT "">
  <cfif Form.action is "add">
    <cfif Trim(Form.fullName) is "" OR Trim(Form.surname) is ""
      OR Trim(Form.email) is "" OR Trim(Form.phone) is "">
        <h2>You must enter a value in every field.</h2>
        <cfset fullNameValue=Form.fullName>
        <cfset surnameValue=Form.surname>
        <cfset emailValue=Form.email>
        <cfset phoneValue=Form.phone>
        <cfset uidValue=Form.uid>
    <cfelse>
      <cfset attributelist="objectclass=top, person,
          organizationalperson, inetOrgPerson;
        cn=#Trim(Form.fullName)#; sn=#Trim(Form.surname)#;
        mail=#Trim(Form.email)#;
        telephonenumber=#Trim(Form.phone)#;
        ou=Human Resources;
        uid=#Trim(Form.uid)#">
      <cfldap action="add"
        attributes="#attributeList#"
        dn="uid=#Trim(Form.uid)#, ou=People, o=Airius.com"
        server=#myServer#
        username=#myUserName#
        password=#myPassword#>
      <cfoutput><h3>Entry for User ID #Form.uid# has been added</h3>
      </cfoutput>
    </cfif>
  </cfif>
</cfif>

<html>
<head>
```

```
  <title>Update LDAP Form</title>
</head>
<body>
<h2>Manage LDAP Entries</h2>

<cfform action="update_ldap.cfm" method="post">
  <table>
    <tr><td>Full Name:</td>
      <td><cfinput type="Text"
          name="fullName"
          value=#fullNameValue#
          size="20"
          maxlength="30"
          tabindex="1"></td>
    </tr>
    <tr><td>Surname:</td>
      <td><cfinput type="Text"
        name="surname"
        Value= "#surnameValue#"
        size="20"
        maxlength="20"
        tabindex="2"></td>
    </tr>
    <tr>
      <td>E-mail Address:</td>
      <td><cfinput type="Text"
        name="email"
        value="#emailValue#"
        size="20"
        maxlength="20"
        tabindex="3"></td>
    </tr>
    <tr>
      <td>Telephone Number:</td>
      <td><cfinput type="Text"
        name="phone"
        value="#phoneValue#"
        size="20"
        maxlength="20"
        tabindex="4"></td>
    </tr>
    <tr>
      <td>User ID:</td>
      <td><cfinput type="Text"
        name="uid"
        value="#uidValue#"
        size="20"
        maxlength="20"
        tabindex="5"></td>
    </tr>
    <tr>
      <td colspan="2">
        <input type="Submit"
         name="action"
         value="Add"
```

```
            tabindex="8"></td>
        </tr>
    </table>
    <br>
    *All fields are required for Add<br>
</cfform>

<!---Output the user list --->
<h2>User List for the Human Resources Department</h2>
<cfldap name="GetList"
    server=#myServer#
    action="query"
    attributes="cn,sn,mail,telephonenumber,uid"
    start="o=Airius.com"
    scope="subtree"
    filter="ou=Human Resources"
    sort="sn,cn"
    sortControl="asc, nocase">

<table border="1">
    <tr>
        <th>Full Name</th>
        <th>Surname</th>
        <th>Mail</th>
        <th>Phone</th>
        <th>UID</th>
    </tr>
    <cfoutput query="GetList">
    <tr>
        <td>#GetList.cn#</td>
        <td>#GetList.sn#</td>
        <td>#GetList.mail#</td>
        <td>#GetList.telephonenumber#</td>
        <td>#GetList.uid#</td>
    </tr>
    </cfoutput>
</table>
</body>
</html>
```

2. At the top of the file, change the myServer, myUserName, and myPassword variable assignments to values that are valid for your LDAP server.
3. Save the page as update_ldap.cfm and run it in your browser.

Reviewing the code

The following table describes the code:

Code	Description

`<cfset myServer="ldap.myco.com">`
`<cfset myUserName="cn=Directory Manager">`
`<cfset myPassword="password">`
	Initializes the LDAP connection information variables. Uses variables for all connection information so that any changes have to be made in only one place.

`<cfparam name="fullNameValue" default="">`
`<cfparam name="surnameValue" default="">`
`<cfparam name="emailValue" default="">`
`<cfparam name="phoneValue" default="">`
`<cfparam name="uidValue" default="">`
	Sets the default values of empty strings for the form field value variables. The data entry form uses `cfinput` fields with `value` attributes so that the form can be prefilled and so that, if the user submits an incomplete form, ColdFusion can retain any entered values in the form when it redisplays the page.

`<cfif isdefined("Form.action") AND Trim(Form.uid) IS NOT "">`	Ensures that the user entered a User ID in the form.
`<cfif Form.action is "add">`	If the user clicks Add, processes the code that follows.

`<cfif Trim(Form.fullName) is ""`
`OR Trim(Form.surname) is ""`
`OR Trim(Form.email) is ""`
`OR Trim(Form.phone) is "">`
`<h2>You must enter a value in every field.</h2>`
`<cfset fullNameValue=Form.fullName>`
`<cfset surnameValue=Form.surname>`
`<cfset emailValue=Form.email>`
`<cfset phoneValue=Form.phone>`
`<cfset uidValue=Form.uid>`
	If any field in the submitted form is blank, display a message and set the other form fields to display data that the user submitted.

`<cfelse>`
`<cfset attributelist="objectclass=top,person,organizationalperson,inetOrgPerson;`
`cn=#Trim(Form.fullName)#;`
`sn=#Trim(Form.surname)#;`
`mail=#Trim(Form.email)#;`
`telephonenumber=#Trim(Form.phone)#;`
`ou=Human Resources;`
`uid=#Trim(Form.uid)#">`
	If the user entered data in all fields, sets the attributelist variable to specify the entry's attributes, including the object class and the organizational unit (in this case, Human Resources).
	The `Trim` function removes leading or trailing spaces from the user data.

Code	Description
<cfldap action="add" attributes="#attributeList#" dn="uid=#Trim(Form.uid)#, ou=People, o=Airius.com" server=#myServer# username=#myUserName# password=#myPassword#> <cfoutput> Entry for User ID #Form.uid# has been added </cfoutput> </cfif> </cfif> </cfif>	Adds the new entry to the directory.
<cfform action="update_ldap.cfm" method="post"> <table> <tr><td>Full Name:</td> <td><cfinput type="Text" name="fullName" value=#fullNameValue# size="20" maxlength="30" tabindex="1"></td> </tr> . . . <tr><td colspan="2"> <input type="Submit" name="action" value="Add" tabindex="6"></td> </tr> </table> *All fields are required for Add </cfform>	Outputs the data entry form, formatted as a table. Each `cfinput` field always has a value, set by the `value` attribute when the page is called. The `value` attribute lets ColdFusion update the form contents when the form is redisplayed after the user clicks Add. The code that handles cases in which the user fails to enter all the required data uses this feature.

Code	Description
<cfldap name="GetList" server=#myServer# action="query" attributes="cn,sn,mail, telephonenumber,uid" start="o=Airius.com" scope="subtree" filter="ou=Human Resources" sort="sn,cn" sortControl="asc, nocase">	Queries the directory and gets the common name, surname, e-mail address, telephone number, and user ID from the matching entries. Searches the subtree from the entry with the DN of o=Airius.com, and selects all entries in which the organizational unit is Human Resources. Sorts the results by surname and then common name (to sort by last name, then first). Sorts in default ascending order that is not case-sensitive.
<table border="1"> <tr> <th>Full Name</th> <th>Surname</th> <th>Mail</th> <th>Phone</th> <th>UID</th> </tr> <cfoutput query="GetList"> <tr> <td>#GetList.cn#</td> <td>#GetList.sn#</td> <td>#GetList.mail#</td> <td>#GetList.telephonenumber#</td> <td>#GetList.uid#</td> </tr> </cfoutput> </table> </body> </html>	Display the query results in a table.

Deleting a directory entry

To delete a directory entry, you must specify the entry DN.

The following example builds on the code that adds an entry. It adds Retrieve and Delete buttons. The Retrieve button lets you view a user's information in the form before you delete it.

To delete an entry:

1. Open update_ldap.cfm, which you created in "Adding a directory entry" on page 528.
2. Between the first and second `</cfif>` tags, add the following code:

```
<cfelseif Form.action is "Retrieve">
  <cfldap name="GetEntry"
    server=#myServer#
    action="query"
    attributes="cn,sn,mail,telephonenumber,uid"
    scope="subtree"
    filter="uid=#Trim(Form.UID)#"
    start="o=Airius.com">
  <cfset fullNameValue = GetEntry.cn[1]>
  <cfset surnameValue = GetEntry.sn[1]>
  <cfset emailValue = GetEntry.mail[1]>
  <cfset phoneValue = GetEntry.telephonenumber[1]>
```

```
    <cfset uidValue = GetEntry.uid[1]>
  <cfelseif Form.action is "Delete">
    <cfldap action="delete"
      dn="uid=#Trim(Form.UID)#, ou=People, o=Airius.com"
      server=#myServer#
      username=#myUserName#
      password=#myPassword#>
    <cfoutput><h3>Entry for User ID #Form.UID# has been deleted
    </h3></cfoutput>
```

3. At the end of the code for the Add button (the `input` tag with `Value=Add` at the bottom of the form), delete the `</td>` end tag.
4. After the end of the Add button `input` tag, add the following code:

```
 
<input type="Submit"
   name="action"
   value="Retrieve"
   tabindex="7">
 
<input type="Submit"
   name="action"
   value="Delete"
   tabindex="8"></td>
```

5. Save the file and run it in your browser.

Reviewing the code

The following table describes the code:

Code	Description

`<cfelseif Form.action is "Retrieve">`
`  <cfldap name="GetEntry"`
`  server=#myServer#`
`  action="query"`
`  attributes="cn,sn,mail,`
`  telephonenumber,uid"`
`  scope="subtree"`
`  filter="uid=#Trim(Form.UID)#"`
`  start="o=Airius.com">`
`  <cfset fullNameValue=`
`  GetEntry.cn[1]>`
`  <cfset surnameValue=GetEntry.sn[1]>`
`  <cfset emailValue=GetEntry.mail[1]>`
`  <cfset phoneValue=`
`  GetEntry.telephonenumber[1]>`
`  <cfset uidValue=GetEntry.uid[1]>`
	If the user clicks Retrieve, queries the directory and gets the information for the specified User ID.
	Sets the form field's `Value` attribute to the corresponding query column.
	This example uses the array index [1] to identify the first row of the GetEntry query object. Because the query always returns only one row, the index can be omitted.

`<cfelseif Form.action is "Delete">`
`  <cfldap action="delete"`
`  dn="uid=#Trim(Form.UID)#,`
`  ou=People, o=Airius.com"`
`  server=#myServer#`
`  username=#myUserName#`
`  password="password">`
`  <cfoutput><h3>Entry for User`
`  ID #Form.UID# has been`
`  deleted</h3></cfoutput>`
	The user clicks delete, deletes the entry with the specified User ID and informs the user that the entry was deleted.

` `
`<input type="Submit"`
`  name="action"`
`  value="Retrieve"`
`  tabindex="7">`
` `
`<input type="Submit"`
`  name="action"`
`  value="Delete"`
`  tabindex="8"></td>`
	Displays submit buttons for the Retrieve and Delete actions.

Updating a directory entry

The `cfldap` tag lets you change the values of entry attributes. To do so, you specify the entry DN in the `dn` attribute, and list the attributes to change and their new values in the `attributes` attribute.

The following example builds on the code that adds and deletes an entry. It can update one or more of an entry's attributes. Because the UID is part of the DN, you cannot change it.

To update an entry:

1. Open update_ldap.cfm, which you created in "Adding a directory entry" on page 528.
2. Between the `cfelseif Form.action is "Retrieve"` block and the `</cfif>` tag, add the following code:

```
<cfelseif Form.action is "Update">
<cfset attributelist="cn=#Trim(form.FullName)#; sn=#Trim(Form.surname)#;
  mail=#Trim(Form.email)#;
  telephonenumber=#Trim(Form.phone)#">
<cfldap action="modify"
  modifytype="replace"
  attributes="#attributeList#"
  dn="uid=#Trim(Form.UID)#, ou=People, o=Airius.com"
  server=#myServer#
  username=#myUserName#
  password=#myPassword#>
<cfoutput><h3>Entry for User ID #Form.UID# has been updated</h3>
</cfoutput>
```

3. At the end of the code for the Delete button (the `input` tag with `Value=Delete)` at the bottom of the form`)`, delete the `</td>` mark.
4. After the end of the Delete button `input` tag, add the following code:

```
 
<input type="Submit"
  name="action"
  value="Update"
  tabindex="9"></td>
```

5. Save the file and run it in your browser.

Reviewing the code

The following table describes the code:

Code	Description

`<cfelseif Form.action is "Update">`
`  <cfset attributelist="cn=#Trim`
`  (form.FullName)#;`
`  sn=#Trim(Form.surname)#;`
`  mail=#Trim(Form.email)#;`
`  telephonenumber=#Trim(Form.phone)#">`
`  <cfldap action="modify"`
`  modifytype="replace"`
`  attributes="#attributeList#"`
`  dn="uid=#Trim(Form.UID)#,`
`  ou=People, o=Airius.com"`
`  server=#myServer#`
`  username=#myUserName#`
`  password=#myPassword#>`
`  <cfoutput><h3>Entry for User ID`
`  #Form.UID# has been updated</h3>`
`  </cfoutput>`
	If the user clicks Update, sets the attribute list to the form field values and replaces the attributes for the entry with the specified UID. Displays a message to indicate that the entry was updated.
	This code replaces *all* of the attributes in a form, without checking whether they are blank. A more complete example would check for blank fields and either require entered data or not include the corresponding attribute in the `attributes` string.

` `
`<input type="Submit"`
`  name="action"`
`  value="Update"`
`  tabindex="9"></td>`
	Defines the Submit button for the update action.

Adding and deleting attributes of a directory entry

The following table lists the `cfldap` tag attributes that you must specify to add and delete LDAP attributes in an entry:

Action	cfldap syntax

Add attribute to entry	`dn = "entry dn"`
	`action = "modify"`
	`modifyType = "add"`
	`attributes = "attribname=attribValue[;...]"`

Delete attribute from entry	`dn = "entry dn"`
	`action = "modify"`
	`modifyType = "delete"`
	`attributes = "attribName[;...]"`

You can add or delete multiple attributes in one statement. To do this, use semicolons to separate the attributes in the attribute string.

The following example specifies the description and seealso LDAP attributes:

```
attributes="description=Senior Technical Writer;seealso=writers"
```

You can change the character that you use to separate values of multivalued attributes in an attribute string. You can also change the character that separates attributes when a string contains multiple attributes. For more information, see "Specifying an attribute that includes a comma or semicolon" on page 538.

You can add or delete attributes only if the directory schema defines them as optional for the entry's object class.

Changing a directory entry's DN

To change the DN of an entry, you must provide the following information in the `cfldap` tag:

```
dn="original DN"
action="modifyDN"
attributes="dn=new DN"
```

For example:

```
<cfldap action="modifyDN"
  dn="#old_UID#, ou=People, o=Airius.com"
  attributes="uid=#newUID#"
  server=#myServer#
  username=#myUserName#
  password=#myPassword#>
```

The new DN and the entry attributes must conform to the directory schema; therefore, you cannot move entries arbitrarily in a directory tree. You can only modify a leaf only. For example, you cannot modify the group name if the group has children.

Note: LDAP v2 does not let you change entry DNs.

Advanced topics

The following sections present advanced topics that enable you to use LDAP directories more effectively.

Specifying an attribute that includes a comma or semicolon

LDAP attribute values can contain commas. The `cfldap` tag normally uses commas to separate attribute values in a value list. Similarly, an attribute can contain a semicolon, which `cfldap` normally uses to delimit (separate) attributes in an attribute list. To override the default separator and delimiter characters, you use the `cfldap` tag `separator` and `delimiter` attributes.

For example, assume you want to add the following attributes to an LDAP entry:

```
cn=Proctor, Goodman, and Jones
description=Friends of the company; Rationalists
```

Use the `cfldap` tag in the following way:

```
<cfldap action="modify"
  modifyType="add"
  attributes="cn=Proctor, Goodman, and Jones: description=Friends
  of the company; Rationalists"
  dn="uid=goodco, ou=People, o=Airius.com"
  separator="&"
  delimiter=":"
  server=#myServer#
  username=#myUserName#
  password=#myPassword#>
```

Using cfldap output

You can create a searchable Verity collection from LDAP data. For an example of building a Verity collection using an LDAP directory, see Chapter 24, "Indexing query results obtained from an LDAP directory," on page 578.

The ability to generate queries from other queries is very useful when `cfldap` queries return complex data. For more information on querying queries, see Chapter 22, "Using Query of Queries," on page 489.

Viewing a directory schema

LDAP v3 exposes a directory's schema information in a special entry in the root DN. You use the directory root subschemaSubentry attribute to access this information.

The following ColdFusion query shows how to get and display the directory schema. It displays information from the schema's object class and attribute type definitions. For object classes, it displays the class name, superior class, required attribute types, and optional attribute types. For attribute types, it displays the type name, type description, and whether the type is single- or multivalued.

The example does not display all the information in the schema. For example, it does not display the matching rules. It also does not display the object class IDs, attribute type IDs, attribute type syntax IDs, or the object class descriptions. (The object class description values are all "Standard Object Class.")

Note: To be able to view the schema for an LDAP server, the server must support LDAP v3.

This example does not work on iPlanet Directory Server 5.0. It does work on a 4.x server.

To view the schema for an LDAP directory:

1. Create a new file that looks like the following:

```
<html>
<head>
  <title>LDAP Schema</title>
</head>

<body>
<!--- Start at Root DSE to get the subschemaSubentry attribute --->
<cfldap
  name="EntryList"
  server="ldap.mycorp.com"
  action="query"
  attributes="subschemaSubentry"
  scope="base"
  start="">

<!--- Use the DN from the subschemaSubEntry attribute to get the schema --->
<cfldap
  name="EntryList2"
  server="ldap.mycorp.com"
  action="query"
  attributes="objectclasses, attributetypes"
```

```
    scope="base"
    filter="objectclass=*"
    start=#entryList.subschemaSubentry#>

<!--- Only one record is returned, so query loop is not required --->
<h2>Object Classes</h2>
<table border="1">
  <tr>
    <th>Name</th>
    <th>Superior class</th>
    <th>Must have</th>
    <th>May have</th>
  </tr>
  <cfloop index = "thisElement" list = #Entrylist2.objectclasses#>
    <cfscript>
      thiselement = Trim(thisElement);
      nameloc = Find("NAME", thisElement);
      descloc = Find("DESC", thisElement);
      suploc = Find("SUP", thisElement);
      mustloc = Find("MUST", thisElement);
      mayloc = Find("MAY", thisElement);
      endloc = Len(thisElement);
    </cfscript>
    <tr>
      <td><cfoutput>#Mid(thisElement, nameloc+6, descloc-nameloc-8)#
        </cfoutput></td>
      <cfif #suploc# NEQ 0>
        <td><cfoutput>#Mid(thisElement, suploc+5, mustloc-suploc-7)#
        </cfoutput></td>
      <cfelse>
        <td>NONE</td>
      </cfif>
      <cfif #mayloc# NEQ 0>
        <td><cfoutput>#Replace(Mid(thisElement, mustloc+6,
          mayloc-mustloc-9), " $ ", ", ", "all")#</cfoutput></td>
        <td><cfoutput>#Replace(Mid(thisElement, mayloc+5, endloc-mayloc-
8),
          " $ ", ", ", "all")#</cfoutput></td>
      <cfelse>
        <td><cfoutput>#Replace(Mid(thisElement, mustloc+6,
          endloc-mustloc-9), " $ ", ", ", "all")#</cfoutput></td>
        <td>NONE</td>
      </cfif>
    </tr>
  </cfloop>
</table>
<br><br>

  <h2>Attribute Types</h2>
<table border="1" >
  <tr>
    <th>Name</th>
    <th>Description</th>
    <th>multivalued?</th>
  </tr>
```

```
    <cfloop index = "thisElement"
      list = #ReplaceNoCase(EntryList2.attributeTypes, ", alias", "<br>
    Alias",
      "all")# delimiters = ",">
      <cfscript>
        thiselement = Trim(thisElement);
        nameloc = Find("NAME", thisElement);
        descloc = Find("DESC", thisElement);
        syntaxloc = Find("SYNTAX", thisElement);
        singleloc = Find("SINGLE", thisElement);
        endloc = Len(thisElement);
      </cfscript>
      <tr>
        <td><cfoutput>#Mid(thisElement, nameloc+6, descloc-nameloc-8)#
          </cfoutput></td>
        <td><cfoutput>#Mid(thisElement, descloc+6, syntaxloc-descloc-8)#
          </cfoutput></td>
        <cfif #singleloc# EQ 0>
           <td><cfoutput>Yes</cfoutput></td>
        <cfelse>
           <td><cfoutput>No</cfoutput></td>
        </cfif>
      </tr>
    </cfloop>
  </table>
  </body>
  </html>
```

2. Change the server from `ldap.mycorp.com` to your LDAP server. You might also need to specify a user ID and password in the `cfldap` tag.
3. Save the template as `ldapschema.cfm` in `myapps` under your web root directory and view it in your browser.

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfldap`
`  name="EntryList"`
`  server="ldap.mycorp.com"`
`  action="query"`
`  attributes="subschemaSubentry"`
`  scope="base"`
`  start="">`
	Gets the value of the subschemaSubentry attribute from the root of the directory server. The value is the DN of the schema.

`<cfldap`
`  name="EntryList2"`
`  server="ldap.mycorp.com"`
`  action="query"`
`  attributes="objectclasses,`
`  attributetypes"`
`  scope="base"`
`  filter="objectclass=*"`
`  start=#entryList.subschemaSubentry#>`
	Uses the schema DN to get the objectclasses and attributetypes attributes from the schema.

Code	Description

Object Classes
<table border="1">
<tr>
<th>Name</th>
<th>Superior class</th>
<th>Must have</th>
<th>May have</th>
</tr>
<cfloop index = "thisElement" list = #Entrylist2.objectclasses#>
<cfscript>
thisElement = Trim(thisElement);
nameloc = Find("NAME", thisElement);
descloc = Find("DESC", thisElement);
suploc = Find("SUP", thisElement);
mustloc = Find("MUST", thisElement);
mayloc = Find("MAY", thisElement);
endloc = Len(thisElement);
</cfscript>
	Displays the object class name, superior class, required attributes, and optional attributes for each object class in a table.
	The schema contains the definitions of all object classes in a comma delimited list, so the code uses a list type `cfloop` tag.
	The thisElement variable contains the object class definition. Trim off any leading or trailing spaces, then use the class definition field keywords in `Find` functions to get the starting locations of the required fields, including the Object class ID. (The ID is not displayed.)
	Gets the length of the thisElement string for use in later calculations.

<tr> <td><cfoutput>#Mid(thisElement, nameloc+6, descloc-nameloc-8)#</cfoutput></td> <cfif #suploc# NEQ 0> <td><cfoutput>#Mid(thisElement, suploc+5, mustloc-suploc-7)#</cfoutput></td> <cfelse> <td>NONE</td> </cfif> <cfif #mayloc# NEQ 0> <td><cfoutput>#Replace (Mid(thisElement, mustloc+6, mayloc-mustloc-9), " $ ", ", ", "all")#</cfoutput></td> <td><cfoutput>#Replace (Mid(thisElement, mayloc+5, endloc-mayloc-8), " $ ", ", ", "all")#</cfoutput></td> <cfelse> <td><cfoutput>#Replace (Mid(thisElement, mustloc+6, endloc-mustloc-9), " $ ", ", ", "all")#</cfoutput></td> <td>NONE</td> </cfif> </tr> </cfloop> </table>	Displays the field values. Uses the `Mid` function to extract individual field values from the thisElement string. The top object class does not have a superior class entry. Handles this special case by testing the suploc location variable. If the value is not 0, handles normally, otherwise, output "NONE". There might not be any optional attributes. Handles this case similarly to the superior class. The calculation of the location of required attributes uses the location of the optional attributes if the field exists; otherwise, uses the end of the object class definition string.

Code	Description
Attribute Types <table border="1" > <tr> <th>Name</th> <th>Description</th> <th>Multivalued?</th> </tr> <cfloop index = "thisElement" list = #ReplaceNoCase(attributeTypes, ", alias", " Alias", "all")# delimiters = ","> <cfscript> thiselement = Trim(thisElement); nameloc = Find("NAME", thisElement); descloc = Find("DESC", thisElement); syntaxloc = Find("SYNTAX", thisElement); singleloc = Find("SINGLE", thisElement); endloc = Len(thisElement); </cfscript> <tr> <td><cfoutput>#Mid(thisElement, nameloc+6, descloc-nameloc-8)# </cfoutput></td> <td><cfoutput>#Mid(thisElement, descloc+6, syntaxloc-descloc-8) #</cfoutput></td> <cfif #singleloc# EQ 0> <td><cfoutput>Yes</cfoutput> </td> <cfelse> <td><cfoutput>No</cfoutput> </td> </cfif> </tr> </cfloop> </table> </cfloop>	Does the same types of calculations for the attribute types as for the object classes. The attribute type field can contain the text ", alias for....". This text includes a comma, which also delimits attribute entries. Use the `ReplaceNoCase` function to replace any comma that precedes the word "alias" with an HTML ` ` tag. The attribute definition includes a numeric syntax identifier, which the code does not display, but uses its location in calculating the locations of the other fields.

Referrals

An LDAP database can be distributed over multiple servers. If the requested information is not on the current server, the LDAP v3 standard provides a mechanism for the server to return a referral to the client that informs the client of an alternate server. (This feature is also included in some LDAP v2-compliant servers.)

ColdFusion can handle referrals automatically. If you specify a nonzero `referral` attribute in the `cfldap` tag, ColdFusion sends the request to the server specified in the referral.

The `referral` attribute value specifies the number of referrals allowed for the request. For example, if the `referral` attribute is 1, and server A sends a referral to server B, which then sends a referral to server C, ColdFusion returns an error. If the `referral` attribute is 2, and server C has the information, the LDAP request succeeds. The value to use depends on the topology of the distributed LDAP directory, the importance of response speed, and the value of response completeness.

When ColdFusion follows a referral, the `rebind` attribute specifies whether ColdFusion uses the `cfldap` tag login information in the request to the new server. The default, No, sends an anonymous login to the server.

Managing LDAP security

When you consider how to implement LDAP security, you must consider server security and application security.

Server security

The `cfldap` tag supports secure socket layer (SSL) v2 security. This security provides certificate-based validation of the LDAP server. It also encrypts data transferred between the ColdFusion server and the LDAP server, including the user password, and ensures the integrity of data passed between the servers. To specify SSL v2 security, set the `cfladap` tag `secure="cfssl_basic"` attribute.

About LDAP Server Security

ColdFusion MX uses Java Native Directory Interface (JNDI), the LDAP provider, and an SSL package to create the client side of an SSL communication. The LDAP server provides the server side. The LDAP server that the `cfldap` tag connects to using SSL holds an SSL server certificate, a certificate that is securely "signed" by a trusted authority and identifies (authenticates) the sender. During the initial SSL connection, the LDAP server presents its server certificate to the client. If the client trusts this certificate, the SSL connection is established and secure LDAP communication can begin.

ColdFusion determines whether to trust the server by comparing the server's certificate with the information in the jre/lib/security/cacerts keystore of the JRE used by ColdFusion MX. The ColdFusion MX default cacerts file contains information about many certificate granting authorities. If you must update the file with additional information, you can use the keytool utility in the ColdFusion jre/bin directory to import certificates that are in X.509 format. For example, enter the following:

```
keytool -import -keystore cacerts -alias ldap -file ldap.crt -keypass bl19mq
```

The keytool utility initial keypass password is "change it". For more information on using the keytool utility, see the Sun JDK documentation.

Once ColdFusion establishes secure communication with the server, it must provide the server with login credentials. You specify the login credentials in the `cfldap` tag `username` and `password` attributes. When the server determines that the login credentials are valid, ColdFusion can access the directory.

Using LDAP security

To use security, first ensure that the LDAP server supports SSL v2 security.

Specify the `cfldap` tag `secure` attribute as follows:

```
secure = "cfssl_basic"
```

For example:

```
<cfldap action="modify"
  modifyType="add"
  atributes="cn=Lizzie"
  dn="uid=lborden, ou=People, o=Airius.com"
  server=#myServer#
  username=#myUserName#
  password=#myPassword#
  secure="cfssl_basic"
  port=636>
```

The `port` attribute specifies the server port used for secure LDAP communications, which is 636 by default. If you do not specify a port, ColdFusion attempts to connect to the default, nonsecure, LDAP port 389.

Application security

To ensure application security, you must prevent outsiders from gaining access to the passwords that you use in `cfldap` tags. The best way to do this is to use variables for your `username` and `password` attributes. You can set these variables on one encrypted application page. For more information on securing applications, see Chapter 16, “Securing Applications,” on page 373.

CHAPTER 24
Building a Search Interface

You can provide a full text search capability for documents and data sources on a ColdFusion site by enabling the Verity search engine. Verity full text search lets people visiting your site use simple one- and two-word searches to quickly find the information thFey need. You can use the more robust Verity Query Language and the Verity advanced search operators to transparently implement business-specific meaning behind searches. This allows even one word searches to return accurate results.

This chapter describes how to build a Verity search interface with which users can perform powerful searches on your site. It also describes how to index your documents and data sources so that users can search them.

Contents

About Verity 547

Creating a search tool for ColdFusion applications. 553

Creating a search page 561

Enhancing search results 564

Working with data returned from a query 572

About Verity

To efficiently search through paragraphs of text or files of varying types, you need full-text search capabilities. Macromedia ColdFusion MX includes the Verity search engine, which provides full-text indexing and searching.

The Verity engine performs searches against collections, not against the actual documents. A *collection* is a special database created by Verity that contains metadata that describes the documents that you have indexed. The *indexing* process examines documents of various types in a collection and creates a metadata description—the *index*—which is specialized for rapid search and retrieval operations.

The ColdFusion implementation of Verity supports collections of the following basic data types:

- Text files such as HTML pages and CFML pages
- Binary documents (see "Supported file types" on page 549)
- Record sets returned from a query and CF query object, including: `cfquery`, `cfldap`, and `cfpop` queries

You can build collections from individual documents or from an entire directory tree. Collections can be stored anywhere, so you have a great deal of flexibility in accessing indexed data.

In your ColdFusion application, you can search multiple collections, each of which can focus on a specific group of documents or queries, according to subject, document type, location, or any other logical grouping. Because you can perform searches against multiple collections, you have substantial flexibility in designing your search interface.

Using Verity with ColdFusion

Here are some ways to use Verity with ColdFusion:

- Index your website and provide a generalized search mechanism, such as a form interface, for executing searches.
- Index specific directories that contain documents for subject-based searching.
- Index specific categories of documents. By organizing your documents into categories, you can let users search specific types of documents. For example, if your website contains FAQs, documentation, and tutorials, you can create a search that lets users search within each of these categories.
- Index `cfquery` record sets, giving users the ability to search against the data. Because collections contain data optimized for retrieval, this method is much faster than performing multiple database queries to return the same data.
- Index `cfldap` and `cfpop` query results.
- Manage and search collections generated outside of ColdFusion using native Verity tools. Collections must be registered with the Verity K2 administration service. To do this you must either use the Verity tools, or map the collection using the `cfcollection` tag.
- Index e-mail generated by ColdFusion application pages and create a searching mechanism for the indexed messages.
- Build collections of inventory data and make those collections available for searching from your ColdFusion application pages.
- Support international users in a range of languages using the `cfindex`, `cfcollection`, and `cfsearch` tags.

Advantages of using Verity

Verity can index the output from queries so that you or a user can search against the record sets. Searching query results has a clear advantage over using SQL to search a database directly in speed of execution because metadata from the record sets are stored in a Verity index that is optimized for searching.

Performing a Verity search has the following advantages over other search methods:

- You can reduce the programming overhead of query constructs by allowing users to construct their own queries and execute them directly. You need to be concerned only with presenting the output to the client web browser.
- Verity can index database text fields, such as notes and product descriptions, that cannot be effectively indexed by native database tools.
- When indexing collections that contain documents in formats such as Adobe Acrobat (PDF) and Microsoft Word, Verity scans for the document title (if one was entered), in addition to the document text, and displays the title in the search results list.
- When Verity indexes web pages, it can return the URL for each document. This is a valuable document management feature.

For more information, see "Indexing data returned by a query" on page 573.

Supported file types

The ColdFusion Verity implementation supports a wide array of file and document types. As a result, you can index web pages, ColdFusion applications, and many binary document types and produce search results that include summaries of these documents.

Verity supports the following formats:

Document format	Format	Version(s)
Text and markup	ANSI (TXT)	All versions
	ASCII (TXT)	All versions
	HTML (HTM)	3
	IBM DCA/RFT (Revisable Form Text) (DC)	SC23-0758-1
	Rich Text Format/WordPad (RTF)	1 through 1.6
	Unicode Text (TXT)	3, 4

Document format	Format	Version(s)
Word processing	Adobe Maker Interchange Format (MIF)	5, 5.5, 6, 7
	Applix Words (AW)	3.11, 4.2, 4.3, 4.4, 4, 41, 4.2
	DisplayWrite (IP)	4
	Folio Flat File (FFF)	3.1
	Fujitsu Oasys (OA2)	7
	JustSystems Ichitaro (JTD)	8, 9, 10, 12
	Lotus AMI Pro (SAM)	2, 3
	Lotus Word Pro (LWP) (Windows only)	96, 97, Millennium Edition R9
	Microsoft Word for PC (DOC)	4, 5, 5.5, 6
	Microsoft Word for Windows (DOC)	1 through 2002
	Microsoft Word for Macintosh (DOC)	4, 5, 6, 98
	Microsoft Works (WPS)	1 through 2000
	Microsoft Windows Write (WRI)	1, 2, 3
	WordPerfect for Windows V5 (WO)	5, 5.1
	WordPerfect for Windows V6 and higher (WPD)	6, 7, 8, 10, 2000
	WordPerfect for Macintosh	1.02, 2, 2.1, 2.2, 3, 3.1
	WordPerfect for Linux	6
	XyWrite (XY4)	4.12
Spreadsheet formats	Applix Spreadsheets (AS)	4.2, 4.3, 4.4
	Comma Separated Values (CSV)	No specific version
	Corel Quattro Pro (QPW, WB3)	5, 6, 7, 8
	Lotus 1-2-3 for SmartSuite (123)	96, 97, Millennium Edition R9
	Lotus 1-2-3 (WK4)	2, 3, 4, 5
	Lotus 1-2-3 Charts (123)	2, 3, 4, 5
	Microsoft Excel for Windows (XLS)	2.2, 3, 4, 5, 96, 97, 2000, 2002
	Microsoft Excel for Macintosh (XLS)	98
	Microsoft Excel Charts (XLS)	2, 3, 4, 5, 6, 7
	Microsoft Works Spreadsheet (S30,S40)	1, 2, 3, 4

Document format	Format	Version(s)
Presentation formats	Applix Presents (AG)	4.0, 4.2, 4.3, 4.4
	Corel Presentations (SHW)	7, 9, 10, 11, 2000
	Lotus Freelance Graphics for Windows (PRE)	2, 96, 97, 98, Millennium Edition R9
	Lotus Freelance Graphics 2 (PRE)	2
	Microsoft PowerPoint for Windows (PPT)	95, 97, 2000, 2002
	Microsoft PowerPoint for PC (PPT)	4
	Microsoft PowerPoint for Macintosh (PPT)	98
	Microsoft Project (MPP)	98, 2000, 2002
Display formats	Adobe Portable Document Format (PDF)	1.1 (Acrobat 2.0) to 1.4 (Acrobat 5.0)
Graphics formats supported for indexing	AutoCAD Drawing format (DWG) (standalone) (does not extract metadata)	R13, R14, and R2000
	AutoCAD Drawing format (DXF) (standalone) (does not extract metadata)	R13, R14, and R2000
	Computer Graphics Metafile (CGM) (embedded)	No specific version
	Enhanced Metafile (EMF) (embedded and standalone)	No specific version
	Lotus Pic (PIC) (standalone)	No specific version
	Microsoft Visio (standalone)	6
	Tagged Image File (TIFF) (standalone)	5
	Windows Metafile (WMF) (embedded and standalone)	3
Multimedia formats	MPEG-1 audio layer 3 (MP3)	ID3 versions 1 and 2
Container formats	DynaZIP	No specific version
	PKZIP (zip)	PKWARE versions through 2.04g
	WinZIP	No specific version
E-mail formats	Microsoft Outlook (msg)	97, 2000, 2002
	Microsoft Outlook Express (eml)	No specific version

Specifying a language

If you install the optional Coldfusion International Search pack, you can specify a language other than English when creating a collection.

ColdFusion supports Verity Locales in European, Asian, and Middle Eastern languages. For more information about installing Verity Locales, see *Installing and Using ColdFusion MX*.

For English language support, Verity provides two options: English (Basic) and English (Advanced). The default language for Verity collections is English (Basic). Indexing a collection using English (Basic) is faster than using English (Advanced), however, English (Advanced) provides better search results.

You must specify a language when you create the collection. The language you specify should match the language the documents were authored in. By specifying the language your documents are written in, Verity is able to correctly interpret accented characters, and use variations of word stems and roots. If you have documents in several languages, create separate collections for each of them.

To specify a language when you are indexing data, select the language from the drop-down list when you create a collection with the ColdFusion MX Administrator. In CFML, the `cfcollection`, `cfindex`, and `cfsearch` tags have an optional `language` attribute that you use to specify the language of the collection.

Use the following table to find the correct value for the `language` attribute for your collection; for example, the following code creates a collection for simplified Chinese:

```
<cfcollection action = "create" collection = "lei_01"
  path = "c:\CFusionMX7\verity\collections"
  language = "simplified_chinese">
```

The following table lists the languages names and attributes that ColdFusion supports:

Language	Language attribute
Arabic	arabic
Chinese (simplified)	simplified_chinese
Chinese (traditional)	traditional_chinese
Czech	czech
Danish	danish
Dutch	dutch
English (Basic)	english
English (Advanced)	englishx
Finnish	finnish
French	french
German	german
Greek	greek
Hebrew	hebrew
Hungarian	hungarian
Italian	italian
Japanese	japanese
Korean	korean

Language	Language attribute
Norwegian	norwegian
Norwegian (Bokmal)	bokmal
Norwegian (Nynorsk)	nynorsk
Polish	polish
Portuguese	portuguese
Russian	russian
Spanish	spanish
Swedish	swedish
Turkish	turkish

You can register collections in the ColdFusion MX Administrator or by creating a collection with the `cfcollection` tag. If you register a given collection with ColdFusion and you specify a `language` attribute, you do not have to specify the `language` attribute when using `cfindex` and `cfsearch` tags for that collection. If you do not register a given collection with ColdFusion, ColdFusion uses English (Basic), the default language, unless you specify the language in the `language` attribute for the `cfindex` and `cfsearch` tags for that collection.

Note: When you search a collection in a language other than English, you must translate operators such as AND and OR into the language of the collection.

Creating a search tool for ColdFusion applications

There are three main tasks in creating a search tool for your ColdFusion application:

1. Create a collection.
2. Index the collection.
3. Design a search interface.

You can perform each task programmatically—that is, by writing CFML code. Alternatively, you can use the ColdFusion MX Administrator to create and index a collection.

This chapter presents the noncoding methods for developing a search tool, followed by code examples that perform the same task.

Creating a collection with the ColdFusion MX Administrator

Use the following procedure to quickly create a collection with the ColdFusion MX Administrator:

To create a collection with the ColdFusion MX Administrator:

1. In the ColdFusion MX Administrator, select Data & Services > Verity Collections.

   The Verity Collections page appears.
2. Enter a name for the collection; for example, DemoDocs.

3. Enter a path for the directory location of the new collection; for example, C:\CFusionMX7\verity\collections\.

   By default in the server configuration, ColdFusion stores collections in *cf_root*\verity\collections\ in Windows and in *cf_root*/verity/collections on UNIX. In the multiserver configuration, the default location for collections is *cf_webapp_root*/verity/collections. In the J2EE configuration, the default location for collections is *verity_root*/verity/collections, where *verity_root* is the directory in which you installed Verity.

   Note: This is the location for the collection, not for the files that you will search.

4. (Optional) Select a language other than English for the collection from the Language drop-down list.

   For more information on selecting a language, see "Specifying a language" on page 551.

5. (Optional) Select Enable Category Support to create a Verity Parametric collection.

   For more information on using categories, see "Narrowing searches using categories" on page 567.

6. Click Create Collection.

   The name and full path of the new collection appears in the list of Verity Collections.

You have successfully created an empty collection. A collection becomes populated with data when you index it. For more information, see the next section, "About indexing a collection" on page 554.

About indexing a collection

In order for information to be searched, it must be indexed. Indexing extracts both meaning and structure from unstructured information by indexing each document that you specify into a separate Verity collection that contains a complete list of all the words used in a given document along with metadata about that document. Indexed collections include information such as word proximity, metadata about physical file system addresses, and URLs of documents.

When you index databases and other record sets that you generated using a query, Verity creates a collection that normalizes both the structured and unstructured data. Search requests then check these collections rather than scanning the actual documents and database fields. This provides a faster search of information, regardless of the file type and whether the source is structured or unstructured.

Just as with creating a collection, you can index a collection either programmatically or using the ColdFusion MX Administrator. Use the following guidelines to determine which method to use:

Use the Administrator	Use the cfindex tag
To index document files	To index ColdFusion query results
When the collection does not require frequent updates	When the collection requires frequent updates

Use the Administrator	Use the cfindex tag
To create the collection without writing any CFML code	To dynamically update a collection from a ColdFusion application page
To create a collection once	When the collection requires updating by others

You can use `cfcollection action="optimize"` if you notice that searches on a collection take longer than they did previously.

Updating an index

Documents are modified frequently in many user environments. After you index your documents, any changes that you make are not reflected in subsequent Verity searches until you re-index the collection. Depending on your environment, you can create a scheduled task to automatically keep your indexes current. For more information on scheduled tasks, see *Configuring and Administering ColdFusion MX*.

Creating a ColdFusion search tool programmatically

You can create a Verity search tool for your ColdFusion application in CFML. Although writing CFML code can take more development time than using these tools, there are situations in which writing code is the preferred development method.

Creating a collection with the cfcollection tag

The following are cases in which you might prefer using the `cfcollection` tag rather than the ColdFusion MX Administrator to create a collection:

- You want your ColdFusion application to be able to create, delete, and maintain a collection.
- You do not want to expose the ColdFusion MX Administrator to users.
- You want to create indexes on servers that you cannot access directly; for example, if you use a hosting company.

When using the `cfcollection` tag, you can specify the same attributes as in the ColdFusion MX Administrator:

Attribute	Description
action	(Optional) The action to perform on the collection (create, delete, repair, or optimize). The default value for the `action` attribute is `list`. For more information, see `cfcollection` in *CFML Reference*.
collection	The name of the new collection, or the name of a collection upon which you will perform an action.
path	The location for the Verity collection.
language	language
categories	(Optional) Specifies that `cfcollection` create a Verity Parametric Index (PI) for this collection. By default, the categories attribute is set to `False`. To create a collection using categories specify `Yes`.

You can create a collection by directly assigning a value to the `collection` attribute of the `cfcollection` tag, as shown in the following code:

```
<cfcollection action = "create"
  collection = "a_new_collection"
  path = "c:\CFusionMX7\verity\collections\">
```

If you want your users to be able to dynamically supply the name and location for a new collection, use the following procedures to create form and action pages.

To create a simple collection form page:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Collection Creation Input Form</title>
</head>

<body>
<h2>Specify a collection</h2>
<form action="collection_create_action.cfm" method="POST">

  <p>Collection name:
  <input type="text" name="CollectionName" size="25"></p>

  <p>What do you want to do with the collection?</p>
  <input type="radio"
    name="CollectionAction"
    value="Create" checked>Create<br>
  <input type="radio"
    name="CollectionAction"
    value="Optimize">Optimize<br>
  <input type="submit"
    name="submit"
    value="Submit">
</form>

</body>
</html>
```

2. Save the file as collection_create_form.cfm in the myapps directory under the web root directory.

Note: The form will not work until you write an action page for it, which is the next procedure.

To create a collection action page:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>cfcollection</title>
</head>

<body>
<h2>Collection creation</h2>

<cfoutput>

  <cfswitch expression=#Form.collectionaction#>
    <cfcase value="Create">
      <cfcollection action="Create"
      collection="#Form.CollectionName#"
      path="c:\CFusionMX7\verity\collections\">
      <p>The collection #Form.CollectionName# is created.</p>
    </cfcase>

    <cfcase value="Repair">
      <cfcollection action="Repair"
      collection="#Form.CollectionName#">
      <p>The collection #Form.CollectionName# is repaired.</p>
    </cfcase>

    <cfcase value="Optimize">
      <cfcollection action="Optimize"
      collection="#Form.CollectionName#">
      <p>The collection #Form.CollectionName# is optimized.</p>
    </cfcase>

    <cfcase value="Delete">
      <cfcollection action="Delete"
      collection="#Form.CollectionName#">
      <p>The collection is deleted.</p>
    </cfcase>
  </cfswitch>
</cfoutput>
</body>
</html>
```

2. Save the file as collection_create_action.cfm in the myapps directory under the web root directory.
3. In the web browser, enter the following URL to display the form page:

   http://hostname:portnumber/myapps/collection_create_form.cfm
4. Enter a collection name; for example, CodeColl.
5. Verify that Create is selected and submit the form.
6. (Optional) In the ColdFusion MX Administrator, reload the Verity Collections page.

The name and full path of the new collection appear in the list of Verity Collections.

You successfully created a collection, named CodeColl, that currently has no data. For information on indexing your collection using CFML, see "Indexing a collection using the cfindex tag" on page 558.

Indexing a collection using the cfindex tag

You can index a collection in CFML using the `cfindex` tag, which eliminates the need to use the ColdFusion MX Administrator. The `cfindex` tag populates the collection with metadata that is then used to retrieve search results. You can use the `cfindex` tag to index either physical files (documents stored within your website's root folder), or the results of a database query.

Note: Prior to indexing a collection, you must create a Verity collection using either the ColdFusion MX Administrator, or the `cfcollection` tag. For more information, see "Creating a collection with the ColdFusion MX Administrator" on page 553, or "Creating a collection with the cfcollection tag" on page 555.

When using the `cfindex` tag, the following attributes correspond to the values that you would enter using the ColdFusion MX Administrator to index a collection:

Attribute	Description
`collection`	The name of the collection.
`action`	Specifies what the `cfindex` tag should do to the collection. The default `action` is to update the collection, which generates a new index. Other `actions` are to delete, purge, or refresh the collection.
`type`	Specifies the type of files or other data to which the `cfindex` tag applies the specified action. The value you assign to the `type` attribute determines the value to use with the `key` attribute (see the following list). When you enter a value for the `type` attribute, `cfindex` expects a corresponding value in the `key` attribute. For example, if you specify `type=file`, `cfindex` expects a directory path and filename for the `key` attribute. The `type` attribute has the following possible values: • `file` Specifies a directory path and filename for the file that you are indexing. • `path` Specifies a directory path that contains the files that you are indexing. • `custom` Specifies custom data, such as a record set returned from a query.
`extensions`	(Optional) The delimited list of file extensions that ColdFusion uses to index files if `type="path"`.
`key`	The value that you specify for the `key` attribute depends on the value set for the `type` attribute: • If `type="file"`, the `key` is the directory path and filename for the file you are indexing. • If `type="path"`, the `key` is the directory path that contains the files you are indexing. • If `type="custom"`, the `key` is a unique identifier specifying the location of the documents you are indexing; for example, the URL of a specific web page or website whose contents you want to index. If you are indexing data returned by a query (from a database for example), the `key` is the name of the record set column that contains the primary key.

Attribute	Description
URLpath	(Optional) The URL path for files if type="file" and type="path". When the collection is searched with the cfsearch tag, ColdFusion MX works as follows: • type="file" The URLpath attribute contains the URL to the file. • type="path" The path name is automatically prefixed to filenames and returned as the URLpath attribute.
recurse	(Optional) Yes or No. If type = "path", Yes specifies that directories below the path specified in the key attribute are included in the indexing operation.
language	(Optional) The language of the collection. The default is English Basic. To learn more about support for languages, see "Specifying a language" on page 551.

You can use form and action pages similar to the following examples to select and index a collection.

To select which collection to index:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Select the Collection to Index</title>
</head>
<body>

<h2>Specify the index you want to build</h2>

<form method="Post" action="collection_index_action.cfm">
  <p>Enter the collection you want to index:
  <input type="text" name="IndexColl" size="25" maxLength="35"></p>
  <p>Enter the location of the files in the collection:
  <input type="text" name="IndexDir" size="50" maxLength="100"></p>
  <p>Enter a Return URL to prepend to all indexed files:
  <input type="text" name="urlPrefix" size="80" maxLength="100"></p>

  <input type="submit" name="submit" value="Index">

</form>

</body>
</html>
```

2. Save the file as collection_index_form.cfm in the myapps directory under the *web_root*.

Note: The form does not work until you write an action page for it, which is the next procedure.

To use cfindex to index a collection:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>Creating Index</title>
</head>
<body>
<h2>Indexing Complete</h2>

<cfindex collection="#Form.IndexColl#"
  action="refresh"
  extensions=".htm, .html, .xls, .txt, .mif, .doc"
  key="#Form.IndexDir#"
  type="path"
  urlpath="#Form.urlPrefix#"
  recurse="Yes"
  language="English">

<cfoutput>
  The collection #Form.IndexColl# has been indexed.
</cfoutput>
</body>
</html>
```

2. Save the file as collection_index_action.cfm.
3. In the web browser, enter the following URL to display the form page:

   http://*hostname:portnumber*/myapps/collection_index_form.cfm
4. Enter a collection name; for example, CodeColl.
5. Enter a file location; for example, C:\CFusionMX7\wwwroot\vw_files.
6. Enter a URL prefix; for example, http://localhost:8500/vw_files (assuming that you are using the built-in web server).
7. Click Index.

   A confirmation message appears on successful completion.

Note: For information about using the `cfindex` tag with a database to index a collection, see "Working with data returned from a query" on page 572.

Indexing a collection with the ColdFusion MX Administrator

As an alternative to programmatically indexing a collection, use the following procedure to index a collection with the ColdFusion MX Administrator.

To use ColdFusion MX Administrator to index a collection:

1. In the list of Verity Collections, select a collection name; for example, CodeColl.
2. Click Index to open the index page.
3. For File Extensions, enter the types of files to index. Use a comma to separate multiple file types; for example, .htm, .html, .xls, .txt, .mif, .doc.

4. Enter (or Browse to) the directory path that contains the files to be indexed; for example, C:\Inetpub\wwwroot\vw_files.
5. (Optional) To extend the indexing operation to all directories below the selected path, select the Recursively index subdirectories check box.
6. (Optional) Enter a Return URL to prepend to all indexed files.

   This step lets you create a link to any of the files in the index; for example, http://127.0.0.1/vw_files/.
7. (Optional) Select a language other than English.

   For more information, see "Specifying a language" on page 551.
8. Click Submit Changes.

   On completion, the Verity Collections page appears.

   Note: The time required to generate the index depends on the number and size of the selected files in the path.

This interface lets you easily build a very specific index based on the file extension and path information you enter. In most cases, you do not need to change your server file structures to accommodate the generation of indices.

Creating a search page

You use the `cfsearch` tag to search an indexed collection. Searching a Verity collection is similar to a standard ColdFusion query: both use a dedicated ColdFusion tag that requires a `name` attribute for their searches and both return a query object that contains rows matching the search criteria. The following table compares the two tags:

cfquery	cfsearch
Searches a data source	Searches a collection
Requires a `name` attribute	Requires a `name` attribute
Uses SQL statements to specify search criteria	Uses a criteria attribute to specify search criteria
Returns variables keyed to database table field names	Returns a unique set of variables
Uses `cfoutput` to display query results	Uses `cfoutput` to display search results

Note: You receive an error if you attempt to search a collection that has not been indexed.

The following are important attributes for the `cfsearch` tag:

Attribute	Description
`name`	The name of the search query.
`collection`	The name of the collection(s) being searched. Separate multiple collections with a comma; for example, `collection = "sprocket_docs,CodeColl"`.

Attribute	Description
criteria	The search target (can be dynamic).
maxrows	The maximum number of records returned by the search. Always specify this attribute to ensure optimal performance (start with 300 or less, if possible).

Each cfsearch returns variables that provide the following information about the search:

Attribute	Description
RecordCount	The total number of records returned by the search.
CurrentRow	The current row of the record set.
RecordsSearched	The total number of records in the index that were searched. If no records were returned in the search, this property returns a null value.
Summary	Automatic summary saved by the cfindex tag.
Context	A context summary that contains the search terms, highlighted in bold (by default). This is enabled if you set the contextpassages attribute to a number greater than zero.

Additionally, if you specify the status attribute, the cfsearch tag returns the status structure, which contains the information in the following table:

Variable	Description
found	The number of documents that contain the search criteria.
searched	The number of documents searched. Corresponds to the recordsSearched column in the search results.
time	The number of milliseconds the search took, as reported by the Verity K2 search service.
suggestedQuery	An alternative query, as suggested by Verity, that may produce better results. This often contains corrected spellings of search terms. Present only when the suggestions tag attribute criteria is met.
Keywords	A structure that contains each search term as a key to an array of up to five possible alternative terms in order of preference. Present only when the suggestions tag attribute criteria is met.

You can use search form and results pages similar to the following examples to search a collection.

To create a search form:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Searching a collection</title>
</head>
<body>
<h2>Searching a collection</h2>

<form method="post" action="collection_search_action.cfm">
  <p>Enter search term(s) in the box below. You can use AND, OR, NOT, and
  parentheses. Surround an exact phrase with quotation marks.</p>
  <p><input type="text" name="criteria" size="50" maxLength="50">
  </p>
  <input type="submit" value="Search">
</form>
</body>
</html>
```

2. Save the file as collection_search_form.cfm.

Enter search target words in this form, which ColdFusion passes as the variable `criteria` to the action page, which displays the search results.

To create the results page:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Search Results</title>
</head>
<body>
<cfsearch
  name = "codecoll_results"
  collection = "CodeColl"
  criteria = "#Form.criteria#"
  contextPassages = "1"
  contextBytes = "300"
  maxrows = "100">

<h2>Search Results</h2>
<cfoutput>
Your search returned #codecoll_results.RecordCount# file(s).
</cfoutput>

<cfoutput query="codecoll_results">
  <p>
  File: <a href="#URL#">#Key#</a><br>
  Document Title (if any): #Title#<br>
  Score: #Score#<br>
  Summary: #Summary#<br>
  Highlighted Summary: #context#</p>
</cfoutput>
</body>
</html>
```

2. Save the file as collection_search_action.cfm.
3. View collection_search_form.cfm in the web browser.
4. Enter a target word(s) and click Search.

Note: As part of the indexing process, Verity automatically produces a summary of every document file or every query record set that gets indexed. The default summary result set column selects the best sentences, based on internal rules, up to a maximum of 500 characters. Every `cfsearch` operation returns summary information by default. For more information on this topic, see "Using Verity Search Expressions" on page 581. Alternatively, you can use the context result set column, which provides a context summary with highlighted search terms.

Enhancing search results

ColdFusion lets you enhance the results of searches by letting you incorporate search features that let users more easily find the information they need. The following search enhancements are available through Verity:

- "Highlighting search terms" on page 564
- "Providing alternative spelling suggestions" on page 565
- "Narrowing searches using categories" on page 567

Highlighting search terms

Term highlighting lets users quickly scan retrieved documents to determine whether they contain the desired information. This can be especially useful when searching lengthy documents, letting users quickly locate relevant information returned by the search.

To implement term highlighting, use the following `cfsearch` attributes in the search results page:

Attributes	Description
ContextHighlightBegin	Specifies the HTML tag to prepend to the search term within the returned documents. This attribute must be used in conjunction with ContextHighlightEnd to highlight the resulting search terms. The default HTML tag is ‹b›, which highlights search terms using bold type.
ContextHighlightEnd	Specifies the HTML tag to append to the search term within the returned documents.
ContextPassages	The number of passages/sentences Verity returns in the context summary (the context column of the results). The default value is 0; this disables the context summary.
ContextBytes	The total number of bytes that Verity returns in the context summary. The default is 300 bytes.

The following example adds to the previous search results example, highlighting the returned search terms with bold type.

To create a search results page using term highlighting:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Search Results</title>
</head>
<body>
<cfsearch
  name = "codecoll_results"
  collection = "CodeColl"
  criteria = "#Form.Criteria#">
  ContextHighlightBegin="<b>"
  ContextHighlightEnd="</b>"
  ContextPassages="1"
  ContextBytes="500"
  maxrows = "100">
<h2>Search Results</h2>
<cfoutput>
Your search returned #codecoll_results.RecordCount# file(s).
</cfoutput>

<cfoutput query="codecoll_results">
  <p>
  File: <a href="#URL#">#Key#</a><br>
  Document Title (if any): #Title#<br>
  Score: #Score#<br>
  Summary: #Summary#<br>
  Highlighted Summary: #context#</p>
</cfoutput>
</body>
</html>
```

2. Save the file as collection_search_action.cfm.

   Note: This overwrites the previous ColdFusion example page.

3. View collection_search_form.cfm in the web browser:
4. Enter a target word(s) and click Search.

Providing alternative spelling suggestions

Many unsuccessful searches are the result of incorrectly spelled query terms. Verity can automatically suggest alternative spellings for misspelled queries using a dictionary that is dynamically built from the search index.

To implement alternative spelling suggestions, you use the `cfsearch` tag's `suggestions` attribute with an integer value. If the number of documents returned by the search is less than or equal to the value you specify, Verity provides alternative search term suggestions. In addition to using the `suggestions` attribute, you may also use the `cfif` tag to output the spelling suggestions, and a link through which to search on the suggested terms.

Note: Using alternative spelling suggestions incurs a small performance penalty. This occurs because the `cfsearch` tag must also look up alternative spellings in addition to the specified search terms.

The following example specifies that if the number of search results returned is less than 5, an alternative search term—which is displayed using the `cfif` tag—is displayed with a link that the user can click to activate the alternate search.

To create a search results page providing alternative spelling suggestions:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Search Results</title>
</head>
<body>
<cfsearch
  name = "codecoll_results"
  collection = "CodeColl"
  criteria = "#Form.Criteria#">
  status = "info"
  suggestions="5"
  ContextPassages = "1"
  ContextBytes = "300"
  maxrows = "100">
<h2>Search Results</h2>
<cfoutput>
Your search returned #codecoll_results.RecordCount# file(s).
</cfoutput>
<cfif info.FOUND LTE 5 AND isDefined("info.SuggestedQuery)>
  Did you mean:
  <a href="search,cfm?query=#info.SuggestedQuery#>#info.SuggestedQuery#</a>
</cfif>
<cfoutput query="codecoll_results">
  <p>
  File: <a href="#URL#">#Key#</a><br>
  Document Title (if any): #Title#<br>
  Score: #Score#<br>
  Summary: #Summary#<br>
  Highlighted Summary: #context#</p>
</cfoutput>
</body>
</html>
```

2. Save the file as collection_search_action.cfm.

   Note: This overwrites the previous ColdFusion example page.

3. View collection_search_form.cfm in the web browser:
4. Enter any misspelled target words and click Search.

Narrowing searches using categories

Verity lets you organize your searchable documents into *categories*. Categories are groups of documents (or database tables) that you define, and then let users search within them. For example, if you wanted to create a search tool for a software company, you might create categories such as whitepapers, documentation, release notes, and marketing collateral. Users can then specify one or more categories in which to search for information. Thus, if users visiting the website wanted to learn about a conceptual aspect of your company's technology, they might restrict their search to the whitepaper and marketing categories.

Typically, you will want to provide users with pop-up menus or check boxes from which they can select categories to narrow their searches. Alternately, you might create a form that lets users enter both a category name in which to search, and search keywords.

To create a search application that uses categories:

1. Create a collection with support for categories enabled.

   See "Creating collections with support for categories" on page 567.

2. Index the collection, specifying the `category` and `categoryTree` attributes appropriate to the collection.

   For more information on indexing Verity collections with support for categories, see "Indexing collections that contain categories" on page 568.

3. Create a search page that lets users search within the categories that you've created.

   Create a search page using the `cfsearch` tag that lets users more easily search for information by restricting searches to the specified category and, if specified, its hierarchical tree.

   For more information on searching Verity collections with support for categories, see "Searching collections that contain categories" on page 569.

Creating collections with support for categories

You can either select Enable Category Support from the ColdFusion MX Administrator, or write a `cfcollection` tag that uses the `category` attribute. By enabling category support, you create a collection that contains a Verity Parametric Index (PI).

```
<cfcollection
  action = "action"
  collection = "collectionName"
  path = "path_to_verity_collection"
  language = "English"
  categories = "yes">
```

For more information on using the `cfcollection` tag to create Verity collections with support for categories, see `cfcollection` in Chapter 2, "ColdFusion Tags," in *CFML Reference*.

Indexing collections that contain categories

When you index a collection with support for categories enabled, you must do the following:

- Specify a category name using the `category` attribute. The name (or names) that you provide identifies the category so that users can specify searches on the documents that the collection contains. For example, you might create five categories named *taste*, *touch*, *sight*, *sound*, and *smell*. When performing a search, users could select from either a pop-up menu or check box to search within one or more of the categories, thereby limiting their search within a given range of topics.

```
<cfindex collection="#Form.IndexColl#"
  action="update"
  extensions=".htm, .html, .xls, .txt, .mif, .doc, .pdf"
  key="#Form.IndexDir#"
  type="path"
  urlpath="#Form.urlPrefix#"
  recurse="Yes"
  language="English"
  category="taste, touch, sight, sound, smell">
```

- Specify a hierarchical document tree (similar to a file system tree) within which you can limit searches, when you use the `categoryTree` attribute. With the `categoryTree` attribute enabled, ColdFusion limits searches to documents contained within the specified path.

  To use the `categoryTree` attribute, you specify a hierarchical document tree by listing each category as a string, and separating them using forward slashes (/). The tree structure that you specify in a search is the root of the document tree from which you want the search to begin. The `type=path` attribute appends directory names to the end of the returned value (as it does when specifying the `urlpath` attribute).

  Note: You can specify only a single category tree.

```
<cfindex collection="#Form.IndexColl#"
  action="update"
  extensions=".htm, .html, .xls, .txt, .mif, .doc, .pdf"
  key="#Form.IndexDir#"
  type="path"
  urlpath="#Form.urlPrefix#"
  recurse="Yes"
  language="English"
  category="taste, touch, sight, sound, smell"
  categoryTree="human/senses/taste">
```

  For more information on using the `cfindex` tag to create Verity collections with support for categories, see `cfindex` in Chapter 2, "ColdFusion Tags," in *CFML Reference*.

Searching collections that contain categories

When searching data in a collection created with categories, you specify `category` and `categoryTree`. The values supplied to these attributes specify what category should be searched for the specified search string (the `criteria` attribute). The `category` attribute can contain a comma separated list of categories to search. Both attributes can be specified at the same time.

```
<cfsearch collection="collectionName"
  name="results"
  maxrows = "100"
  criteria="search keywords"
  category="FAQ,Technote"
  categoryTree="Docs/Tags">
```

Note: If `cfsearch` is executed on a collection that was created without category information, an exception is thrown.

To search collections that contain categories, you use the `cfsearch` tag, and create an application page that searches within specified categories. The following example lets the user enter and submit the name of the collection, the category in which to search, and the document tree associated with the category through a form. By restricting the search in this way, the users are better able to retrieve the documents that contain the information they are looking for. In addition to searching with a specified category, this example also makes use of the `contextHighlight` attribute, which highlights the returned search results.

```
<cfparam name="collection" default="test-pi">

<cfoutput>
<form action="#CGI.SCRIPT_NAME#" method="POST">
 Collection Name: <input Type="text" Name="collection" value="#collection#">
 <P>
 Category: <input Type="text" Name="category" value=""><br>
 CategoryTree: <input Type="text" Name="categoryTree" value=""><br>
 <P>
Search: <input Type="text" Name="criteria">
 <input Type="submit" Value="Search">
</form>
</cfoutput>

<cfif isdefined("Form.criteria")>
  <cfoutput>Search results for: <b>#criteria#</b></cfoutput>
  <br>
  <cfsearch collection="#form.collection#"
    category="#form.category#"
    categoryTree="#form.categoryTree#"
    name="sr"
    status="s"
    criteria="#form.criteria#"
    contextPassages="3"
    contextBytes="300"
    contextHighlightBegin="<i><b>"
    contextHighlightEnd="</b></i>"
    maxrows="100">
  <cfdump var="#s#">
```

```
  <cfoutput>
  <p>Number of records in query: #sr.recordcount#</P>
  </cfoutput>

  <cfdump var="#sr#">

  <cfoutput Query="sr">
  Title: <i>#title#</i><br>
  URL: #url#<br>
  Score: #score#<br>
  <hr>
  #context#<br>
  <br>
  #summary#<br>
  <hr>
  </cfoutput>
</cfif>
```

For more information on using the `cfindex` tag to create Verity collections with support for categories, see `cfsearch` in Chapter 2, "ColdFusion Tags," in *CFML Reference*.

Retrieving information about the categories contained in a collection

You can retrieve the category information for a collection by using the `cfcollection` tag's `categoryList` action. The `categoryList` action returns a structure that contains two keys:

Variable	Description
categories	The name of the category and its hit count, where hit count is the number of documents in the specified category.
categorytrees	The document tree (a/b/c) and hit count, where hit count is the number of documents at or below the branch of the document tree.

You can use the information returned by `categoryList` to display to users the number of documents available for searching, as well the document tree available for searching. You can also create a search interface that lets the user select what category to search within based on the results returned by `categoryList`.

```
<cfcollection
  action="categoryList"
  collection="collectionName"
  "name="info">

<cfoutput>
  <cfset catStruct=info.categories>
  <cfset catList=StructKeyList(catStruct)>
  <cfloop list="catList" index="cat"> Category: #cat# <br>
    Documents: #catStruct[cat]#<br>
  </cfloop>
</cfoutput>
```

To retrieve information about the categories contained in a collection, you use the `cfcollection` tag, and create an application page that retrieves category information from the collection and displays the number of documents contained by each category. This example lets the user enter and submit the name of the collection via a form, and then uses the `categoryList` action to retrieve information about the number of documents contained by the collection, and the hierarchical tree structure into which the category is organized.

```
<html>
<head>
  <title>Category information</title>
</head>
<body>
<cfoutput>
<form action="#CGI.SCRIPT_NAME#" method="POST">
  Enter Collection Name: <input Type="text" Name="collection"
    value="#collection#"><br>
 <input Type="submit" Value="GetInfo">
</form>
</cfoutput>
<cfif isdefined("Form.collection")>
  <cfoutput>
  Getting collection info...
  <br>
  <cfflush>
  <cfcollection
    action="categorylist"
    collection="#collection#"
    name="out">
  <br>
  <cfset categories=out.categories>
  <cfset tree=out.categorytrees>
  <cfset klist=StructKeyList(categories)>
  <table border=1>
  <th>Category</th> <th>Documents</th>
  <cfloop index="x" list="#klist#">
  <tr>
  <td>#x#</td> <td align="center">#categories[x]#</td>
  </tr>
  </cfloop>
  </table>
  <cfset klist=StructKeyList(tree)>
  <table border=1>
  <th>Category</th> <th>Documents</th>
  <cfloop index="x" list="#klist#">
  <tr>
   <td>#x#</td> <td align="center">#tree[x]#</td>
  </tr>
  </cfloop>
  </table>
  </cfoutput>
</cfif>
</body>
```

For more information on using the cfcollection tag to create Verity collections with support for categories, see cfcollection in Chapter 2, "ColdFusion Tags," in *CFML Reference*.

Working with data returned from a query

Using Verity, you can search data returned by a query—such as a database record set—as if it were a collection of documents stored on your web server. Using Verity to search makes implementing a search interface much easier, as well as letting users more easily find information contained in database files. This section describes the reasons and procedures for indexing the results of database and other queries. It also describes how a database can direct the indexing process, using different values for the type attribute of the cfindex tag.

This section describes the following topics:

- "Recordsets and types of queries" on page 572
- "Indexing data returned by a query" on page 573
- "Indexing a file returned using a query" on page 576
- "Indexing a path returned using a query" on page 577
- "Indexing query results obtained from an LDAP directory" on page 578
- "Indexing cfpop query results" on page 579

Recordsets and types of queries

When indexing record sets generated from a query (using the cfquery, cfldap, or cfpop tag), cfindex creates indexes based on the type attribute and its set value:

Type	Attribute values
File	The key attribute is the name of a column in the query that contains a full filename (including path).
Path	The key attribute is the name of a column in the query that contains a directory pathname.
Custom	The key attribute specifies a column name that can contain anything you want. In this case, the body attribute is required, and is a comma-delimited list of the names of the columns that contain the text data that is to be indexed.

The cfindex tag treats all collections the same, whether they originate from a database recordset, or if they are a collection of documents stored within your website's root folder.

Indexing data returned by a query

Indexing the results of a query is similar to indexing physical files located on your website, with the added step that you must write a query that retrieves the data you want to search. The following are the steps to perform a Verity search on record sets returned from a query:

1. Create a collection.
2. Write a query that retrieves the data you want to search, and generate a record set.
3. Index the record set using the `cfindex` tag.

   The `cfindex` tag indexes the record set as if it were a collection of documents in a folder within your website.
4. Search the collection.

   The information returned from the collection includes the database key and other selected columns. You can then use the information as-is, or use the key value to retrieve the entire row from the database table.

You should use Verity to search databases in the following cases:

- You want to perform full-text search on database data. You can search Verity collections that contain textual data much more efficiently with a Verity search than using SQL to search database tables.
- You want to give your users access to data without interacting directly with the data source itself.
- You want to improve the speed of queries.
- You want users to be able to execute queries, but not update database tables.

Unlike indexing documents stored on your web server, indexing information contained in a database requires an additional step—you must first write a query (using the `cfquery`, `cfldap`, or `cfpop` tag) that retrieves the data you want to let your users search. You then pass the information retrieved by the query to a `cfindex` tag, which indexes the data.

When indexing data with the `cfindex` tag, you must specify which column of the query represents the filename, which column represents the document title, and which column (or columns) represents the document's body (the information that you want to make searchable).

When indexing a recordset retrieved from a database, the `cfindex` tag uses the following attributes that correspond to the data source:

Attribute	Description
key	Primary key column of the data source table.
title	Specifies a query column name.
body	Columns that you want to search for the index.
type	If set to `custom`, this attribute specifies the columns that you want to index. If set to `file` or `path`, this is a column that contains either a directory path and filename, or a directory path that contains the documents to be indexed.

Using the cfindex tag to index tabular data is similar to indexing documents, with the exception that you refer to column names from the generated record set in the body attribute. In the following example, the type attribute is set to custom, specifying that the cfindex tag index the contents of the record set columns Emp_ID, FirstName, LastName, and Salary, which are identified using the body attribute. The Emp_ID column is listed as the key attribute, making it the primary key for the record set.

To index a ColdFusion query:

1. Create a Verity collection for the data that you want to index.

   The following example assumes that you have a Verity collection named CodeColl. You can use the ColdFusion MX Administrator to create the collection, or you can create the collection programmatically by using the cfcollection tag. For more information, see "Creating a collection with the ColdFusion MX Administrator" on page 553 or "Creating a collection with the cfcollection tag" on page 555.

2. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Adding Query Data to an Index</title>
</head>
<body>

<!--- Retrieve data from the table. --->
<cfquery name="getEmps" datasource="cfdocexamples">
  SELECT * FROM EMPLOYEE
</cfquery>

<!--- Update the collection with the above query results. --->
<cfindex
  query="getEmps"
  collection="CodeColl"
  action="Update"
  type="Custom"
  key="Emp_ID"
  title="Emp_ID"
  body="Emp_ID,FirstName,LastName,Salary">

<h2>Indexing Complete</h2>

<!--- Output the record set. --->
<p>Your collection now includes the following items:</p>
<cfoutput query="getEmps">
  <p>#Emp_ID# #FirstName# #LastName# #Salary#</p>
</cfoutput>
</body>
</html>
```

3. Save the file as collection_db_index.cfm in the myapps directory under the web root directory.
4. Open the file in the web browser to index the collection.

   The resulting record set appears.

To search and display the query results:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Searching a collection</title>
</head>
<body>

<h2>Searching a collection</h2>

<form method="post" action="collection_db_results.cfm">
  <p>Collection name: <input type="text" name="collname" size="30"
  maxLength="30"></p>

  <p>Enter search term(s) in the box below. You can use AND, OR, NOT,
  and parentheses. Surround an exact phrase with quotation marks.</p>
  <p><input type="text" name="criteria" size="50" maxLength="50">
  </p>
  <p><input type="submit" value="Search"></p>
</form>

</body>
</html>
```

2. Save the file as collection_db_search_form.cfm in the myapps directory under the *web_root*.

   This file is similar to collection_search_form.cfm, except the form uses collection_db_results.cfm, which you create in the next step, as its action page.

3. Create another ColdFusion page with the following content:

```
<html>
<head>
<title>Search Results</title>
</head>

<body>

<cfsearch
   collection="#Form.collname#"
   name="getEmps"
   criteria="#Form.Criteria#"
  maxrows = "100">

<!--- Output the record set. --->
<cfoutput>
Your search returned #getEmps.RecordCount# file(s).
</cfoutput>

<cfoutput query="getEmps">
  <p><table>
  <tr><td>Title: </td><td>#Title#</td></tr>
  <tr><td>Score: </td><td>#Score#</td></tr>
  <tr><td>Key: </td><td>#Key#</td></tr>
  <tr><td>Summary: </td><td>#Summary#</td></tr>
```

```
    <tr><td>Custom 1:</td><td>#Custom1#</td></tr>
    <tr><td>Column list: </td><td>#ColumnList#</td></tr>
  </table></p>

  </cfoutput>
  </body>
  </html>
```

4. Save the file as collection_db_results.cfm in the myapps directory under the *web_root*.
5. View collection_db_search_form.cfm in the web browser and enter the name of the collection and search terms.

Indexing a file returned using a query

You can index an individual file using a query by retrieving a table row whose contents are a filename. In this case, the key specifies the column that contains the complete filename. The file is indexed using the `cfindex` tag as if it were a document under the web server root folder.

In the following example, the `cfindex` tag's `type` attribute has been set to `file`, and the specified key is the name of the column that contains the full path to the file and the filename.

To index a file specified by a query:

```
<cfquery name="getEmps" datasource="cfdocexamples">
SELECT * FROM EMPLOYEE WHERE EMP_ID = 1
</cfquery>
<cfindex
  query="getEmps"
  collection="CodeColl"
  action="Update"
  type="file"
  key="Contract_File"
  title="Contract_File"
  body="Emp_ID,FirstName,LastName,Contract_File">
```

To search and display the file:

1. Create a ColdFusion page that contains the following content:

```
<!--- Output the record set.--->
<p>Your collection now includes the following items:</p>
<cfoutput query="getEmps">
<p>#Emp_ID# #FirstName# #LastName# #Contract_File#</p>
</cfoutput>
<cfsearch
  collection="#Form.collname#"
  name="getEmps"
  criteria="#Form.Criteria#"
  maxrows = "100">

<!--- Output the file name contained in the record set. --->
<cfoutput>
Your search returned #getEmps.RecordCount# file(s).
</cfoutput>
<cfoutput query="getEmps">
<p><table>
```

```
  <tr><td>Title: </td><td>#Title#</td></tr>
  <tr><td>Score: </td><td>#Score#</td></tr>
  <tr><td>Key: </td><td>#Key#</td></tr>
  <tr><td>Summary: </td><td>#Summary#</td></tr>
  <tr><td>Custom 1:</td><td>#Custom1#</td></tr>
  <tr><td>Column list: </td><td>#ColumnList#</td></tr>
</table></p>
</cfoutput>
```

Indexing a path returned using a query

You can index a directory path to a document (or collection of documents) using a query by retrieving a row whose contents are a full directory path name. In this case, the key specifies the column that contains the complete directory path. Documents located in the directory path are indexed using the `cfindex` tag as if they were under the web server root folder.

In this example, the `type` attribute is set to `path`, and the `key` attribute is assigned the column name Project_Docs. The Project_Docs column contains directory paths, which Verity indexes as if they were specified as a fixed path pointing to a collection of documents without the use of a query.

To index a directory path within a query:

1. Create a ColdFusion page that contains the following content:

```
<cfquery name="getEmps" datasource="cfdocexamples">
SELECT * FROM EMPLOYEE WHERE Emp_ID = 15
</cfquery>
<!--- Update the collection with the above query results. --->
<!--- Key specifies a column that contains a directory path. --->
<cfindex
  query="getEmps"
  collection="CodeColl"
  action="update"
  type="path"
  key="Project_Docs"
  title="Project_Docs"
  body="Emp_ID,FirstName,LastName,Project_Docs">

<h2>Indexing Complete</h2>
<p>Your collection now includes the following items:</p>
<cfoutput query="getEmps">
<p>#Emp_ID# #FirstName# #LastName# #Project_Docs#</p>
</cfoutput>
```

2. Save the file as indexdir.cfm in the myapps directory.

The ColdFusion `cfindex` tag indexes the contents of the specified directory path.

To search and display the directory path:

1. Create a ColdFusion page that contains the following content:

```
<cfsearch
  collection="#Form.collname#"
  name="getEmps"
  criteria="#Form.Criteria#"
  maxrows = "100">

<!--- Output the directory path contained in the record set. --->
<cfoutput>
Your search returned #getEmps.RecordCount# file(s).
</cfoutput>

<cfoutput query="getEmps">
<p><table>
  <tr><td>Title: </td><td>#Title#</td></tr>
  <tr><td>Score: </td><td>#Score#</td></tr>
  <tr><td>Key: </td><td>#Key#</td></tr>
  <tr><td>Summary: </td><td>#Summary#</td></tr>
  <tr><td>Custom 1:</td><td>#Custom1#</td></tr>
  <tr><td>Column list: </td><td>#ColumnList#</td></tr>
</table></p>
</cfoutput>
```

2. Save the file as displaydir.cfm.

Indexing query results obtained from an LDAP directory

The widespread use of the Lightweight Directory Access Protocol (LDAP) to build searchable directory structures, internally and across the web, gives you opportunities to add value to the sites that you create. You can index contact information or other data from an LDAP-accessible server and let users search it.

When creating an index from an LDAP query, remember the following considerations:

- Because LDAP structures vary greatly, you must know the server's directory schema and the exact name of every LDAP attribute that you intend to use in a query.
- The records on an LDAP server can be subject to frequent change.

In the following example, the search criterion is records with a telephone number in the 617 area code. Generally, LDAP servers use the Distinguished Name (`dn`) attribute as the unique identifier for each record so that attribute is used as the `key` value for the index.

```
<!--- Run the LDAP query. --->
<cfldap name="OrgList"
  server="myserver"
  action="query"
  attributes="o, telephonenumber, dn, mail"
  scope="onelevel"
  filter="(|(O=a*) (O=b*))"
  sort="o"
  start="c=US">

<!--- Output query record set. --->
```

```
<cfoutput query="OrgList">
  DN: #dn# <br>
  O: #o# <br>
  TELEPHONENUMBER: #telephonenumber# <br>
  MAIL: #mail# <br>
==============================<br>
</cfoutput>

<!--- Index the record set. --->
<cfindex action="update"
  collection="ldap_query"
  key="dn"
  type="custom"
  title="o"
  query="OrgList"
  body="telephonenumber">

<!--- Search the collection. --->
<!--- Use the wildcard * to contain the search string. --->
<cfsearch collection="ldap_query"
  name="s_ldap"
  criteria="*617*"
  maxrows = "100">

<!--- Output returned records. --->
<cfoutput query="s_ldap">
  #Key#, #Title#, #Body# <br>
</cfoutput>
```

Indexing cfpop query results

The contents of mail servers are generally volatile; specifically, the message number is reset as messages are added and deleted. To avoid mismatches between the unique message number identifiers on the server and in the Verity collection, you must re-index the collection before processing a search.

As with the other query types, you must provide a unique value for the `key` attribute and enter the data fields to index in the `body` attribute.

The following example updates the pop_query collection with the current mail for user1, and searches and returns the message number and subject line for all messages that contain the word *action*:

```
<!--- Run POP query. --->
<cfpop action="getall"
  name="p_messages"
  server="mail.company.com"
  userName="user1"
  password="user1">

<!--- Output POP query record set. --->
<cfoutput query="p_messages">
  #messagenumber# <br>
  #from# <br>
```

```
  #to# <br>
  #subject# <br>
  #body# <br>
<hr>
</cfoutput>

<!--- Index record set. --->
<cfindex action="refresh"
  collection="pop_query"
  key="messagenumber"
  type="custom"
  title="subject"
  query="p_messages"
  body="body">

<!--- Search messages for the word "action". --->
<cfsearch collection="pop_query"
  name="s_messages"
  criteria="action"
  maxrows = "100">

<!--- Output search record set. --->
<cfoutput query="s_messages">
  #key#, #title# <br>
</cfoutput>
```

CHAPTER 25
Using Verity Search Expressions

This chapter describes Verity search expressions and how you can refine your searches to yield the most accurate results.

Contents

About Verity query types 581
Using simple queries 582
Using explicit queries 583
Using natural queries 587
Using Internet queries 587
Composing search expressions 591
Refining your searches with zones and fields 601

About Verity query types

When you search a Verity collection, you can use a simple, explicit, natural, or Internet query. The following table compares the query types:

Query type	Content	Use of operators and modifiers	CFML example
Simple	One or more words	Uses STEM operator and MANY modifier, by default	`<cfsearch name = "band_search" collection="bbb" type = "simple" criteria="film">`
Explicit	Words, operators, modifiers	Must be specified	`<cfsearch name = "my_search" collection="bbb" type = "explicit" criteria="<WILDCARD>'sl[iau]m'">`

Query type	Content	Use of operators and modifiers	CFML example
Natural	One or more words	Uses STEM operator and MANY modifier, by default	`<cfsearch name = "my_search" collection="bbb" type = "natural" criteria="Boston subway maps">`
Internet	Words, operators, modifiers		`<cfsearch name = "my_search" collection="bbb" type = "Internet" criteria="Boston subway maps">`

The query type determines whether the search words that you enter are stemmed, and whether the retrieved words contribute to relevance-ranked scoring. Both of these conditions occur by default in simple queries. For more information on the STEM operator and MANY modifier, see "Stemming in simple queries" on page 583.

Note: Operators and modifiers are formatted as uppercase letters in this chapter solely to enhance legibility. They might be all lowercase or uppercase.

Using simple queries

The simple query is the default query type and is appropriate for the vast majority of searches. When entering text on a search form, you perform a simple query by entering a word or comma-delimited strings, with optional wildcard characters. Verity treats each comma as a logical OR. If you omit the commas, Verity treats the expression as a phrase.

Caution: Many web search engines assume a logical AND for multiple word searches, and search for a phrase only if you use quotation marks. Because Verity treats multiple word searches differently, it might help your users if you provide examples on your search page or a brief explanation of how to search.

The following table shows examples of simple searches:

Example	Search result
low,brass,instrument	low or brass or instrument
low brass instrument	the phrase, low brass instrument
film	film, films, filming, or filmed
filming AND fun	film, films, filming, or filmed, and fun
filming OR fun	film, films, filming, or filmed, or fun
filming NOT fun	film, films, filming, or filmed, but not fun

The operators AND and OR, and the modifier NOT, do not require angle brackets (<>). Operators typically require angle brackets and are used in explicit queries. For more information about operators and modifiers, see "Operators and modifiers" on page 592.

Stemming in simple queries

By default, Verity interprets words in a simple query as if you entered the STEM operator (and MANY modifier). The STEM operator searches for words that derive from a common stem. For example, a search for instructional returns files that contain instruct, instructs, instructions, and so on.

The STEM operator works on words, not word fragments. A search for "instrument" returns documents containing "instrument," "instruments," "instrumental," and "instrumentation," whereas a search for "instru" does not. (A wildcard search for instru* returns documents with these words, and also those with instruct, instructional, and so on.)

Note: The MANY modifier presents the files returned in the search as a list based on a relevancy score. A file with more occurrences of the search word has a higher score than a file with fewer occurrences. As a result, the search engine ranks files according to word density as it searches for the word that you specify, as well as words that have the same stem. For more information on the MANY modifier, see "Modifiers" on page 601.

In CFML, enter your search terms, operators, and modifiers in the `criteria` attribute of the `cfsearch` tag:

```
<cfsearch name="search_name"
  collection="bbb"
  type="simple"
criteria="instructional">
```

Preventing stemming

When entering text on a search form, you can prevent Verity from implicitly adding the STEM operator by doing one of the following:

- Perform an explicit query. For more information, see the next section, "Using explicit queries" on page 583.
- Use the WORD operator. For more information, see "Operators" on page 592.
- Enclose the search term that has double-quotation marks with single-quotation marks, as follows:

```
<cfsearch name="search_name"
  collection="bbb"
  type="simple"
criteria='"instructional"'
```

Using explicit queries

In an explicit query, the Verity search engine literally interprets your search terms. The following are two ways to perform an explicit query:

- On a search form, use quotation marks around your search term(s).
- In CFML, use `type="explicit"` in the `cfsearch` tag.

When you put a search term in quotation marks, Verity does not use the STEM operator. For example, a search for "instructional"—enclosed in quotation marks, as shown in "Preventing stemming" on page 583—does not return files that contain instruct, instructs, instructions, and so on (unless the files also contain instructional).

Note: The Verity products and documentation refers to the Explicit parser as the BooleanPlus parser.

When you specify `type="explicit"` the search expression must be a valid Verity Query Language expression. As a result, an individual search term must be in explicit quotation marks. The following table shows valid and invalid criteria:

Attribute	Effect
`criteria="government"`	Generates an error
`criteria="'government'" or criteria='"government"'`	Finds only government
`criteria="<WORD>government"`	Finds only government
`criteria="<STEM>government"`	Finds government, governments, and governmental
`criteria="<MANY><STEM>government"`	Finds government, governments, and governmental ranked by relevance
`criteria="<WILDCARD>governmen*"`	Finds government, governments, and governmental

Using AND, OR, and NOT

Verity has many powerful operators and modifiers available for searching (for more information, see "Operators and modifiers" on page 592). However, users might only use the most basic operators—AND, OR, and the modifier NOT. The following are a few important points:

- You can type operators in uppercase or lowercase letters.
- Verity reads operators from left to right.
- The AND operator takes precedence over the OR operator.
- Use parentheses to clarify the search. Terms enclosed in parentheses are evaluated first; innermost parentheses are evaluated first when there are nested parentheses.
- To search for a literal AND, OR, or NOT, enclose the literal term in double-quotation marks; for example:

  ```
  love "and" marriage
  ```

Note: Although NOT is a modifier, you use it only with the AND and OR operators. Therefore, it is sometimes casually referred to as an operator.

The following table gives examples of searches and their results:

Search term	Returns files that contain
doctorate AND nausea	both doctorate and nausea
doctorate "and" nausea	the phrase doctorate and nausea
"doctorate and nausea"	the phrase doctorate and nausea

Search term	Returns files that contain
masters OR doctorate AND nausea	masters, or the combination of doctorate and nausea
masters OR (doctorate AND nausea)	masters, or the combination of doctorate and nausea
(masters OR doctorate) AND nausea	either masters or doctorate, and nausea
masters OR doctorate NOT nausea	either masters or doctorate, but not nausea

Using wildcards and special characters

Part of the strength of the Verity search is its use of wildcards and special characters to refine searches. Wildcard searches are especially useful when you are unsure of the correct spelling of a term. Special characters help you search for tags in your code.

Searching with wildcards

The following table shows the wildcard characters that you can use to search Verity collections:

Wildcard	Description	Example	Search result
?	Matches any single alphanumeric character.	apple?	apples or applet
*	Matches zero or more alphanumeric characters. Avoid using the asterisk as the first character in a search string. An asterisk is ignored in a set, ([]) or an alternative pattern ({}).	app*ed	Appleseed, applied, appropriated, and so on
[ ]	Matches any one of the characters in the brackets. Square brackets indicate an implied OR.	‹WILDCARD› 'sl[iau]m'	slim, slam, or slum
{ }	Matches any one of a set of patterns separated by a comma,	‹WILDCARD› 'hoist{s,ing,ed}'	hoists, hoisting, or hoisted
^	Matches any character not in the set.	‹WILDCARD›'sl[^ia]m'	slum, but not slim or slam
-	Specifies a range for a single character in a set.	‹WILDCARD› 'c[a-r]t'	cat, cot, but not cut (that is, every word beginning with c, ending with t, and containing any single letter from a to r)

To search for a wildcard character as a literal, place a backslash character before it; for example:

- To match a question mark or other wildcard character, precede the ? with one backslash. For example, type the following in a search form: Checkers\?
- To match a literal asterisk, you precede the * with two backslashes, and enclose the search term with either single or double quotation marks. For example, type the following in a search form: 'M*' (or "M*") The following is the corresponding CFML code:

```
<cfsearch name = "quick_search"
  collection="bbb"
  type = "simple"
  criteria="'M\\*'">
```

Note: The last line is equivalent to `criteria='"M\\*"'>`.

Searching for special characters

The search engine handles a number of characters in particular ways as the following table describes:

Characters	Description
`,()[`	These characters end a text token. A *token* is a variable that stores configurable properties. It lets the administrator or user configure various settings and options.
`=><!`	These characters also end a text token. They are terminated by an associated end character.
`'`<{[!`	These characters signify the start of a delimited token. They are terminated by an associated end character.

To search for special characters as literals, precede the following nonalphanumeric characters with a backslash character (\) in a search string:

- comma (,)
- left parenthesis (
- right parenthesis)
- double-quotation mark (")
- backslash (\)
- left curly brace ({)
- left bracket ([)
- less than sign (<)
- backquote (`)

In addition to the backslash character, you can use paired backquote characters (` `) to interpret special characters as literals. For example, to search for the wildcard string “a{b” you can surround the string with back quotation marks, as follows:

```
`a{b`
```

To search for a wildcard string that includes the literal backquote character (`) you must use two backquote characters together and surround the entire string in back quotation marks:

```
`*n``t`
```

You can use paired back quotation marks or backslashes to escape special characters. There is no functional difference between the two. For example, you can query for the term: <DDA> using `\<DDA\>` or `` `<DDA>` `` as your search term.

Using natural queries

The Natural parser supports searching for similar documents, a search method sometimes referred to as similarity searching. The Natural parser supports searching the full text of documents only. The Natural parser does not support searching collection fields and zones. The Natural parser does not support Verity query language except for topics.

Note: The Verity products and documentation refer to the Natural parser as the Query-By-Example parser, as well as the Free Text parser.

Meaningful words are automatically treated as if they were preceded by the MANY modifier and the STEM operator. By implicitly applying the STEM operator, the search engine searches not only for the meaningful words themselves, but also for words that have the same stem. By implicitly applying the MANY modifier, Verity calculates each document's score based on the word density it finds for meaningful words; the denser the occurrences of a word in a document, the higher the document's score.

By default, common words (such as *the*, *has*, and *for*) are stripped away, and the query is built based on the more significant words (such as *personnel*, *interns*, *schools*, and *mentors*). Therefore, the results of a natural language search are likely to be less precise than a search performed using the simple or explicit parser.

The Natural parser interprets topic names as topic objects. This means that if the specified text block contains a topic name, the query expression represented by the topic is considered in the search.

Using Internet queries

With the Internet query parser, users can search entire documents or parts of documents (zones and fields) entering words, phrases, and plain language similar to that used by many web search engines. ColdFusion MX supports two Internet query parsers in the `cfsearch type` attribute.

Internet Uses standard, web-style query syntax. For more information, see "Query syntax" on page 589.

Internet_basic Similar to Internet. This query parser enhances performance, but produces less accurate relevancy statistics.

Note: Verity also includes the Internet_BasicWeb and Internet_AdvancedWeb query parsers, which are not directly supported by ColdFusion MX.

Search terms

In a search form enabled with the Internet query parser, users can enter words, phrases, and plain language. The Internet parser does not support the Verity query language (VQL).

Words

To search for multiple words, separate them with spaces.

Phrases

To search for an exact phrase, surround it with double-quotation marks. A string of capitalized words is assumed to be a name. Separate a series of names with commas. Commas aren't needed when the phrases are surrounded by quotation marks.

The following example searches for a document that contains the phrases "San Francisco" and "sourdough bread":

```
"San Francisco" "sourdough bread"
```

Plain language

To search with plain language, enter a question or concept. The Internet Query Parser identifies the important words and searches for them. For example, enter a question such as:

```
Where is the sales office in San Francisco?
```

This query produces the same results as entering:

```
sales office San Francisco
```

Including and excluding search terms

You can limit searches by excluding or requiring search terms, or by limiting the areas of the document that are searched.

A minus sign (–) immediately preceding a search term (word or phrase) excludes documents containing the term.

A plus sign (+) immediately preceding a search term (word or phrase) means returned documents are guaranteed to contain the term.

If neither sign is associated with the search term, the results may include documents that do not contain the specified term as long as they meet other search criteria.

Field searches

The Internet parser lets users perform field searches. The fields that are available for searching depend on field extraction rules based on the document type of the documents in the collection.

To search a document field, type the name of the field, a colon (:), and the search term with no spaces.

```
field:term
```

If you enter a minus sign (–) immediately preceding field, documents that contain the specified term are excluded from the search results. For example, if you enter `-field:term`, documents that contain the specified term in the specified field are excluded from the results of the search.

If you enter a plus sign (+) immediately proceeding the field search specification, such as `+field:term`, documents are included in the search results only if the search term is present in the specified field.

Field searches are enabled by the `enableField` parameter in a template file. This parameter, set to 0 by default, must be set to 1 to allow searching a document field.

Caution: The `enableField` parameter is the only thing in a template file that should be modified.

Query syntax

The query syntax is very similar to the syntax that users expect to use on the web. Queries are interpreted according to the following rules:

- Individual search terms are separated by whitespace characters, such as a space, tab, or comma; for example:

  cake recipes
- Search phrases are entered within double-quotation marks; for example:

  "chocolate cake" recipe
- Exclude terms with the negation operator, minus (-), or the NOT operator; for example:

  cake recipes -rum

  cake recipes NOT rum
- Require a compulsory term with the unary inclusion operator, plus sign (+); in this example, the term *chocolate* must be included:

  cake recipes +chocolate
- Require compulsory terms with the binary inclusion operator AND; in this example, the terms *recipes* and *chocolate* must be included:

  cake recipes and chocolate

Field searches

You can search fields or zones by specifying `name: term`, where:

`name` is the name of the field or zone

`term` is an individual search term or phrase

For example:

```
bakery city:"San Francisco"
bakery city:Sunnyvale
```

For more information, see "Refining your searches with zones and fields" on page 601.

Pass-through of terms

Search terms are passed through to the VDK-level and are interpreted as Verity Query Language (VQL) syntax. No issues arise if the terms contain only alphabetic or numeric characters. Other kinds of characters might be interpreted by the language you're using. If a term contains a character that is not handled by the specified language, it might be interpreted as VQL. For example, a search term that includes an asterisk (*) might be interpreted as a wildcard.

Stop words

The configurable Internet query parser uses its own stop-word list, qp_inet.stp, to specify terms to ignore for natural language processing.

Note: You can override the "stop out" by using quotation marks around the word.

For example, the following stop words are provided in the query parser's stop-word file for the English (Basic) template:

a	did	i	or	what
also	do	i'm	should	when
an	does	if	so	where
and	find	in	than	whether
any	for	is	that	which
am	from	it	the	who
are	get	its	there	whose
as	got	it's	to	why
at	had	like	too	will
be	has	not	want	with
but	have	of	was	would
can	how	on	were	<or>

Verity provides a populated stop-word file for the English and English (Advanced) languages. You do not need to modify the qp_inet.stp file for these languages. If you use the configurable Internet query parser for another language, you must provide your own qp_inet.stp file that contains the stop words that you want to ignore in that language. This stop-word file must contain, at a minimum, the language-equivalent words for *or* and *<or>*.

Note: The configurable Internet query parser's stop-word file contains a different word list than the vdk30.stp word file, which is used for other purposes, such as summarization.

Composing search expressions

The following rules apply to the composition of search expressions.

Case sensitivity

Verity searches are case-sensitive only when the search term is entered in mixed case. For example, a search for *zeus* finds zeus, Zeus, or ZEUS; however, a search for *Zeus* finds only Zeus.

To have your application always ignore the case that the user types, use the ColdFusion `LCase()` function in the `criteria` attribute of `cfsearch`. The following code converts user input to lowercase, thereby eliminating case-sensitivity concerns:

```
<cfsearch name="results"
  collection="#form.collname#"
  criteria="#LCase(form.criteria)#"
  type="#form.type#">
```

Prefix and infix notation

By default, Verity uses *infix notation*, in which precedence is implicit in the expression; for example, the AND operator takes precedence over the OR operator.

You can use *prefix notation* with any operator except an evidence operator (typically, STEM, WILDCARD, or WORD; for a description of evidence operators, see "Evidence operators" on page 597). In prefix notation, the expression explicitly specifies precedence. Rather than repeating an operator, you can use prefix notation to list the operator once and list the search targets in parentheses. For example, the following expressions are equivalent:

- Moses <NEAR> Larry <NEAR> Jerome <NEAR> Daniel <NEAR> Jacob
- <NEAR>(Moses,Larry,Jerome,Daniel,Jacob)

The following prefix notation example searches first for documents that contain Larry and Jerome, and then for documents that contain Moses:

OR (Moses, AND (Larry,Jerome))

The infix notation equivalent of this is as follows:

Moses OR (Larry AND Jerome)

Commas in expressions

If an expression includes two or more search terms within parentheses, a comma is required between the elements (whitespace is ignored). The following example searches for documents that contain any combination of Larry and Jerome together:

AND (Larry, Jerome)

Precedence rules

Expressions are read from left to right. The AND operator takes precedence over the OR operator; however, terms enclosed in parentheses are evaluated first. When the search engine encounters nested parentheses, it starts with the innermost term.

Example	Search result
Moses AND Larry OR Jerome	documents that contain Moses and Larry, or Jerome
(Moses AND Larry) OR Jerome	(same as above)
Moses AND (Larry OR Jerome)	documents that contain Moses and either Larry or Jerome

Delimiters in expressions

You use angle brackets (< >), double quotation marks ("), and backslashes (\) to delimit various elements in a search expression, as the following table describes:

Character	Usage
‹ ›	Left and right angle brackets are reserved for designating operators and modifiers. They are optional for the AND, OR, and NOT, but required for all other operators.
"	Use double quotation marks in expressions to search for a word that is otherwise reserved as an operator or modifier, such as AND, OR, and NOT.
\	To include a backslash in a search expression, insert two backslashes for each backslash character that you want included in the search; for example, C:\\CFusionmMX7\\bin.

Operators and modifiers

You are probably familiar with searches containing AND, OR, and NOT. Verity has many additional operators and modifiers, of various types, that offer you a high degree of specificity in setting search parameters.

Operators

An *operator* represents logic to be applied to a search element. This logic defines the qualifications that a document must meet to be retrieved. You can use operators to refine your search or to influence the results in other ways.

For example, you can construct an HTML form for conducting searches. In the form, you can search for a single term. You can refine the search by limiting the search scope in a number of ways. Operators are available for limiting a query to a sentence or paragraph, and you can search words based on proximity.

Ordinarily, you use operators in explicit searches, as follows:

```
"<operator>search_string"
```

The following operator types are available:

Operator type	Purpose
Concept	Identifies a concept in a document by combining the meanings of search elements.
Relational	Searches fields in a collection.
Evidence	Specifies basic and intelligent word searches.
Proximity	Specifies the relative location of words in a document.
Score	Manipulates the score returned by a search element. You can set the score percentage display to four decimal places.

The following table shows the operators, according to type, that are available for conducting searches of ColdFusion Verity collections:

Concept	Relational	Evidence	Proximity	Score
ACCRUE	<	STEM	NEAR	YESNO
ALL	<=	WILDCARD	NEAR/N	PRODUCT
AND	=	WORD	PARAGRAPH	SUM
ANY	>	THESAURUS	PHRASE	COMPLEMENT
OR	>=	SOUNDEX	SENTENCE	
	CONTAINS	TYPO/N	IN	
	MATCHES			
	STARTS			
	ENDS			
	SUBSTRING			

Concept operators

Concept operators combine the meaning of search elements to identify a concept in a document. Documents retrieved using concept operators are ranked by relevance. The following table describes each concept operator:

Operator	Description
AND	Selects documents that contain all the search elements that you specify.
OR	Selects documents that show evidence of at least one of the search elements that you specify.
ACCRUE	Selects documents that include at least one of the search elements that you specify. Documents are ranked based on the number of search elements found.

Operator	Description
ALL	Selects documents that contain all of the search elements that you specify. A score of 1.00 is assigned to each retrieved document. ALL and AND retrieve the same results, but queries using ALL are always assigned a score of 1.00.
ANY	Selects documents that contain at least one of the search elements that you specify. A score of 1.00 is assigned to each retrieved document. ANY and OR retrieve the same results, but queries using ANY are always assigned a score of 1.00.

Relational operators

Relational operators search document fields (such as AUTHOR) that you defined in the collection. Documents that contain specified field values are returned. Documents retrieved using relational operators are not ranked by relevance, and you cannot use the MANY modifier with relational operators.

You use the following operators for numeric and date comparisons:

Operator	Description
=	Equal
!=	Not equal
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

For example, to search for documents that contain values for 1999 through 2002, you perform either of the following searches:

- A simple search for 1999,2000,2001,2002
- An explicit search using the = operator: >=1999,<=2002

If a document field named PAGES is defined, you can search for documents that are 5 pages or fewer by entering **PAGES < 5** in your search. Similarly, if a document field named DATE is defined, you can search for documents dated prior to and including December 31, 1999 by entering **DATE <= 12-31-99** in your search.

The following relational operators compare text and match words and parts of words:

Operator	Description	Example
CONTAINS	Selects documents by matching the word or phrase that you specify with the values stored in a specific document field. Documents are selected only if the search elements specified appear in the same sequential and contiguous order in the field value.	• In a document field named TITLE, to retrieve documents whose titles contain music, musical, or musician, search for *TITLE ‹CONTAINS› Musi**. • To retrieve CFML and HTML pages whose meta tags contain Framingham as a content word, search for *KEYWORD ‹CONTAINS› Framingham*.
MATCHES	Selects documents by matching the query string with values stored in a specific document field. Documents are selected only if the search elements specified match the field value exactly. If a partial match is found, a document is not selected. When you use the MATCHES operator, you specify the field name to search, and the word, phrase, or number to locate. You can use ? and * to represent individual and multiple characters, respectively, within a string.	For examples, see the text immediately following this table.
STARTS	Selects documents by matching the character string that you specify with the starting characters of the values stored in a specific document field.	In a document field named REPORTER, to retrieve documents written by Clark, Clarks, and Clarkson, search for *REPORTER ‹STARTS› Clark*.
ENDS	Selects documents by matching the character string that you specify with the ending characters of the values stored in a specific document field.	In a document field named OFFICER, to retrieve arrest reports written by Tanner, Garner, and Milner, search for *OFFICER ‹ENDS› ner*.
SUBSTRING	Selects documents by matching the query string that you specify with any portion of the strings in a specific document field.	In a document field named TITLE, to retrieve documents whose titles contain words such as solution, resolution, solve, and resolve, search for *TITLE ‹SUBSTRING› sol*.

For example, assume a document field named SOURCE includes the following values:

- Computer
- Computerworld
- Computer Currents
- PC Computing

To locate documents whose source is Computer, enter the following:

```
SOURCE <MATCHES> computer
```

To locate documents whose source is Computer, Computerworld, and Computer Currents, enter the following:

```
SOURCE <MATCHES> computer*
```

To locate documents whose source is Computer, Computerworld, Computer Currents, and PC Computing, enter the following:

```
SOURCE <MATCHES> *comput*
```

For an example of ColdFusion code that uses the CONTAINS relational operator, see "Field searches" on page 603.

You can use the SUBSTRING operator to match a character string with data stored in a specified data source. In the example described in this section, a data source called TEST1 contains the table YearPlaceText, which contains three columns: Year, Place, and Text. Year and Place make up the primary key. The following table shows the TEST1 schema:

Year	Place	Text
1990	Utah	Text about Utah 1990
1990	Oregon	Text about Oregon 1990
1991	Utah	Text about Utah 1991
1991	Oregon	Text about Oregon 1991
1992	Utah	Text about Utah 1992

The following application page matches records that have 1990 in the TEXT column and are in the Place Utah. The search operates on the collection that contains the TEXT column and then narrows further by searching for the string *Utah* in the CF_TITLE document field. Document fields are defaults defined in every collection corresponding to the values that you define for URL, TITLE, and KEY in the `cfindex` tag.

```
<cfquery name="GetText"
  datasource="TEST1">
  SELECT Year+Place AS Identifier, text
    FROM YearPlaceText
</cfquery>

<cfindex collection="testcollection"
  action="Update"
  type="Custom"
  title="Identifier"
  key="Identifier"
  body="TEXT"
  query="GetText">

<cfsearch name="GetText_Search"
  collection="testcollection"
  type="Explicit"
  criteria="1990 and CF_TITLE <SUBSTRING> Utah">
<cfoutput>
  Record Counts: <br>
  #GetText.RecordCount# <br>
```

```
  #GetText_Search.RecordCount# <br>
</cfoutput>

Query Results --- Should be 5 rows <br>
<cfoutput query="Gettext">
  #Identifier# <br>
</cfoutput>

Search Results -- should be 1 row <br>
<cfoutput query="GetText_Search">
  #GetText_Search.TITLE# <br>
</cfoutput>
```

Evidence operators

Evidence operators let you specify a basic word search or an intelligent word search. A *basic word search* finds documents that contain only the word or words specified in the query. An *intelligent word search* expands the query terms to create an expanded word list so that the search returns documents that contain variations of the query terms.

Documents retrieved using evidence operators are not ranked by relevance unless you use the MANY modifier.

The following table describes the evidence operators:

Operator	Description	Example
STEM	Expands the search to include the word that you enter and its variations. The STEM operator is automatically implied in any simple query.	*‹STEM›believe* retrieves matches such as "believe," "believing," and "believer".
WILDCARD	Matches wildcard characters included in search strings. Certain characters automatically indicate a wildcard specification, such as apostrophe (*) and question mark(?).	*spam** retrieves matches such as, spam, spammer, and spamming.
WORD	Performs a basic word search, selecting documents that include one or more instances of the specific word that you enter. The WORD operator is automatically implied in any SIMPLE query.	*‹WORD› logic* retrieves logic, but not variations such as logical and logician.
THESAURUS	Expands the search to include the word that you enter and its synonyms. Collections do not have a thesaurus by default; to use this feature you must build one.	*‹THESAURUS› altitude* retrieves documents containing synonyms of the word altitude, such as height or elevation.

Operator	Description	Example
SOUNDEX	Expands the search to include the word that you enter and one or more words that "sound like," or whose letter pattern is similar to, the word specified. Collections do not have sound-alike indexes by default; to use this feature you must build sound-alike indexes.	*‹SOUNDEX› sale* retrieves words such as sale, sell, seal, shell, soul, and scale.
TYPO/N	Expands the search to include the word that you enter plus words that are similar to the query term. This operator performs "approximate pattern matching" to identify similar words. The optional N variable in the operator name expresses the maximum number of errors between the query term and a matched term, a value called the error distance. If N is not specified, the default error distance is 2.	*‹TYPO› swept* retrieves kept.

The following example uses an evidence operator:

```
<cfsearch name = "quick_search"
  collection="bbb"
  type = "explicit"
  criteria="<WORD>film">
```

Proximity operators

Proximity operators specify the relative location of specific words in the document. To retrieve a document, the specified words must be in the same phrase, paragraph, or sentence. In the case of NEAR and NEAR/N operators, retrieved documents are ranked by relevance based on the proximity of the specified words. Proximity operators can be nested; phrases or words can appear within SENTENCE or PARAGRAPH operators, and SENTENCE operators can appear within PARAGRAPH operators.

The following table describes the proximity operators:

Operator	Description	Example
NEAR	Selects documents containing specified search terms. The closer the search terms are to one another within a document, the higher the document's score. The document with the smallest possible region containing all search terms always receives the highest score. Documents whose search terms are not within 1000 words of each other are not selected.	*war ‹NEAR› peace* retrieves documents that contain stemmed variations of these words within close proximity to each other (as defined by Verity). To control search proximity, use NEAR/N.
NEAR/N	Selects documents containing two or more search terms within N number of words of each other, where N is an integer between 1 and 1024. NEAR/1 searches for two words that are next to each other. The closer the search terms are within a document, the higher the document's score. You can specify multiple search terms using multiple instances of NEAR/N as long as the value of N is the same.	*commute ‹NEAR/10› bicycle ‹NEAR/10› train ‹NEAR/10›* retrieves documents that contain stemmed variations of these words within 10 words of each other.
PARAGRAPH	Selects documents that include all of the words you specify within the same paragraph. To search for three or more words or phrases in a paragraph, you must use the PARAGRAPH operator between each word or phrase.	*‹PARAGRAPH› (mission, goal, statement)* retrieves documents that contain these terms within a paragraph.
PHRASE	Selects documents that include a phrase you specify. A phrase is a grouping of two or more words that occur in a specific order.	*‹PHRASE› (mission, oak)* returns documents that contain the phrase mission oak.
SENTENCE	Selects documents that include all of the words you specify within the same sentence.	*‹SENTENCE› (jazz, musician)* returns documents that contain these words in the same sentence.
IN	Selects documents that contain specified values in one or more document zones. A document zone represents a region of a document, such as the document's summary, date, or body text. To search for a term only within the one or more zones that have certain conditions, you qualify the IN operator with the WHEN operator.	*Chang ‹IN› author* searches document zones named author for the word Chang.

The following example uses a proximity operator:

```
<cfsearch name = "quick_search"
  collection="bbb"
  type = "explicit"
  criteria="red<near>socks">
```

For an example using the IN proximity operator to search XML documents , see "Zone searches" on page 602.

Score operators

Score operators control how the search engine calculates scores for retrieved documents. The maximum score that a returned search element can have is 1.000. You can set the score to display a maximum of four decimal places.

When you use a score operator, the search engine first calculates a separate score for each search element found in a document, and then performs a mathematical operation on the individual element scores to arrive at the final score for each document.

The document's score is available as a result column. You can use the SCORE result column to get the relevancy score of any document retrieved; for example:

```
<cfoutput>
  <a href="#Search1.URL#">#Search1.Title#</a><br>
  Document Score=#Search1.SCORE#<BR>
</cfoutput>
```

The following table describes the score operators:

Operator	Description	Example
YESNO	Forces the score of an element to 1 if the element's score is nonzero.	*‹YESNO›mainframe*. If the retrieval result of the search on mainframe is 0.75, the YESNO operator forces the result to 1. You can use YESNO to avoid relevance ranking.
PRODUCT	Multiplies the scores for the search elements in each document matching a query.	*‹PRODUCT›(computers, laptops)* takes the product of the resulting scores.
SUM	Adds the scores for the search element in each document matching a query, up to a maximum value of 1.	*‹SUM›(computers, laptops)* takes the sum of the resulting scores.
COMPLEMENT	Calculates scores for documents matching a query by taking the complement (subtracting from 1) of the scores for the query's search elements. The new score is 1 minus the search element's original score.	*‹COMPLEMENT›computers*. If the search element's original score is .785, the COMPLEMENT operator recalculates the score as .215.

Modifiers

You combine modifiers with operators to change the standard behavior of an operator in some way. The following table describes the available modifiers:

Modifier	Description	Example
CASE	Specifies a case-sensitive search. Normally, Verity searches are case-insensitive for search text entered in all uppercase or all lowercase, and case-sensitive for mixed-case search strings.	*‹CASE›Java OR ‹CASE›java* retrieves documents that contain Java or java, but not JAVA.
MANY	Counts the density of words, stemmed variations, or phrases in a document and produces a relevance-ranked score for retrieved documents. Use with the following operators: • WORD • WILDCARD • STEM • PHRASE • SENTENCE • PARAGRAPH	*‹PARAGRAPH›‹MANY›javascript ‹AND› vbscript*. You cannot use the MANY modifier with the following operators: • AND • OR • ACCRUE • Relational operators
NOT	Excludes documents that contain the specified word or phrase. Use only with the AND and OR operators.	*Java ‹AND› programming ‹NOT› coffee* retrieves documents that contain Java and programming, but not coffee.
ORDER	Specifies that the search elements must occur in the same order in which you specify them in the query. Use with the following operators: • PARAGRAPH • SENTENCE • NEAR/N Place the ORDER modifier before any operator.	*‹ORDER›‹PARAGRAPH› ("server", "Java")* retrieves documents that contain server before Java.

Refining your searches with zones and fields

One of the strengths of Verity is its ability to perform full-text searches on documents of many formats. However, there are often times when you want to restrict a search to certain portions of a document, to improve search relevance. If a Verity collection contains some documents about baseball and other documents about caves, a search for the word bat might retrieve several irrelevant results.

If the documents are structured documents, you can take advantage of the ability to search zones and fields. The following are some examples of structured documents:

- Documents created with markup languages (XML, SGML, HTML)
- Internet Message Format documents
- Documents created by many popular word-processing applications

Note: Although your word processor might open with what appears to be a blank page, the document has many regions such as title, subject, and author. Refer to your application's documentation or online help system for how to view a document's properties.

Zone searches

You can perform zone searches on markup language documents. The Verity zone filter includes built-in support for HTML and several file formats; for a list of supported file formats, see "Building a Search Interface" on page 547. Verity searches XML files by treating the XML tags as zones. When you use the zone filter, the Verity engine builds zone information into the collection's full-word index. This index, enhanced with zone information, permits quick and efficient searches over zones. The zone filter can automatically define a zone, or you can define it yourself in the style.zon file. You can use zone searching to limit your search to a particular zone. This can produce more accurate, but not necessarily faster, search results than searching an entire file.

Note: The contents of a zone cannot be returned in the results list of an application.

Examples

The following examples perform zone searching on XML files. In a list of rock bands, you could have XML files with tags for the instruments and for comments. In the following XML file, the word Pete appears in a comment field:

```
<band.xml>
  <Lead_Guitar>Dan</Lead_Guitar>
  <Rhythm_Guitar>Jake</Rhythm_Guitar>
  <Bass_Guitar>Mike</Bass_Guitar>
  <Drums>Chris</Drums>
  <COMMENT_A>Dan plays guitar, better than Pete.</COMMENT_A>
  <COMMENT_B>Jake plays rhythm guitar.</COMMENT_B>
</band.xml>
```

The following CFML code shows a search for the word Pete:

```
<cfsearch name = "band_search"
  collection="my_collection"
  type = "simple"
criteria=”Pete”>
```

The above search for Pete returns this XML file because this search target is in the COMMENT_A field. In contrast, Pete is the lead guitarist in the following XML file:

```
<band.xml>
  <Lead_Guitar>Pete</Lead_Guitar>
  <Rhythm_Guitar>Roger</Rhythm_Guitar>
  <Bass_Guitar>John</Bass_Guitar>
  <Drums>Kenny</Drums>
  <COMMENT_A>Who knows who's better than this band?</COMMENT_A>
  <COMMENT_B>Ticket prices correlated with decibels.</COMMENT_B>
</band.xml>
```

To retrieve only the files in which Pete is the lead guitarist, perform a zone search using the IN operator according to the following syntax:

```
(query) <IN> (zone1, zone2, ...)
```

Note: As with other operators, IN might be uppercase or lowercase. Unlike AND, OR, or NOT, you must enclose IN within brackets.

Thus, the following explicit search retrieves files in which Pete is the lead guitarist:

```
(Pete) <in> Lead_Guitar
```

This is expressed in CFML as follows:

```
<cfsearch name = "band_search"
  collection="my_collection"
  type = "explicit"
  criteria="(Pete) <in> Lead_Guitar">
```

To retrieve files in which Pete plays either lead or rhythm guitar, use the following explicit search:

```
(Pete) <in> (Lead_Guitar,Rhythm_Guitar)
```

This is expressed in CFML as follows:

```
<cfsearch name = "band_search"
  collection="bbb"
  type = "explicit"
  criteria="(Pete) <in> (Lead_Guitar,Rhythm_Guitar)">
```

Field searches

Fields are extracted from the document and stored in the collection for retrieval and searching, and can be returned on a results list. Zones, on the other hand, are merely the definitions of "regions" of a document for searching purposes, and are not physically extracted from the document in the same way that fields are extracted.

You must define a region of text as a zone before it can be a field. Therefore, it can be only a zone, or it can be both a field and a zone. Whether you define a region of text as a zone only or as both a field and a zone depends on your particular requirements.

A field must be defined in the style file, style.ufl, before you create the collection. To map zones to fields (to display field data), you must define and add these extra fields to style.ufl.

You can specify the values for the `cfindex` attributes TITLE, KEY, and URL as document fields for use with relational operators in the `criteria` attribute. (The SCORE and SUMMARY attributes are automatically returned by a `cfsearch`; these attributes are different for each record of a collection as the search criteria changes.) Text comparison operators can reference the following document fields:

- cf_title
- cf_key
- cf_url
- cf_custom1
- cf_custom2
- cf_custom3
- cf_custom4

Text comparison operators can also reference the following automatically populated document fields:

- title
- key
- url
- vdksummary
- author
- mime-type

To explore how to use document fields to refine a search, consider the following database table, named Calls. This table has four columns and three records, as the following table shows:

call_ID	Problem_Description	Short_Description	Product
1	Can't bold text properly under certain conditions	Bold Problem	HomeSite+
2	Certain optional attributes are acting as required attributes	Attributes Problem	ColdFusion
3	Can't do a File/Open in certain cases	File Open Problem	HomeSite+

A Verity search for the word certain returns three records. However, you can use the document fields to restrict your search; for example, a search to retrieve HomeSite+ problems with the word certain in the problem description.

These are the requirements to run this procedure:

- Create and populate the Calls table in a database of your choice
- Create a collection named Training (you can do this in CFML or in the ColdFusion MX Administrator).

The following table shows the relationship between the database column and `cfindex` attribute:

Database column	The cfindex attribute	Comment
call_ID	key	The primary key of a database table is often a key attribute.
Problem_Description	body	This column is the information to be indexed.
Short_Description	title	A short description is conceptually equivalent to a title, as in a running title of a journal article.
Product	custom1	This field refines the search.

You begin by selecting all data in a query:

```
<cfquery name = "Calls" datasource = "MyDSN">
  Select * from Calls
</cfquery>
```

The following code shows the `cfindex` tag for indexing the collection (the `type` attribute is set to custom for tabular data):

```
<cfindex
  query = "Calls"
  collection = "training"
  action = "UPDATE"
  type = "CUSTOM"
  title = "Short_Description"
  key = "Call_ID"
  body = "Problem_Description"
  custom1 = "Product">
```

To perform the refined search for HomeSite+ problems with the word certain in the problem description, the `cfsearch` tag uses the CONTAINS operator in its `criteria` attribute:

```
<cfsearch
  collection = "training"
  name = "search_calls"
  criteria = "certain and CF_CUSTOM1 <CONTAINS> HomeSite">
```

The following code displays the results of the refined search:

```
<table border="1" cellspacing="5">
<tr>
  <th align="LEFT">KEY</th>
  <th align="LEFT">TITLE</th>
  <th align="LEFT">CUSTOM1</th>
</tr>

<cfoutput query = "search_calls">
<tr>
  <td>#KEY#</td>
  <td>#TITLE#</td>
  <td>#CUSTOM1#</td>
</tr>
</cfoutput>
</table>
```

PART V
Requesting and Presenting Information

This part describes how to dynamically request information from users and display information on their browsers. It includes information on using the HTML form tag, CFML `cfform` tag, and other ColdFusion tags to request data from users; how to use the `cfchart` tag to graphically display data; and how to use the Flash Remoting service to provide information to Macromedia Flash applications for display.

The following chapters are included:

Chapter 26: Introduction to Retrieving and Formatting Data 609
Chapter 27: Building Dynamic Forms with cfform Tags 631
Chapter 28: Validating Data 659
Chapter 29: Creating Forms in Macromedia Flash 687
Chapter 30: Creating Skinnable XML Forms 709
Chapter 31: Creating Charts and Graphs 733
Chapter 32: Creating Reports for Printing 763
Chapter 33: Using the Flash Remoting Service 807
Chapter 34: Using Server-Side ActionScript 823

CHAPTER 26
Introduction to Retrieving and Formatting Data

This chapter provides an introduction to retrieving and formatting data. It describes how to use forms to get user data and how to control the data that is displayed by a dynamic web page. It also describes how to populate a table with query results and how to use Macromedia ColdFusion MX functions to format and manipulate data. This chapter assumes that you are familiar with HTML forms.

Contents

Using forms in ColdFusion MX 609
Working with action pages 613
Working with queries and data 617
Returning results to the user 621
Dynamically populating list boxes 625
Creating dynamic check boxes and multiple-selection list boxes 627

Using forms in ColdFusion MX

ColdFusion MX lets you use a variety of types of forms. You can use plain HTML or CFML, and you can generate HTML, Macromedia Flash, or skinned XML forms. This section describes your form options and introduces a basic ColdFusion MX form.

ColdFusion forms tags

You can use HTML or CFML tags to define your form. ColdFusion includes the following CFML tags that correspond to HTML tags, but provide additional functionality:

- cfapplet
- cfform
- cfinput
- cfselect
- cftextarea

These tags support all the attributes of their HTML counterparts, plus ColdFusion attributes and features.

ColdFusion also provides the following forms tags that have no direct equivalent in HTML:

- cfcalendar Lets users select dates from a Flash month-by-month calendar.
- cfgrid Displays and lets users enter data in a row and column grid format; can get data directly from a query.
- cfslider Lets users input data by moving a sliding marker.
- cftree Displays data in a hierarchical tree format with graphical indicators; can get data directly from a query.

ColdFusion Form tag features

ColdFusion forms tags provide the following features:

Built-in validation support You can validate data in the client browser or on the server. You can specify that a field is required, contains a specific type of data, has a maximum length, or is in a range of values. You can also use data masking to control user input. For more information on validation, see Chapter 28, "Validating Data," on page 659.

Note: ColdFusion also provides a method of doing on-server validation of HTML form controls.

Flash format forms and elements You can display a form as Flash, which works identically on a variety of platforms and provides additional display features not available in HTML. These features include accordion-style and multiple-tab form panes and automatic element positioning. You can also display `cftree`, `cfgrid`, and `cfcalendar` form elements as Flash items in an otherwise-HTML form. For more information on Flash forms and form elements, see Chapter 29, "Creating Forms in Macromedia Flash," on page 687.

XML Skinable forms ColdFusion MX can generate XML forms and apply XSLT skins to format the forms. XML format forms let you separate the form presentation from the form logic and data field information. They give you detailed control over the appearance of the forms by applying custom skins, and let you create custom controls. For more information on XML skinnable forms, see Chapter 30, "Creating Skinnable XML Forms," on page 709.

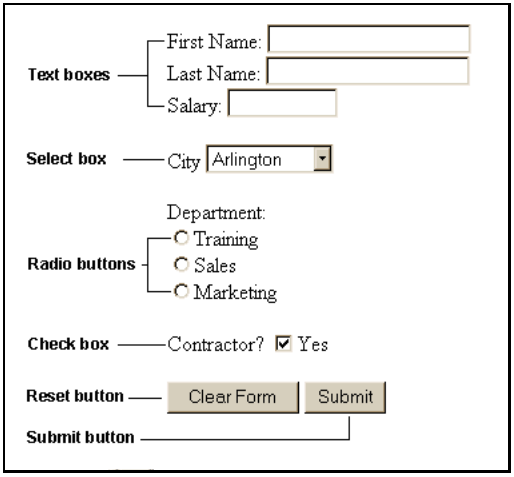
Direct support for ColdFusion variables You can easily use ColdFusion variables directly to populate your form controls. For example you can specify a query result to populate the `cfgrid` and `cftree` tags.

These features make CFML forms tags powerful and flexible, and let you easily develop fully featured, pleasing forms.

This chapter uses CFML tags, but does not describe or use most of their special features. Chapter 27, "Building Dynamic Forms with cfform Tags," on page 631 describes how to use many of the tags that are specific to ColdFusion, such as `cftree` and `cfgrid`. The other chapters listed above cover other CFML form tag features in detail.

Creating a basic form

The following simple form shows how you can create a form that lets a user enter data. This form uses basic CFML form tags. It does not use any of the advanced features of ColdFusion MX, such as validation, Flash or XML format, or special input controls. You could convert it to a purely HTML form by removing the initial "cf" prefix from the tag names, and the form would work.



The following table shows the format of form control tags:

Control	Code
Text control	`<cfinput type="Text" name="ControlName" size="Value" maxlength="Value">`

List (select) box	`<cfselect name="ControlName">`
	`  <option value="Value1">DisplayName1`
	`  <option value="Value2">DisplayName2`
	`  <option value="Value3">DisplayName3`
	`</cfselect>`

Radio buttons	`<cfinput type="Radio" name="ControlName" value="Value1">DisplayName1`
	`<cfinput type="Radio" name="ControlName" value="Value2">DisplayName2`
	`<cfinput type="Radio" name="ControlName" value="Value3">DisplayName3`

Check box	`<cfinput type="Checkbox" name="ControlName" value="Yes|No">Yes`
Reset button	`<cfinput type="Reset" name="ControlName" value="DisplayName">`
Submit button	`<cfinput type="Submit" name="ControlName" value="DisplayName">`

The following listing shows the form source in detail. To test the form and use it as input for later examples in this chapter, save this code as formpage.cfm.

```
<html>
<head>
<title>Input form</title>
</head>
<body>
<!--- Specify the action page in the form tag. The form variables will
     pass to this page when the form is submitted. --->
```

```
<cfform action="actionpage.cfm" method="post">

<!--- Text box. --->
<p>
First Name: <cfinput type="Text" name="FirstName" size="20"
  maxlength="35"><br>
Last Name: <cfinput type="Text" name="LastName" size="20" maxlength="35"><br>
Salary: <cfinput type="Text" name="Salary" size="10" maxlength="10">
</p>

<!--- List box. --->
<p>
City
<cfselect name="City">
  <option value="Arlington">Arlington
  <option value="Boston">Boston
  <option value="Cambridge">Cambridge
  <option value="Minneapolis">Minneapolis
  <option value="Seattle">Seattle
</cfselect>
</p>

<!--- Radio buttons. --->
<p>
Department:<br>
<cfinput type="radio" name="Department" value="Training">Training<br>
<cfinput type="radio" name="Department" value="Sales">Sales<br>
<input type="radio" name="Department"
  value="Marketing">Marketing<br>
</p>

<!--- Check box. --->
<p>
Contractor? <cfinput type="checkbox" name="Contractor"
  value="Yes" checked>Yes
</p>

<!--- Reset button. --->
<cfinput type="Reset" name="ResetForm" value="Clear Form">
<!--- submit button --->
<cfinput type="Submit" name="SubmitForm" value="Submit">

</cfform>
</body>
</html>
```

Forms guidelines

When using forms, keep the following guidelines in mind:

- To make the coding process easy to follow, name form controls the same as target database fields. For example, if a text control corresponds to a data source FirstName field, use FirstName as the control name.
- For ease of use, limit radio buttons to between three and five mutually exclusive options. If you need more options, consider a drop-down list.
- Use list boxes to allow the user to choose from many options or to choose multiple items from a list.
- Check boxes, radio buttons, and list boxes do not pass data to action pages unless they are selected on a form. If you try to reference these variables on the action page, you receive an error if they are not present. For information on how to determine whether a variable exists on the action page, see "Testing for a variable's existence" on page 616.
- You can dynamically populate drop-down lists using query data. For more information, see "Dynamically populating list boxes" on page 625.

Working with action pages

When the user submits a form, ColdFusion runs the action page specified by the `cfform` or `form` tag `action` attribute. A ColdFusion action page is like any other application page, except that you can use the form variables that are passed to it from an associated form. The following sections describe how to create effective action pages.

Processing form variables on action pages

The action page gets a form variable for every form control that contains a value when the form is submitted.

Note: If multiple controls have the same name, one form variable is passed to the action page with a comma-delimited list of values.

A form variable's name is the name that you assigned to the form control on the form page. Refer to the form variable by name within tags, functions, and other expressions on an action page.

On the action page, the form variables are in the Form scope, so you should prefix them with "Form." to explicitly tell ColdFusion that you are referring to a form variable. For example, the following code references the LastName form variable for output on an action page:

```
<cfoutput>
  #Form.LastName#
</cfoutput>
```

The Form scope also contains a list variable called `Form.fieldnames`. It contains a list of all form variables submitted to the action page. If no form variables are passed to the action page, ColdFusion does not create the `Form.fieldnames` list.

Using form data to generate SQL statements

As described in previous chapters, you can retrieve a record for every employee in a database table by composing a query like the following:

```
<cfquery name="GetEmployees" datasource="cfdocexamples">
  SELECT  FirstName, LastName, Contract
  FROM   Employee
</cfquery>
```

When you want to return information about employees that matches user search criteria, you use the SQL WHERE clause with a SQL SELECT statement. When the WHERE clause is processed, it filters the query data based on the results of the comparison.

For example, to return employee data for only employees with the last name of Smith, you build a query that looks like the following:

```
<cfquery name="GetEmployees" datasource="cfdocexamples">
  SELECT FirstName, LastName, Contract
  FROM    Employee
  WHERE  LastName = 'Smith'
</cfquery>
```

However, instead of putting the LastName directly in the SQL WHERE clause, you can use the text that the user entered in the form for comparison:

```
<cfquery name="GetEmployees" datasource="cfdocexamples">
  SELECT FirstName, LastName, Salary
  FROM Employee
  WHERE LastName=<cfqueryparam value="#Form.LastName#"
  CFSQLType="CF_SQL_VARCHAR">
</cfquery>
```

For security, this example encapsulates the form variable within the `cfqueryparam` tag to ensure that the user passed a valid string value for the LastName. For more information on using the `cfqueryparam` tag with queries and on dynamic SQL, see Chapter 20, “Accessing and Retrieving Data,” on page 465.

Creating action pages

Use the following procedure to create an action page for the formpage.cfm page that you created in the previous example.

To create an action page for the form:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>Retrieving Employee Data Based on Criteria from Form</title>
</head>

<body>
<cfquery name="GetEmployees" datasource="cfdocexamples">
  SELECT FirstName, LastName, Salary
  FROM Employee
```

```
   WHERE LastName=<cfqueryparam value="#Form.LastName#"
   CFSQLType="CF_SQL_VARCHAR">
</cfquery>
<h4>Employee Data Based on Criteria from Form</h4>
<cfoutput query="GetEmployees">
#FirstName#
#LastName#
#Salary#<br>
</cfoutput>
<br>
<cfoutput>Contractor: #Form.Contractor#</cfoutput>
</body>
</html>
```

2. Save the page as actionpage.cfm in the myapps directory.
3. View the formpage.cfm page in your browser.
4. Enter data, for example, Smith, in the Last Name box and submit the form.

   The browser displays a line with the first and last name and salary for each entry in the database that match the name you typed, followed by a line with the text "Contractor: Yes".
5. Click Back in your browser to redisplay the form.
6. Remove the check mark from the check box and submit the form again.

   This time an error occurs because the check box does not pass a variable to the action page. For information on modifying the actionpage.cfm page to fix the error, see "Testing for a variable's existence" on page 616.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
`<cfquery name="GetEmployees" datasource="cfdocexamples">`	Queries the data source cfdocexamples and names the query GetEmployees.
`SELECT FirstName, LastName, Salary FROM Employee WHERE LastName=<cfqueryparam value="#Form.LastName#" CFSQLType="CF_SQL_VARCHAR">`	Retrieves the FirstName, LastName, and Salary fields from the Employee table, but only if the value of the LastName field matches what the user entered in the LastName text box in the form on formpage.cfm.
`<cfoutput query="GetEmployees">`	Displays results of the GetEmployees query.
`#FirstName# #LastName# #Salary# `	Displays the value of the FirstName, LastName, and Salary fields for a record, starting with the first record, then goes to the next line. Keeps displaying the records that match the criteria you specified in the SELECT statement, followed by a line break, until you run out of records.

Code	Description
`</cfoutput>`	Closes the `cfoutput` block.
` ` `<cfoutput>Contractor: #Form.Contractor#` `</cfoutput>`	Displays a blank line followed by the text "Contractor": and the value of the form Contractor check box. A more complete example would test to ensure the existence of the variable and would use the variable in the query.

Testing for a variable's existence

Before relying on a variable's existence in an application page, you can test to see if it exists using the ColdFusion `IsDefined` function. A *function* is a named procedure that takes input and operates on it. For example, the `IsDefined` function determines whether a variable exists. CFML provides a large number of functions, which are documented in *CFML Reference*.

The following code prevents the error in the previous example by checking to see whether the Contractor Form variable exists before using it:

```
<cfif IsDefined("Form.Contractor")>
  <cfoutput>Contractor: #Form.Contractor#</cfoutput>
</cfif>
```

The argument passed to the `IsDefined` function must always be enclosed in double-quotation marks. For more information on the `IsDefined` function, see *CFML Reference*.

If you attempt to evaluate a variable that you did not define, ColdFusion cannot process the page and displays an error message. To help diagnose such problems, turn on debugging in the ColdFusion MX Administrator. The Administrator debugging information shows which variables are being passed to your application pages.

Requiring users to enter values in form fields

One of the limitations of HTML forms is the inability to define input fields as required. Because this is a particularly important requirement for database applications, ColdFusion lets you require users to enter data in fields. To specify a field as required, you can do either of the following:

- Use the `required` attribute of the `cfinput`, `cfselect`, `cftextarea`, and `cftree` tags.
- Use a hidden field that has a `name` attribute composed of the field name and the suffix "_required". You can use this technique with CFML and HTML form tags.

For example, to require that the user enter a value in the FirstName field of a `cfinput` tag, use the following syntax:

```
<cfinput type="Text" name="FirstName" size="20" maxlength="35" required="Yes">
```

To require that the user enter a value in the FirstName field of an HTML `input` tag, use the following syntax:

```
<input type="Text" name="FirstName" size="20" maxlength="35">
<input type="hidden" name="FirstName_required">
```

In either of these examples, if the user leaves the FirstName field empty, ColdFusion rejects the form submittal and returns a message informing the user that the field is required. You can customize the contents of this error message.

If you use a `required` attribute, you customize the message by using the `message` attribute, as follows:

```
<cfinput type="Text" name="FirstName" size="20" maxlength="35" required="Yes"
  message="You must enter your first name.">
```

If you use a hidden field tag, you customize the message using the `value` attribute of the hidden field, as follows:

```
<input type="hidden" name="FirstName_required"
  value="You must enter your first name.">
```

Form variable notes and considerations

When using form variables in an action page, keep the following guidelines in mind:

- A form variable is available on the action page and pages that it includes.
- Prefix form variables with "Form." when referencing them on the action page.
- Surround variable values with number signs (#) for output.
- Variables for check boxes, radio buttons, and list boxes with `size` attributes greater than 1 only get passed to the action page if you select an option. Text boxes, passwords, and textarea fields pass an empty string if you do not enter text.
- An error occurs if the action page tries to use a variable that was not passed.
- If multiple controls have the same name, one form variable is passed to the action page with a comma-delimited list of values.
- You can validate form variable values on the client or the server.

Working with queries and data

The ability to generate and display query data is one of the most important and flexible features of ColdFusion. The following sections describe more about using queries and displaying their results. Some of these tools are effective for presenting any data, not just query results.

Using HTML tables to display query results

You can use HTML tables to specify how the results of a query appear on a page. To do so, you put the `cfoutput` tag *inside* the table tags. You can also use the HTML `th` tag to put column labels in a header row. To create a row in the table for each row in the query results, put the `tr` block inside the `cfoutput` tag.

In addition, you can use CFML functions to format individual pieces of data, such as dates and numeric values.

To put the query results in a table:

1. Open the ColdFusion actionpage.cfm page in your editor.
2. Modify the page so that it appears as follows:

```
<html>
<head>
<title>Retrieving Employee Data Based on Criteria from Form</title>
</head>

<body>
<cfquery name="GetEmployees" datasource="cfdocexamples">
  SELECT FirstName, LastName, Salary
  FROM Employee
  WHERE LastName=<cfqueryparam value="#Form.LastName#"
  CFSQLType="CF_SQL_VARCHAR">
</cfquery>
<h4>Employee Data Based on Criteria from Form</h4>
<table>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Salary</th>
</tr>
<cfoutput query="GetEmployees">
<tr>
<td>#FirstName#</td>
<td>#LastName#</td>
<td>#Salary#</td>
</tr>
</cfoutput>
</table>
<br>
<cfif IsDefined("Form.Contractor")>
  <cfoutput>Contractor: #Form.Contractor#</cfoutput>
</cfif>
</body>
</html>
```

3. Save the page as actionpage.cfm in the myapps directory.
4. View the formpage.cfm page in your browser.
5. Enter Smith in the Last Name text box and submit the form.

   The records that match the criteria specified in the form appear in a table.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
`<table>`	Puts data into a table.

`<tr>`
`  <th>First Name</th>`
`  <th>Last Name</th>`
`  <th>Salary</th>`
`</tr>`
	In the first row of the table, includes three columns, with the headings: First Name, Last Name, and Salary.

`<cfoutput query="GetEmployees">`	Tells ColdFusion to display the results of the GetEmployees query.

`<tr>`
`  <td>#FirstName#</td>`
`  <td>#LastName#</td>`
`  <td>#Salary#</td>`
`</tr>`
	For each record in the query, creates a new row in the table, with three columns that display the values of the FirstName, LastName, and Salary fields of the record.

`</cfoutput>`	Ends the output region.
`</table>`	Ends the table.

Formatting individual data items

You can format individual data items. For example, you can format the salary data as monetary values. To format the salary data using the dollar format, you use the CFML function `DollarFormat(number)`.

To change the format of the Salary:

1. Open the file actionpage.cfm in your editor.
2. Change the following line:

   ```
   <td>#Salary#</td>
   ```

   to

   ```
   <td>#DollarFormat(Salary)#</td>
   ```

3. Save the page.

Building flexible search interfaces

One option with forms is to build a search based on the form data. For example, you could use form data as part of the WHERE clause to construct a database query.

To give users the option to enter multiple search criteria in a form, you can wrap conditional logic around a SQL AND clause as part of the WHERE clause. The following action page allows users to search for employees by department, last name, or both.

Note: ColdFusion MX provides the Verity search utility that you can also use to perform a search. For more information, see Chapter 24, "Building a Search Interface," on page 547.

To build a more flexible search interface:

1. Open the ColdFusion actionpage.cfm page in your editor.
2. Modify the page so that it appears as follows:

```
<html>
<head>
<title>Retrieving Employee Data Based on Criteria from Form</title>
</head>
<body>
<cfquery name="GetEmployees" datasource="cfdocexamples">
  SELECT Departmt.Dept_Name,
    Employee.FirstName,
    Employee.LastName,
    Employee.StartDate,
    Employee.Salary
  FROM Departmt, Employee
  WHERE Departmt.Dept_ID = Employee.Dept_ID
  <cfif IsDefined("Form.Department")>
  AND Departmt.Dept_Name=<cfqueryparam value="#Form.Department#"
      CFSQLType="CF_SQL_VARCHAR">
  </cfif>
  <cfif Form.LastName IS NOT "">
    AND Employee.LastName=<cfqueryparam value="#Form.LastName#"
      CFSQLType="CF_SQL_VARCHAR">
  </cfif>
</cfquery>

<h4>Employee Data Based on Criteria from Form</h4>
<table>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Salary</th>
</tr>
<cfoutput query="GetEmployees">
<tr>
<td>#FirstName#</td>
<td>#LastName#</td>
<td>#Salary#</td>
</tr>
</cfoutput>
</table>
</body>
</html>
```

3. Save the file.
4. View the formpage.cfm page in your browser.
5. Select a department, optionally enter a last name, and submit the form.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description

`SELECT  Departmt.Dept_Name,`
`  Employee.FirstName,`
`  Employee.LastName,`
`  Employee.StartDate,`
`  Employee.Salary`
`  FROM Departmt, Employee`
`  WHERE Departmt.Dept_ID =`
`  Employee.Dept_ID`
	Retrieves the fields listed from the Departmt and Employee tables, joining the tables based on the Dept_ID field in each table.

`<cfif IsDefined("FORM.Department")>`
`AND Departmt.Dept_Name =`
`<cfqueryparam`
`  value="#Form.Department#"`
`  CFSQLType="CF_SQL_VARCHAR">`
`</cfif>`
	If the user specified a department on the form, only retrieves records where the department name is the same as the one that the user specified. You must use number signs (#) in the SQL AND statement to identify Form.Department as a ColdFusion variable, but not in the `IsDefined` function.

`<cfif Form.LastName IS NOT "">`
`AND Employee.LastName = <cfqueryparam`
`  value="#Form.LastName#"`
`  CFSQLType="CF_SQL_VARCHAR">`
`</cfif>`
	If the user specified a last name in the form, only retrieves the records in which the last name is the same as the one that the user entered in the form.

Returning results to the user

When you return your results to the user, you must make sure that your pages respond to the user's needs and are appropriate for the type and amount of information. In particular, you must consider the following situations:

- When there are no query results
- When you return partial results

Handling no query results

Your code must accommodate the cases in which a query does not return any records. To determine whether a search has retrieved records, use the `RecordCount` query variable. You can use the variable in a conditional logic expression that determines how to display search results appropriately to users.

Note: For more information on query variables, including `RecordCount`, see Chapter 20, "Accessing and Retrieving Data," on page 465.

For example, to inform the user when no records were found by the GetEmployees query, insert the following code before displaying the data:

```
<cfif GetEmployees.RecordCount IS "0">
  No records match your search criteria. <BR>
<cfelse>
```

You must do the following:

- Prefix `RecordCount` with the query name.
- Add a procedure after the `cfif` tag that displays a message to the user.
- Add a procedure after the `cfelse` tag to format the returned data.
- Follow the second procedure with a `</cfif>` tag end to indicate the end of the conditional code.

To return search results to users:

1. Edit the actionpage.cfm page.
2. Change the page so that it appears as follows:

```
<html>
<head>
<title>Retrieving Employee Data Based on Criteria from Form</title>
</head>

<body>
<cfquery name="GetEmployees" datasource="cfdocexamples">
  SELECT Departmt.Dept_Name,
    Employee.FirstName,
    Employee.LastName,
    Employee.StartDate,
    Employee.Salary
  FROM Departmt, Employee
  WHERE Departmt.Dept_ID = Employee.Dept_ID
  <cfif isdefined("Form.Department")>
    AND Departmt.Dept_Name = <cfqueryparam value="#Form.Department#"
      CFSQLType="CF_SQL_VARCHAR">
  </cfif>
  <cfif Form.LastName is not "">
    AND Employee.LastName = <cfqueryparam value="#Form.LastName#"
      CFSQLType="CF_SQL_VARCHAR">
  </cfif>
</cfquery>

<cfif GetEmployees.recordcount is "0">
No records match your search criteria. <br>
Please go back to the form and try again.
<cfelse>
<h4>Employee Data Based on Criteria from Form</h4>
<table>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Salary</th>
</tr>
<cfoutput query="GetEmployees">
<tr>
<td>#FirstName#</td>
<td>#LastName#</td>
<td>#Salary#</td>
</tr>
```

```
</cfoutput>
</cfif>
</table>
</body>
</html>
```

3. Save the file.
4. Return to the form, enter search criteria, and submit the form.
5. If no records match the criteria you specified, the message appears.

Returning results incrementally

You can use the `cfflush` tag to incrementally display long-running requests to the browser before a ColdFusion page is fully processed. This tag lets you give the user quick feedback when it takes a long time to complete processing a request. For example, when a request takes time to return results, you can use the `cfflush` tag to display the message, "Processing your request -- please wait." You can also use it to incrementally display a long list as it gets retrieved.

The first time you use the `cfflush` tag on a page, it sends to the browser all of the HTML headers and any other available HTML. Subsequent `cfflush` tags on the page send only the output that ColdFusion generated after the previous flush.

You can specify an `interval` attribute to tell ColdFusion to flush the output each time that at least the specified number of bytes become available. (The count does not include HTML headers and any data that is already available when you make this call.) You can use the `cfflush` tag in a `cfloop` tag to incrementally flush data as it becomes available. This format is particularly useful when a query responds slowly with large amounts of data.

When you flush data, make sure that a sufficient amount of information is available, because some browsers might not respond if you flush only a very small amount. Similarly, if you use an `interval` attribute, set it for a reasonable size, such as a few hundred bytes or more, but not many thousands of bytes.

Limitations of the cfflush tag Because the `cfflush` tag sends data to the browser when it executes, it has several limitations, including the following:

- Using any of the following tags or functions on a page anywhere after the `cfflush` tag can cause errors or unexpected results: `cfcontent`, `cfcookie`, `cfform`, `cfheader`, `cfhtmlhead`, `cflocation`, and `SetLocale`. (These tags and functions normally modify the HTML header, but cannot do so after a `cfflush` tag, because the `cfflush` tag sends the header.)
- Using the cfset tag to set a cookie anywhere on a page that has a `cfflush` tag does not set the cookie in the browser.
- Using the `cfflush` tag within the body of several tags, including `cfsavecontent`, `cfquery`, and custom tags, can cause errors.
- If you save Client variables as cookies, any client variables that you set after a `cfflush` tag are not saved in the browser.
- You can catch `cfflush` errors, except Cookie errors, with a `cfcatch type="template"` tag. Catch cookie errors with a `cfcatch type="Any"` tag.

Example: using the cfloop tag and Rand function

The following example uses the `cfloop` tag and the `Rand` random number generating function to artificially delay the generation of data for display. It simulates a situation in which it takes time to retrieve the first data and additional information becomes available slowly.

```
<html>
<head>
  <title>Your Magic numbers</title>
</head>

<body>
<h1>Your Magic numbers</h1>
<P>It will take us a little while to calculate your ten magic numbers.
It takes a lot of work to find numbers that truly fit your personality.
So relax for a minute or so while we do the hard work for you.</P>
<h2>We are sure you will agree it was worth the short wait!</h2>
<cfflush>

<cfflush interval=10>
<!--- Delay Loop to make is seem harder. --->
<cfloop index="randomindex" from="1" to="200000" step="1">
  <cfset random=rand()>
</cfloop>

<!--- Now slowly output 10 random numbers. --->
<cfloop index="Myindex" from="1" to="10" step="1">
  <cfloop index="randomindex" from="1" to="100000" step="1">
    <cfset random=rand()>
  </cfloop>
  <cfoutput>
    Magic number #Myindex# is:  #RandRange(
  100000, 999999)#<br><br>
  </cfoutput>
</cfloop>
</body>
</html>
```

Reviewing the code

The following table describes the code and its function:

Code	Description
`<h2>We are sure you will agree it was worth the short wait!</h2>` `<cfflush>`	Sends the HTML header and all HTML output to the `cfflush` tag to the user. This displays the explanatory paragraph and H2 tag contents.
`<cfflush interval=10>`	Flushes additional data to the user whenever at least 10 bytes are available.

Code	Description
<cfloop index="randomindex" from="1" to="200000" step="1"> <cfset random=Rand()> </cfloop>	Inserts an artificial delay by using the Rand function to calculate many random numbers.
<cfloop index="Myindex" from="1" to="10" step="1"> <cfloop index="randomindex" from="1" to="100000" step="1"> <cfset random=rand()> </cfloop> <cfoutput> Magic number #Myindex# is: #RandRange (100000,999999)# </cfoutput> </cfloop>	Generates and displays 10 random numbers. This code uses two loops. The outer loop repeats ten times, once for each number to display. The inner loop uses the Rand function to create another delay by generating more (unused) random numbers. It then calls the RandRange function to generate a six-digit random number for display.

Dynamically populating list boxes

The code in "Creating a basic form" on page 611 hard-coded the form's list box options. Instead of manually entering the information on a form, you can dynamically populate a list box with database fields. When you write code this way, the form page automatically reflects the changes that you make to the database.

You use two tags to dynamically populate a list box:

- Use the cfquery tag to retrieve the column data from a database table.
- Use the cfoutput tag with the query attribute within the select tag to dynamically populate the options of this form control.

To dynamically populate a list box:

1. Open the formpage.cfm page.
2. Modify the file so that it appears as follows:

```
<html>
<head>
<title>Input form</title>
</head>
<body>
<cfquery name="GetDepartments" datasource="cfdocexamples">
  SELECT DISTINCT Location
  FROM Departmt
</cfquery>

<!--- Define the action page in the form tag.
  The form variables pass to this page
  when the form is submitted --->

<cfform action="actionpage.cfm" method="post">

<!--- Text box. --->
<p>
```

```
First Name: <cfinput type="Text" name="FirstName" size="20"
  maxlength="35"><br>
Last Name: <cfinput type="Text" name="LastName" size="20"
  maxlength="35"><br>
Salary: <cfinput type="Text" name="Salary" size="10" maxlength="10">
</p>

<!--- List box. --->
City
<cfset optsize=getDepartments.recordcount + 1>
<cfselect name="City" query="GetDepartments" value="Location"
  size="#optsize#">
  <option value="">Select All
</cfselect>

<!--- Radio buttons. --->
<p>
Department:<br>
<cfinput type="radio" name="Department" value="Training">Training<br>
<cfinput type="radio" name="Department" value="Sales">Sales<br>
<cfinput type="radio" name="Department" value="Marketing">Marketing<br>
<cfinput type="radio" name="Department" value="HR">HR<br>
</p>

<!--- Check box. --->
<p>
Contractor? <cfinput type="checkbox" name="Contractor" value="Yes"
  checked>Yes
</p>

<!--- Reset button. --->
<cfinput type="reset" name="ResetForm" value="Clear Form">

<!--- Submit button. --->
<cfinput type="submit" name="SubmitForm" value="Submit">
</cfform>
</body>
</html>
```

3. Save the page as formpage.cfm.
4. View the formpage.cfm page in a browser.

   The changes that you just made appear in the form.

   Remember that you need an action page to submit values.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description

`<cfquery name="GetDepartments" datasource="cfdocexamples">`
`SELECT DISTINCT Location`
`FROM Departmt`
`</cfquery>`
	Gets the locations of all departments in the Departmt table. The DISTINCT clause eliminates duplicate location names from the returned query results.

`<cfset optsize=getDepartments.recordcount + 1>`	Sets the optsize variable to the number of entries to add dynamically to the selection list, plus one for the manually coded Select All option.

`<cfselect name="City" query="GetDepartments" value="Location" size="#optsize#">`
`<option value="">Select All`
`</cfselect>`
	Populates the City selection list from the Location column of the GetDepartments query. The control has one option for each row returned by the query. Adds an option that allows users to select all locations. If the user selects this option, the form value is an empty string. The action page must check for the empty string and handle it appropriately.

Creating dynamic check boxes and multiple-selection list boxes

When an HTML or CFML form contains a list of check boxes with the same name or a multiple-selection list box (that is, a box in which users can select multiple items from the list), the user's entries are made available as a comma-delimited list with the selected values. These lists can be very useful for a wide range of input types.

Note: If the user does not select a check box or make a selection from a list box, no variable is created. The `cfinsert` and `cfupdate` tags do not work correctly if there are no values. To prevent errors, make the form fields required, use dynamic SQL, or use the `cfparam` tag to set a default value for the form field.

Check boxes

When you put a series of check boxes with the same name in a form, the variable that is created contains a comma-delimited list of values. The values can be either numeric values or alphanumeric strings. These two types of values are treated slightly differently.

Handling numeric values

Suppose you want a user to select one or more departments using check boxes. You then query the database to retrieve detailed information on the selected department(s). The code for a simple set of check boxes that lets the user select departments looks like the following:

```
<cfinput type="checkbox"
  name="SelectedDepts"
  value="1">
  Training<br>

<cfinput type="checkbox"
```

```
  name="SelectedDepts"
  value="2">
  Marketing<br>

<cfinput type="checkbox"
  name="SelectedDepts"
  value="3">
  HR<br>

<cfinput type="checkbox"
  name="SelectedDepts"
  value="4">
  Sales<br>
</html>
```

The user sees the name of the department, but the `value` attribute of each check box is a number that corresponds to the underlying database primary key for the department's record.

If the user checks the Marketing and Sales items, the value of the SelectedDepts form field is 2,4 and you use the SelectedDepts value in the following SQL statement:

```
SELECT *
  FROM Departmt
  WHERE Dept_ID IN ( #Form.SelectedDepts# )
```

The ColdFusion server sends the following statement to the database:

```
SELECT *
  FROM Departmt
  WHERE Dept_ID IN ( 2,4 )
```

Handling string values

To search for a database field that contains string values (instead of numeric), you must modify the `checkbox` and `cfquery` syntax to make sure that the string values are sent to the data source in single-quotation marks (').

The first example searched for department information based on a numeric primary key field called Dept_ID. Suppose, instead, that the primary key is a database field called Dept_Name that contains string values. In that case, your code for check boxes should look like the following:

```
<cfinput type="checkbox"
  name="SelectedDepts"
  value="Training">
  Training<br>

<cfinput type="checkbox"
  name="SelectedDepts"
  value="Marketing">
  Marketing<br>

<cfinput type="checkbox"
  name="SelectedDepts"
  value="HR">
  HR<br>
```

```
<cfinput type="checkbox"
  name="SelectedDepts"
  value="Sales">
  Sales<br>
```

If the user checked Marketing and Sales, the value of the SelectedDepts form field would be the list Marketing,Sales and you use the following SQL statement:

```
SELECT *
  FROM Departmt
  WHERE Dept_Name IN
  (#ListQualify(Form.SelectedDepts,"'")#)
```

In SQL, all strings must be surrounded in single-quotation marks. The `ListQualify` function returns a list with the specified qualifying character (here, a single-quotation mark) around each item in the list.

If you select the second and fourth check boxes in the form, the following statement gets sent to the database:

```
SELECT *
  FROM Departmt
  WHERE Dept_Name IN ('Marketing','Sales')
```

Multiple selection lists

A multiple-selection list box is defined by a `select` or `cfselect` tag with a `multiple` or `multipe="yes"` attribute and a `size` attribute value greater than 1. ColdFusion treats the result when a user selects multiple choices from a multiple-selection list box like the results of selecting multiple check boxes. The data made available to your page from any multiple-selection list box is a comma-delimited list of the entries selected by the user; for example, a list box could contain the four entries: Training, Marketing, HR, and Sales. If the user selects Marketing and Sales, the form field variable value is Marketing,Sales.

You can use multiple-selection lists to search a database in the same way that you use check boxes. The following sections describe how you can use different types of multiple-selection data values.

Handling numeric values

Suppose you want the user to select departments from a multiple-selection list box. The query retrieves detailed information on the selected department(s), as follows:

```
Select one or more companies to get more information on:
<cfselect name="SelectDepts" multiple>
  <option value="1">Training
  <option value="2">Marketing
  <option value="3">HR
  <option value="4">Sales
</cfselect>
```

If the user selects the Marketing and Sales items, the value of the SelectDepts form field is 2,4. If this parameter is used in the following SQL statement:

```
SELECT *
  FROM Departmt
  WHERE Dept_ID IN (#form.SelectDepts#)
```

The following statement is sent to the database:

```
SELECT *
  FROM Departmt
  WHERE Dept_ID IN (2,4)
```

Handling string values

Suppose you want the user to select departments from a multiple-selection list box. The database search field is a string field. The query retrieves detailed information on the selected department(s), as follows:

```
<cfselect name="SelectDepts" multiple>
  <option value="Training">Training
  <option value="Marketing">Marketing
  <option value="HR">HR
  <option value="Sales">Sales
</cfselect>
```

If the user selects the Marketing and Sales items, the SelectDepts form field value is Marketing,Sales.

Just as you did when using check boxes to search database fields containing string values, use the ColdFusion `ListQualify` function with multiple-selection list boxes:

```
SELECT *
  FROM Departmt
  WHERE Dept_Name IN (#ListQualify(Form.SelectDepts,"'")#)
```

The following statement is sent to the database:

```
SELECT *
  FROM Departmt
  WHERE Dept_Name IN ('Marketing','Sales')
```

CHAPTER 27
Building Dynamic Forms with cfform Tags

This chapter describes some of the ways that you can use the `cfform` tag to create rich, dynamic forms with sophisticated graphical controls, including several Java applet or Macromedia Flash controls. You can use these controls without writing a line of Java or Flash code. This chapter describes the `cfform` tag and its child tags, and covers using several of the form controls, independent of the output format.

Contents

Creating custom forms with the cfform tag 631

Building tree controls with the cftree tag 634

Building drop-down list boxes 642

Building slider bar controls 643

Creating data grids with the cfgrid tag 644

Embedding Java applets 655

Creating custom forms with the cfform tag

The `cfform` tag and its CFML subtags let you create dynamic forms in three formats:

HTML Generates standard HTML tags wherever possible, and uses applets or Flash for more complex controls, such as grids, trees, and calendars. HTML format lets you present a familiar appearance, but does not let you easily separate data and presentation, or provide some of the more complex structures, such as Flash tabbed navigators or accordions, or customized XML controls.

Flash Presents a modern, visually pleasing appearance. Flash format supports several controls, such as tabbed navigators and accordions, that are not available in HTML format. Flash forms are also browser-independent. In Flash format, Flash Player works in all commonly used browsers on Windows and Macintosh systems, and in Netscape and Mozilla on Linux.

XML Lets you specify an Extensible Stylesheet Language Transformation (XSLT) skin that converts the XML into styled HTML output. ColdFusion MX provides several skins that you can use, and you can write your own custom skins and support custom controls.

The cfform tag and its subtags also provide you with several methods for validating input data. For example, you can perform the validation on the browser or on the server. You can check the data type, or you can mask data input.

Individual cfform tags have additional dynamic features. Several of the tags do not have HTML counterparts, and others directly support dynamically populating the control from data sources. Also, the cfform tag preservedata attribute retains user input in a form after the user submits the form, so the data reappears if the form gets redisplayed.

This chapter describes features of the cfform tag and focuses on using several of the cfform child tags that do not have HTML counterparts. The following chapters describe other features of ColdFusion forms that you create using the cfform tag:

- Chapter 28, "Validating Data," on page 659
- Chapter 29, "Creating Forms in Macromedia Flash," on page 687
- Chapter 30, "Creating Skinnable XML Forms," on page 709

The cfform controls

The following table describes the ColdFusion controls that you use in forms created using the cfform tag. You can use these tags only inside a cfform tag. Unless otherwise stated, these controls are supported in HTML, Flash, and XML skinable forms.

Control	Description	For more information
cfapplet	Embeds a custom Java applet in the form. Not supported in Flash format forms.	"Embedding Java applets" on page 655.
cfcalendar	Displays an interactive Macromedia Flash calendar that can be included in an HTML or Flash format form. ignored in XML skinable forms. The calendar lets a user select a date for submission as a form variable.	The cfcalendar tag in *CFML Reference*
cfformgroup	Creates a container control for organizing and formatting multiple form controls. Used in the cfform tag body of Macromedia Flash and XML skinable forms. Ignored in HTML forms.	Chapter 29, "Creating Forms in Macromedia Flash", Chapter 30, "Creating Skinnable XML Forms"
cfformitem	Inserts a horizontal line, a vertical line, or formatted or unformatted text in a Flash form. Used in the cfform or cfformgroup tag body for Flash and XML forms. Ignored in HTML forms.	Chapter 29, "Creating Forms in Macromedia Flash", Chapter 30, "Creating Skinnable XML Forms"
cfgrid	Creates a Java applet or Flash data grid that you can populate from a query or by defining the contents of individual cells. You can also use grids to insert, update, and delete records from a data source.	"Creating data grids with the cfgrid tag" on page 644
cfinput	Equivalent to the HTML input tag, with the addition of input validation.	"Creating a basic form" on page 611

Control	Description	For more information
cfselect	Displays a selection box. Equivalent to the HTML select tag, with the addition of input validation.	"Building drop-down list boxes" on page 642
cfslider	Creates a Java applet-based control that lets users enter data by moving a slider. Not supported in Flash format forms.	"Building slider bar controls" on page 643
cftextarea	Displays a text input area. Equivalent to the HTML textarea tag, with the addition of input validation.	The cftextarea tag in *CFML Reference*
cftree	Creates a Java applet or Flash hierarchical tree-format control that can include graphical images for the different elements. Can also generate a ColdFusion structure that represents the tree data and attributes.	"Building tree controls with the cftree tag" on page 634

Preserving input data with the preservedata attribute

The cfform preservedata attribute tells ColdFusion to continue displaying the user data in a form after the user submits the form. Data is preserved in the cfinput, cfslider, cftextinput, and cftree controls and in cfselect controls populated by queries. If you specify a default value for a control, and a user overrides that default in the form, the user input is preserved.

You can retain data on the form when the same page contains the form and the form's action code; that is, the form submits to itself. You can also retain the data if the action page has a copy of the form, and the control names are the same in the forms on both pages. (The action page form does not need to be identical to the initial form. It can have more or fewer elements than the initial page form; only the form elements with identical names on both pages keep their data.)

Note: The preservedata setting on the action page controls the preservation of the data.

For example, if you save this form as preserve.cfm, it continues to display any text that you enter after you submit it, as follows:

```
<cfform action="preserve.cfm" preservedata="Yes">
  <p>Please enter your name:
  <cfinput type="Text" name="UserName" required="Yes"><p>
  <input type="Submit" name=""> <input type="RESET">
</cfform>
```

Usage notes for the preservedata attribute

When you use the preservedata attribute, follow these guidelines:

- In the cftree tag, the preservedata attribute causes the tree to expand to the previously selected element. For this to work correctly, you must also set the completePath attribute to True.
- The preservedata attribute has no effect on a cfgrid tag. If you populate the control from a query, you must update the data source with the new data (typically by using a cfgridupdate tag) before redisplaying the grid. The grid then displays the updated database information.

Browser considerations

The applet-based versions of the `cfgrid`, `cfslider`, and `cftree` forms use JavaScript and Java to display their content. To allow them to display consistently across a variety of browsers, these applets use the Java plug-in. As a result, they are independent of the level of Java support provided by the browser.

ColdFusion downloads and installs the browser plug-in if necessary. Some browsers display a single permission dialog box asking you to confirm the plug-in installation. Other browsers, particularly older versions of Netscape, require you to navigate some simple option windows.

Because the controls use JavaScript to return data to ColdFusion, if you disable JavaScript in your browser, it cannot properly run forms that contain these controls. In that case, the controls still display, but data return and validation does not work and you can receive a JavaScript error.

Because Java is handled by the plug-in and not directly by the browser, disabling Java execution in the browser does not affect the operation of the controls. If for some other reason, however, the browser is unable to render the controls as requested, a "not supported" message appears in place of the control.

You can use the `cfform` tag's `notsupported` attribute to specify an alternative error message.

You can avoid browser Java and JavaScript issues with the `cfgrid` and `cftree` controls by using the Flash format versions of these controls. These controls work on Windows, Mac OS X, and Linux, and do not rely on Java support. There is no Flash format version of the `cfslider` control, and there is no applet format version of the cfcalendar control.

Building tree controls with the cftree tag

The `cftree` tag lets you display hierarchical information within a form in a space-saving collapsible tree populated from data source queries. To build a tree control with the `cftree` tag, you use individual `cftreeitem` tags to populate the control..

You can create trees in three formats:

Applet Creates a Java applet that the client must download. Downloading an applet takes time; therefore, using the `cftree` tag can be slightly slower than using an HTML form element to retrieve the same information. In addition, browsers must be Java-enabled for the `cftree` tag to work properly.

Flash Generates a Flash control that you can include in an HTML or Flash format form. For more information on Flash Format see Chapter 29, "Creating Forms in Macromedia Flash," on page 687.

Object Creates a hierarchical ColdFusion structure that represents the tree data and many of the `cftree` and `cftreeitem` attributes.

The different formats support different sets of features and attributes. This section discusses general techniques that apply to all three formats, and indicates any techniques that do not apply to a specific format. It uses applet format for all examples, which use applet-specific attributes. For details on the features and attributes supported in each format, see the `cftree` entry in *CFML Reference*.

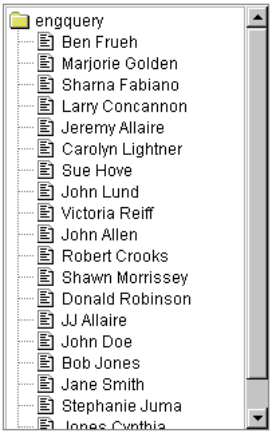
To create and populate a tree control from a query:

1. Create a ColdFusion page with the following content:

```
<cfquery name="engquery" datasource="cfdocexamples">
  SELECT FirstName + ' ' + LastName AS FullName
  FROM Employee
</cfquery>
<cfform name="form1" action="submit.cfm">
<cftree name="tree1"
  required="Yes"
  hscroll="No">
  <cftreeitem value="FullName"
    query="engquery"
    queryasroot="Yes"
    img="folder,document">
</cftree>
</cfform>
```

2. Save the page as tree1.cfm and view it in your browser.

   The following figure shows the output of this CFML page:



Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
`<cftree name="tree1"`	Creates a tree and names it tree1.
`required="Yes"`	Specifies that a user must select an item in the tree.
`hscroll="No"`	Does not allow horizontal scrolling.
`<cftreeitem value="FullName" query="engquery"`	Creates an item in the tree and puts the results of the query named engquery in it. Because this tag uses a query, it puts one item on the tree per query entry.

Code	Description
`queryasroot="Yes"`	Specifies the query name as the root level of the tree control.
`img="folder,document"`	Uses the folder and document images that ship with ColdFusion in the tree structure. When populating a `cftree` tag with data from a `cfquery` tag, you can specify images or filenames for each level of the tree as a comma-separated list.

Grouping output from a query

In a query that you display using a `cftree` control, you might want to organize your employees by department. In this case, you separate column names with commas in the `cftreeitem value` attribute.

To organize the tree based on ordered results of a query:

1. Create a ColdFusion page named tree2.cfm with the following content:

```
<!--- CFQUERY with an ORDER BY clause. --->
<cfquery name="deptquery" datasource="cfdocexamples">
  SELECT Dept_ID, FirstName + ' ' + LastName
  AS FullName
  FROM Employee
  ORDER BY Dept_ID
</cfquery>

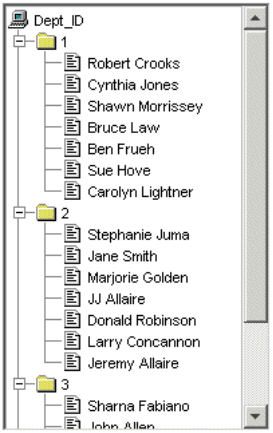
<!--- Build the tree control. --->
<cfform name="form1" action="submit.cfm">

<cftree name="tree1"
  hscroll="No"
  border="Yes"
  height="350"
  required="Yes"
>

<cftreeitem value="Dept_ID, FullName"
  query="deptquery"
  queryasroot="Dept_ID"
  img="computer,folder,document"
  imgopen="computer,folder"
  expand="yes">

</cftree>
<br>
<br><input type="Submit" value="Submit">
</cfform>
```

2. Save the page and view it in your browser. It should look as follows



Reviewing the code

The following table describes the highlighted code and its function

Code	Description
`ORDER BY Dept_ID`	Orders the query results by department.
`<cftreeitem value="Dept_ID,FullName"`	Populates the tree with the department ID, and under each department, the full name for each employee in the department.
`queryasroot="Dept_ID"`	Labels the root "Dept_ID".
`img="computer,folder,document" imgopen="computer,folder"`	Uses the ColdFusion supplied computer image for the root level, folder image for the department IDs, and document for the names, independent of whether any level is expanded (open) or collapsed. The `imgopen` attribute has only two items, because the employee names can never be open.

The `cftreeitem` comma-separated `value`, `img`, and `imgopen` attributes correspond to the tree level structure. In applet format, if you omit the `img` attribute, ColdFusion uses the folder image for all levels in the tree; if you omit the `imgopen` attribute, ColdFusion uses the folder image for all expanded levels in the tree. Flash format ignores the `img` and `imgopen` attributes and always uses folders for levels with children and documents for nodes without children.

The cftree form variables

The cftree tag lets you force a user to select an item from the tree control by setting the required attribute to Yes. With or without the required attribute, ColdFusion passes two form variables to the application page specified in the cfform action attribute:

- Form.*treename*.path Returns the complete path of the user selection, in the form: *[root]\node1\node2\node_n\value*
- Form.*treename*.node Returns the node of the user selection.

To return the root part of the path, set the completepath attribute of the cftree tag to Yes; otherwise, the path value starts with the first node. If you specify a root name for a tree item using the queryasroot tag, that value is returned as the root. If you do not specify a root name, ColdFusion returns the query name as the root. If there is no query name, ColdFusion returns the tree name as the root.

In the previous example, if the user selects the name "John Allen" in the tree, ColdFusion returns the following form variables:

```
Form.tree1.path = 3\John Allen
Form.tree1.node = John Allen
```

The deptquery root does not appear in the path, because the cftree tag does not specify completePath="Yes". You can specify the character used to delimit each element of the path form variable in the cftree delimiter attribute. The default is a backslash character (\).

Input validation

Although the cftree tag does not include a validate attribute, you can use the required attribute to force a user to select an item from the tree control. In addition, you can use the onValidate attribute to specify your own JavaScript code to perform validation.

Structuring tree controls

Tree controls built with the cftree tag can be very complex. Knowing how to specify the relationship between multiple cftreeitem entries helps you handle the most complex cftree constructs.

Creating a one-level tree control

The following example consists of a single root and a number of individual items:

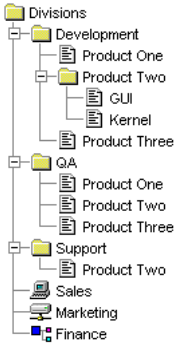
```
<cfquery name="deptquery" datasource="cfdocexamples">
  SELECT Dept_ID, FirstName + ' ' + LastName
  AS FullName
  FROM Employee
  ORDER BY Dept_ID
  </cfquery>

<cfform name="form1" action="submit.cfm">
  <cftree name="tree1">
    <cftreeitem value="FullName"
    query="deptquery"
```

```
    queryasroot="Department">
    img="folder,document">
  </cftree>
<br>
<cfinput type="submit" value="Submit">
</cfform>
```

Creating a multilevel tree control

The following figure shows an example of a multilevel tree control:



When populating a `cftree` control, you create the multilevel structure of the tree by specifying a parent for each item in the tree. The `parent` attribute of the `cftreeitem` tag allows your `cftree` tag to show relationships between elements in the tree.

In this example, every `cftreeitem` tag, except the top level Divisions, specifies a parent. For example, the `cftreeitem value="Development"` tag specifies Divisions as its parent.

The following code populates the tree directly, not from a query:

```
<cfform name="form2" action="cfform_submit.cfm">
<cftree name="tree1" hscroll="No" vscroll="No"
  border="No">
  <cftreeitem value="Divisions">
  <cftreeitem value="Development"
    parent="Divisions" img="folder">
  <cftreeitem value="Product One"
    parent="Development" img="document">
  <cftreeitem value="Product Two"
    parent="Development">
  <cftreeitem value="GUI"
    parent="Product Two" img="document">
  <cftreeitem value="Kernel"
    parent="Product Two" img="document">
  <cftreeitem value="Product Three"
    parent="Development" img="document">
  <cftreeitem value="QA"
    parent="Divisions" img="folder">
  <cftreeitem value="Product One"
    parent="QA" img="document">
  <cftreeitem value="Product Two"
```

```
    parent="QA" img="document">
  <cftreeitem value="Product Three"
    parent="QA" img="document">
  <cftreeitem value="Support"
    parent="Divisions" img="fixed">
  <cftreeitem value="Product Two"
    parent="Support" img="document">
  <cftreeitem value="Sales"
    parent="Divisions" img="computer">
  <cftreeitem value="Marketing"
    parent="Divisions" img="remote">
  <cftreeitem value="Finance"
    parent="Divisions" img="element">
</cftree>
</cfform>
```

Image names in a cftree tag

Note: This section applies to applet format trees. In Flash format, you cannot control the tree icons. Flash format uses open and closed folders and documents as the icons. In object format, the image information is preserved in fields in the object structure.

The default image displayed in a tree is a folder. However, you can use the `img` attribute of the `cftreeitem` tag to specify a different image.

When you use the `img` attribute, ColdFusion displays the specified image beside the tree items when they are not open. When you use the `imgopen` attribute, ColdFusion displays the specified image beside the tree items when they are open (expanded). You can specify a built-in ColdFusion image name, the file path to an image file, or the URL of an image of your choice, such as http://localhost/Myapp/Images/Level3.gif. You cannot use a custom image in Flash format. As a general rule, make the height of your custom images less than 20 pixels.

When populating a `cftree` control with data from a `cfquery` tag, you can use the `img` attribute of `cftreeitem` tag to specify images or filenames for each level of the tree as a comma-separated list.

The following are the ColdFusion built-in image names:

- computer
- document
- element
- folder
- floppy
- fixed
- remote

Note: In applet format, you can also control the tree appearance by using the `cftree` tag `lookAndFeel` attribute to specify a Windows, Motif, or Metal look.

Embedding URLs in a cftree tag

The `href` attribute in the `cftreeitem` tag lets you designate tree items as links. To use this feature in a `cftree` control, you define the destination of the link in the `href` attribute of the `cftreeitem` tag. The URL for the link can be a relative URL or an absolute URL, as in the following examples.

To embed links in a cftree control:

1. Create a ColdFusion page named tree3.cfm with the following contents:

```
<cfform action="submit.cfm">
<cftree name="oak"
  highlighthref="Yes"
  height="100"
  width="200"
  hspace="100"
  vspace="6"
  hscroll="No"
  vscroll="No"
  border="No">
  <cftreeitem value="Important Links">
  <cftreeitem value="Macromedia Home"
    parent="Important Links"
    img="document"
    href="http://www.macromedia.com">
  <cftreeitem value="ColdFusion Developer Center"
    parent="Important Links"
    img="document"
    href="http://www.macromedia.com/devnet/mx/coldfusion/">
</cftree>
</cfform>
```

2. Save the page and view it in your browser.

   The following figure shows the output of this code:



Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
`href="http://www.macromedia.com">`	Makes the node of the tree a link.
`href="http://www.macromedia.com/devnet/mx/coldfusion/">`	Makes the node of the tree a link. Although this example does not show it, the `href` attribute can refer to the name of a column in a query if that query populates the tree item.

Specifying the tree item in the URL

When a user clicks on a tree item to link to a URL, the `cftreeItemKey` variable, which identifies the selected value, is appended to the URL in the following form:

```
http://myserver.com?CFTREEITEMKEY=selected_item_value_attribute
```

If the value attribute includes spaces, ColdFusion replaces the spaces with plus characters (+).

Automatically passing the name of the selected tree item as part of the URL makes it easy to implement a basic "drill down" application that displays additional information based on the selection. For example, if the specified URL is another ColdFusion page, it can access the selected value as the variable `URL.CFTREEITEMKEY`.

To disable this behavior, set the `appendkey` attribute in the `cftree` tag to No.

Building drop-down list boxes

The drop-down list box that you can create in a `cfform` tag with a `cfselect` tag is similar to the HTML `select` tag. However, the `cfselect` tag gives you more control over user inputs, provides error handling, and, most importantly, lets you automatically populate the selection list from a query.

You can populate the drop-down list box from a query, or using lists of option elements created by the `option` tag. The syntax for the `option` tag with the `cfselect` tag is the same as for the HTML `option` tag.

When you populate a `cfselect` tag with data from a query, you only need to specify the name of the query that is supplying data for the `cfselect` tag and the query column name for each list element to display.

To populate a drop-down list box with query data using the cfselect tag:

1. Create a ColdFusion page with the following content:

```
<cfquery name="getNames"
  datasource="cfdocexamples">
  SELECT * FROM Employee
</cfquery>

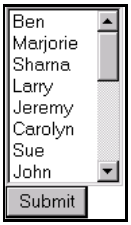
<cfform name="Form1" action="submit.cfm">

  <cfselect name="employees"
    query="getNames"
    value="Emp_ID"
    display="FirstName"
    required="Yes"
    multiple="Yes"
    size="8">
  </cfselect>

  <br><input type="Submit" value="Submit">
</cfform>
```

2. Save the file as selectbox.cfm and view it in your browser.

   The following figure shows the output of this code:



Because the tag includes the `multiple` attribute, the user can select multiple entries in the list box. Also, because the `value` tag specifies Emp_ID, the primary key for the Employee table, Employee IDs (not first names) get passed in the Form.Employee variable to the application page specified in the `cfform action` attribute.

You can use a query to create a two-level hierarchical list grouped by one of the query columns. For an example of this use, see the example for the `cfselect` entry in *CFML Reference*.

Building slider bar controls

You can use the `cfslider` control in a `cfform` tag to create a slider control and define a variety of characteristics, including label text, label font name, size, boldface, italics, and color, and slider range, positioning, and behavior. Slider bars are useful because they are highly visual and users can only enter valid values. The `cfslider` tag is not supported in Flash format forms.

To create a slider control:

1. Create a ColdFusion page with the following content:

```
<cfform name="Form1" action="submit.cfm">
  <cfslider name="myslider"
    bgcolor="cyan"
    bold="Yes"
    range="0,1000"
    scale="100"
    value="600"
    fontsize="14"
    label="Slider %value%"
    height="60"
    width="400">
</cfform>
```

2. Save the file as slider.cfm and view it in your browser.

   The following figure shows the output of this code:



To get the value of the slider in the action page, use the variable Form.*slider_name*; in this case, Form.myslider.

Creating data grids with the cfgrid tag

The `cfgrid` tag creates a `cfform` grid control that resembles a spreadsheet table and can contain data populated from a `cfquery` tag or from other sources of data. As with other `cfform` tags, the `cfgrid` tag offers a wide range of data formatting options, as well as the option of validating user selections with a JavaScript validation script.

You can also perform the following tasks with a `cfgrid` tag:

- Sort data in the grid alphanumerically.
- Update, insert, and delete data.
- Display images in the grid.

Note: Flash format grids support a subset of the features available in applet format grids. For details on features supported in each format, see the `cfgrid` tag in *CFML Reference*.

Users can sort the grid entries in ascending order by double-clicking any column header. Double-clicking again sorts the grid in descending order. In applet format, you can also add sort buttons to the grid control.

When users select grid data and submit the form, ColdFusion passes the selection information as form variables to the application page specified in the `cfform action` attribute.

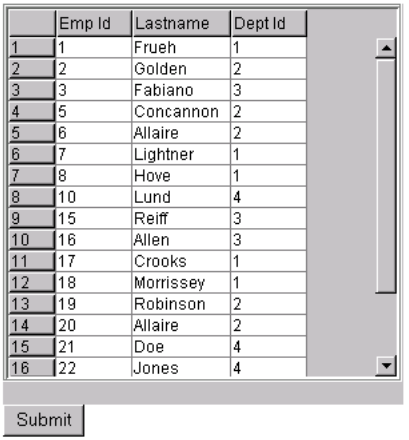
Just as the `cftree` tag uses the `cftreeitem` tag, the `cfgrid` tag uses the `cfgridcolumn` and `cfgridrow` tags. You can define a wide range of row and column formatting options, as well as a column name, data type, selection options, and so on. You use the `cfgridcolumn` tag to define individual columns in the grid or associate a query column with a grid column.

Use the `cfgridrow` tag to define a grid that does not use a query as the source for row data. If a query attribute is specified in the `cfgrid` tag, the `cfgridrow` tags are ignored.

The `cfgrid` tag provides many attributes that control grid behavior and appearance. This chapter describes only the most important of these attributes. For detailed information on these attributes, see the `cfgrid` tag in *CFML Reference*.

Working with a data grid and entering data

The following figure shows an example applet format grid created using a `cfgrid` tag:



The following table describes some navigating tips:

Action	Procedure
Sorting grid rows	Double-click the column header to sort a column in ascending order. Double-click again to sort the rows in descending order.
Rearranging columns	Click any column heading and drag the column to a new position.
Determining editable grid areas	When you click an editable cell, it is surrounded by a yellow box.
Determining noneditable grid areas	When you click a cell (or row or column) that you cannot edit, its background color changes. The default color is salmon pink.
Editing a grid cell	Double-click the cell. You must press Return when you finish entering the data.
Deleting a row	Click any cell in the row and click the Delete button. (Not available in Flash format grids.)
Inserting a row	Click the Insert button. An empty row appears at the bottom of the grid. To enter a value in each cell, double-click the cell, enter the value, and click Return. (Not available in Flash format grids.)

To populate a grid from a query:

1. Create a new ColdFusion page named grid1.cfm with the following contents:

```
<cfquery name="empdata" datasource="cfdocexamples">
  SELECT * FROM Employee
</cfquery>

<cfform name="Form1" action="submit.cfm" >
  <cfgrid name="employee_grid" query="empdata"
      selectmode="single">
    <cfgridcolumn name="Emp_ID">
    <cfgridcolumn name="LastName">
    <cfgridcolumn name="Dept_ID">
  </cfgrid>
  <br>
  <cfinput name="submitit" type="Submit" value="Submit">
</cfform>
```

Note: Use the `cfgridcolumn display="No"` attribute to hide columns that you want to include in the grid but not expose to an end user. You typically use this attribute to include columns such as the table's primary key column in the results returned by the `cfgrid` tag.

2. Save the file and view it in your browser.

   The following figure shows the output of this code:



	Emp Id	Lastname	Dept Id
1	1	Frueh	1
2	2	Golden	2
3	3	Fabiano	3
4	5	Concannon	2
5	6	Allaire	2
6	7	Lightner	1
7	8	Hove	1
8	10	Lund	4
9	15	Reiff	3
10	16	Allen	3
11	17	Crooks	1
12	18	Morrissey	1
13	19	Robinson	2
14	20	Allaire	2
15	21	Doe	4
16	22	Jones	4

Submit

Reviewing the code

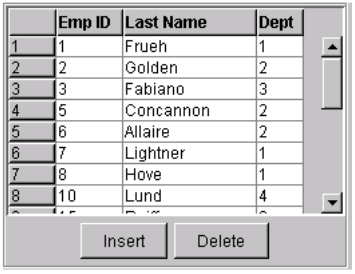
The following table describes the highlighted code and its function:

Code	Description
`<cfgrid name="employee_grid" query="empdata"`	Creates a grid named employee_grid and populate it with the results of the query empdata. If you specify a `cfgrid` tag with a `query` attribute defined and no corresponding `cfgridcolumn` attributes, the grid contains all the columns in the query.
`selectmode="single">`	Allows the user to select only one cell; does not allow editing. Other modes are row, column, and edit.
`<cfgridcolumn name="Emp_ID">`	Puts the contents of the Emp_ID column in the query results in the first column of the grid.
`<cfgridcolumn name="LastName">`	Puts the contents of the LastName column in the query results in the second column of the grid.
`<cfgridcolumn name="Dept_ID">`	Puts the contents of the Dept_ID column in the query results in the third column of the grid.

Creating an editable grid

You can build grids to allow users to edit data within them. Users can edit individual cell data, as well as insert, update, or delete rows. To enable grid editing, you specify `selectmode="edit"` in the `cfgrid` tag.

You can let users add or delete grid rows by setting the `insert` or `delete` attributes in the `cfgrid` tag to Yes. Setting the `insert` and `delete` attribute to Yes causes the `cfgrid` tag to display Insert and Delete buttons as part of the grid, as the following figure shows:



You can use a grid in two ways to make changes to your ColdFusion data sources:

- Create a page to which you pass the `cfgrid` form variables. In that page, perform `cfquery` operations to update data source records based on the form values returned by the `cfgrid` tag.
- Pass grid edits to a page that includes the `cfgridupdate` tag, which automatically extracts the form variable values and passes that data directly to the data source.

Using the `cfquery` tag gives you complete control over interactions with your data source. The `cfgridupdate` tag provides a much simpler interface for operations that do not require the same level of control.

Controlling cell contents

You can control the data that a user can enter into a `cfgrid` cell in the following ways:

- By default, a cell is not editable. Use the `cfgrid` attribute `selectmode="edit"` to edit cell contents.
- Use the `cfgridcolumn type` attribute to control sorting order, to make the fields check boxes, or to display an image.
- Use the `cfgridcolumn values` attribute to specify a drop-down list of values from which the user can choose. You can use the `valuesDisplay` attribute to provide a list of items to display that differs from the actual values that you enter in the database. You can use the `valuesDelimiter` attribute to specify the separator between values in the `values valuesDisplay` lists.
- Although the `cfgrid` tag does not have a `validate` attribute, it does have an `onValidate` attribute that lets you specify a JavaScript function to perform validation.

For more information on controlling the cell contents, see the attribute descriptions for the `cfgridcolumn` tag in *CFML Reference*.

How user edits are returned

When a user inserts or deletes a row in a grid or changes any cells in a row and submits the grid, ColdFusion creates the following arrays as Form variables:

Array name	Description
gridname.colname	Stores the new values of inserted, deleted, or updated cells. (Entries for deleted cells contain empty strings.)
gridname.Original.*colname*	Stores the original values of inserted, deleted, or updated cells.
gridname.RowStatus.Action	Stores the type of change made to the grid rows: D for delete, I for insert, or U for update.

Note: The periods in these names are **not** structure separators; they are part of the text of the array name.

ColdFusion creates a *gridname.colname* array and a *gridname.Original.colname* array for each column in the grid. For each inserted, deleted, or changed row in the grid, ColdFusion creates a row in each of these arrays.

For example, the following arrays are created if you update a `cfgrid` tag called mygrid that consists of two displayable columns (col1, col2) and one hidden column (col3):

```
Form.mygrid.col1
Form.mygrid.col2
Form.mygrid.col3
Form.mygrid.original.col1
Form.mygrid.original.col2
Form.mygrid.original.col3
Form.mygrid.RowStatus.Action
```

The value of the array index increments for each row that is added, deleted, or changed, and does not indicate a grid row number. All rows for a particular change have the same index in all arrays. Unchanged rows do not have entries in the arrays.

If the user makes a change to a single cell in col2, the following array elements contain the edit operation, the edited cell value, and the original cell value:

```
Form.mygrid.RowStatus.Action[1]
Form.mygrid.col2[1]
Form.mygrid.original.col2[1]
```

If the user changes the values of the cells in col1 and col3 in one row and the cell in col2 in another row, the information about the original and changed values is in the following array entries:

```
Form.mygrid.RowStatus.Action[1]
Form.mygrid.col1[1]
Form.mygrid.original.col1[1]
Form.mygrid.col3[1]
Form.mygrid.original.col3[1]
Form.mygrid.RowStatus.Action[2]
Form.mygrid.col2[2]
Form.mygrid.original.col2[2]
```

The remaining cells in the arrays (for example, Form.mygrid.col2[1] and Form.mygrid.original.col2[1]) have the original, unchanged values.

Example: editing data in a grid

The following example creates an editable grid. For code brevity, the example handles only three of the fields in the Employee table. A more realistic example would include, at a minimum, all seven table fields. It might also hide the contents of the Emp_ID column or display the Department name (from the Departmt table), instead of the Department ID.

To create the editable grid:

1. Create a ColdFusion page with the following content:

```
<cfquery name="empdata" datasource="cfdocexamples">
  SELECT * FROM Employee
</cfquery>

<cfform name="GridForm"
  action="handle_grid.cfm">

  <cfgrid name="employee_grid"
    height=425
    width=300
    vspace=10
    selectmode="edit"
    query="empdata"
    insert="Yes"
    delete="Yes">

    <cfgridcolumn name="Emp_ID"
      header="Emp ID"
```

```
        width=50
        headeralign="center"
        headerbold="Yes"
        select="No">

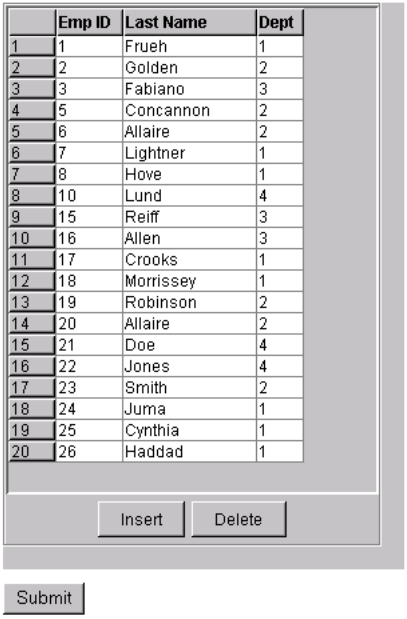
      <cfgridcolumn name="LastName"
        header="Last Name"
        width=100
        headeralign="center"
        headerbold="Yes">

      <cfgridcolumn name="Dept_ID"
        header="Dept"
        width=35
        headeralign="center"
        headerbold="Yes">

    </cfgrid>
    <br>
    <cfinput name="submitit" type="Submit" value="Submit">
</cfform>
```

2. Save the file as grid2.cfm.
3. View the results in your browser.

   The following figure shows the output of this code:



	Emp ID	Last Name	Dept
1	1	Frueh	1
2	2	Golden	2
3	3	Fabiano	3
4	5	Concannon	2
5	6	Allaire	2
6	7	Lightner	1
7	8	Hove	1
8	10	Lund	4
9	15	Reiff	3
10	16	Allen	3
11	17	Crooks	1
12	18	Morrissey	1
13	19	Robinson	2
14	20	Allaire	2
15	21	Doe	4
16	22	Jones	4
17	23	Smith	2
18	24	Juma	1
19	25	Cynthia	1
20	26	Haddad	1

The following sections describe how to write the handle_grid.cfm page to process user edits to the grid.

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfgrid name="employee_grid"`
`height=425`
`width=300`
`vspace=10`
`selectmode="edit"`
`query="empdata"`
`insert="Yes"`
`delete="Yes">`
	Populates a `cfgrid` control with data from the empdata query. Selecting a grid cell enables you to edit it. You can insert and delete rows. The grid is 425 X 300 pixels and has 10 pixels of space above and below it.

`<cfgridcolumn name="Emp_ID"`
`header="Emp ID"`
`width=50`
`headeralign="center"`
`headerbold="Yes"`
`select="No">`
	Creates a 50-pixel wide column for the data in the Emp_ID column of the data source. Centers a header named Emp ID and makes it bold.
	Does not allow users to select fields in this column for editing. Since this field is the table's primary key, users should not be able to change it for existing records, and the DBMS should generate this field as an autoincrement value.

`<cfgridcolumn name="LastName"`
`header="Last Name"`
`width=100`
`headeralign="center"`
`headerbold="Yes">`
	Creates a 100-pixel wide column for the data in the LastName column of the data source. Centers a header named Last Name and makes it bold.

`<cfgridcolumn name="Dept_ID"`
`header="Dept"`
`width=35`
`headeralign="center"`
`headerbold="Yes">`
`</cfgrid>`
	Creates a 35-pixel wide column for the data in the Dept_ID column of the data source. Centers a header named Dept and makes it bold.

Updating the database with the cfgridupdate tag

The `cfgridupdate` tag provides a simple mechanism for updating the database, including inserting and deleting records. It can add, update, and delete records simultaneously. It is convenient because it automatically handles collecting the `cfgrid` changes from the various form variables, and generates appropriate SQL statements to update your data source.

In most cases, use the `cfgridupdate` tag to update your database. However, this tag does not provide the complete SQL control that the `cfquery` tag provides. In particular, the `cfgridupdate` tag has the following characteristics:

- You can update only a single table.
- Rows are deleted first, then rows are inserted, then any changes are made to existing rows. You cannot modify the order of changes.
- Updating stops when an error occurs. It is possible that some database changes are made, but the tag does not provide any information on them.

To update the data source with the cfgridupdate tag:

1. Create a ColdFusion page with the following contents:

```
<html>
<head>
  <title>Update grid values</title>
</head>
<body>

<h3>Updating grid using cfgridupdate tag.</h3>

<cfgridupdate grid="employee_grid"
  datasource="cfdocexamples"
  tablename="Employee">

Click <a href="grid2.cfm">here</a> to display updated grid.

  </body>
</html>
```

2. Save the file as handle_grid.cfm.
3. View the grid2.cfm page in your browser, make changes to the grid, and then submit them.

Note: To update a grid cell, modify the cell contents, and then press Return.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
`<cfgridupdate grid="employee_grid"`	Updates the database from the Employee_grid grid.
`datasource="cfdocexamples"`	Updates the cfdocexamples data source.
`tablename="Employee"`	Updates the Employee table.

Updating the database with the cfquery tag

You can use the `cfquery` tag to update your database from the `cfgrid` changes. This provides you with full control over how the updates are made and lets you handle any errors that arise.

To update the data source with the cfquery tag:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Catch submitted grid values</title>
</head>
<body>

<h3>Grid values for Form.employee_grid row updates</h3>

<cfif isdefined("Form.employee_grid.rowstatus.action")>

  <cfloop index = "counter" from = "1" to =
```

```
      #arraylen(Form.employee_grid.rowstatus.action)#>

      <cfoutput>
        The row action for #counter# is:
        #Form.employee_grid.rowstatus.action[counter]#
        <br>
      </cfoutput>

      <cfif Form.employee_grid.rowstatus.action[counter] is "D">
        <cfquery name="DeleteExistingEmployee"
          datasource="cfdocexamples">
          DELETE FROM Employee
          WHERE Emp_ID=<cfqueryparam
               value="#Form.employee_grid.original.Emp_ID[counter]#"
               CFSQLType="CF_SQL_INTEGER" >
        </cfquery>

      <cfelseif Form.employee_grid.rowstatus.action[counter] is "U">
        <cfquery name="UpdateExistingEmployee"
          datasource="cfdocexamples">
          UPDATE Employee
          SET
            LastName=<cfqueryparam
              value="#Form.employee_grid.LastName[counter]#"
              CFSQLType="CF_SQL_VARCHAR" >,
            Dept_ID=<cfqueryparam
              value="#Form.employee_grid.Dept_ID[counter]#"
              CFSQLType="CF_SQL_INTEGER" >
          WHERE Emp_ID=<cfqueryparam
            value="#Form.employee_grid.original.Emp_ID[counter]#"
            CFSQLType="CF_SQL_INTEGER">
        </cfquery>

      <cfelseif Form.employee_grid.rowstatus.action[counter] is "I">
        <cfquery name="InsertNewEmployee"
          datasource="cfdocexamples">
          INSERT into Employee (LastName, Dept_ID)
          VALUES (<cfqueryparam
            value="#Form.employee_grid.LastName[counter]#"
              CFSQLType="CF_SQL_VARCHAR" >,
            <cfqueryparam value="#Form.employee_grid.Dept_ID[counter]#"
              CFSQLType="CF_SQL_INTEGER" >)
        </cfquery>

      </cfif>
    </cfloop>
  </cfif>

  Click <a href="grid2.cfm">here</a> to display updated grid.

  </body>
  </html>
```

2. Rename your existing handle_grid.cfm file as handle_grid2.cfm to save it, and then save this file as handle_grid.cfm.
3. View the grid2.cfm page in your browser, make changes to the grid, and then submit them.

Reviewing the code

The following table describes the code and its function:

Code	Description
`<cfif isdefined ("Form.employee_grid.rowstatus.action")> <cfloop index = "counter" from = "1" to = #arraylen(Form.employee_grid.rowstatus.action)#>`	If there is an array of edit types, changes the table. Otherwise, does nothing. Loops through the remaining code once for each row to be changed. The `counter` variable is the common index into the arrays of change information for the row being changed.
`<cfoutput> The row action for #counter# is: #Form.employee_grid.rowstatus.action[counter]#   </cfoutput>`	Displays the action code for this row: U for update, I for insert, or D for delete.
`<cfif Form.employee_grid.rowstatus.action[counter] is "D"> <cfquery name="DeleteExistingEmployee" datasource="cfdocexamples"> DELETE FROM Employee WHERE Emp_ID=<cfqueryparam value="#Form.employee_grid.original.Emp_ID [counter]#" CFSQLType="CF_SQL_INTEGER" > </cfquery>`	If the action is to delete a row, generates a SQL DELETE query specifying the Emp_ID (the primary key) of the row to be deleted.
`<cfelseif Form.employee_grid.rowstatus.action [counter] is "U"> <cfquery name="UpdateExistingEmployee" datasource="cfdocexamples"> UPDATE Employee SET LastName=<cfqueryparam value="#Form.employee_grid.LastName[counter]#" CFSQLType="CF_SQL_VARCHAR" >, Dept_ID=<cfqueryparam value="#Form.employee_grid.Dept_ID[counter]#" CFSQLType="CF_SQL_INTEGER" > WHERE Emp_ID=<cfqueryparam value="#Form.employee_grid.original.Emp_ID [counter]#" CFSQLType="CF_SQL_INTEGER"> </cfquery>`	Otherwise, if the action is to update a row, generates a SQL UPDATE query to update the LastName and Dept_ID fields for the row specified by the Emp_ID primary table key.

Code	Description
<cfelseif Form.employee_grid.rowstatus.action[counter] is "I"> <cfquery name="InsertNewEmployee" datasource="cfdocexamples"> INSERT into Employee (LastName, Dept_ID) VALUES (<cfqueryparam value="#Form.employee_grid.LastName[counter]#" CFSQLType="CF_SQL_VARCHAR" >, <cfqueryparam value="#Form.employee_grid.Dept_ID[counter]#" CFSQLType="CF_SQL_INTEGER" > </cfquery>	Otherwise, if the action is to insert a row, generates a SQL INSERT query to insert the employee's last name and department ID from the grid row into the database. The INSERT statement assumes that the DBMS automatically increments the Emp_ID primary key. If you use the version of the cfdocexamples database that is provided for UNIX installations, the record is inserted without an Emp_ID number.
</cfif> </cfloop> </cfif>	Closes the `cfif` tag used to select among deleting, updating, and inserting. Closes the loop used for each row to be changed. Closes the `cfif` tag that surrounds all the active code.

Embedding Java applets

The `cfapplet` tag lets you embed Java applets either on a ColdFusion page or in a `cfform` tag. To use the `cfapplet` tag, you must first register your Java applet using the ColdFusion MX Administrator Java Applets page (under Extensions). In the ColdFusion MX Administrator, you define the interface to the applet, encapsulating it so that each invocation of the `cfapplet` tag is very simple.

The `cfapplet` tag within a form offers several advantages over using the HTML `applet` tag:

- **Return values** The `cfapplet` tag requires a form field `name` attribute, so you can avoid coding additional JavaScript to capture the applet's return values. You can reference return values like any other ColdFusion form variable: `Form.`*`variablename`*.
- **Ease of use** The applet's interface is defined in the ColdFusion MX Administrator, so each instance of the `cfapplet` tag in your pages only needs to reference the applet name and specify a form variable name.
- **Parameter defaults** ColdFusion uses the parameter value pairs that you defined in the ColdFusion MX Administrator. You can override these values by specifying parameter value pairs in the `cfapplet` tag.

When an applet is registered, you enter just the applet source and the form variable name:

```
<cfapplet appletsource="Calculator"
  name="calc_value">
```

By contrast, with the HTML `applet` tag, you must declare all the applet's parameters every time you want to use it in a ColdFusion page.

Registering a Java applet

Before you can use a Java applet in your ColdFusion pages, you must register the applet in the ColdFusion MX Administrator.

To register a Java applet:

1. Open the ColdFusion MX Administrator by clicking the Administrator icon in the Macromedia ColdFusion MX Program group and entering the Administrator password.
2. Under Extensions, click Java Applets.

   The Java Applets page appears.
3. Click the Register New Applet button.

   The Add/Edit Applet page appears.
4. Enter options in the applet registration fields, as described in the ColdFusion MX Administrator online help. Use the Add button to add parameters.
5. Click Submit.

Using the cfapplet tag to embed an applet

After you register an applet, you can use the `cfapplet` tag to place the applet in a ColdFusion page. The `cfapplet` tag has two required attributes: `appletsource` and `name`. Because you registered the applet and you defined each applet parameter with a default value, you can invoke the applet with a very simple form of the `cfapplet` tag:

```
<cfapplet appletSource="appletname" name="form_variable">
```

Overriding alignment and positioning values

To override any of the values defined in the ColdFusion MX Administrator for the applet, you can use the optional `cfapplet` parameters to specify custom values. For example, the following `cfapplet` tag specifies custom spacing and alignment values:

```
<cfapplet appletSource="myapplet"
  name="applet1_var"
  height=400
  width=200
  vspace=125
  hspace=125
  align="left">
```

Overriding parameter values

You can override the values that you assigned to applet parameters in the ColdFusion MX Administrator by providing new values for any parameter. To override a parameter, you must have already defined the parameter and its default value in the ColdFusion MX Administrator Applets page. The following example overrides the default values of two parameters, Param1 and Param2:

```
<cfapplet appletSource="myapplet"
  name="applet1_var"
  Param1="registered parameter1"
  Param2="registered parameter2">
```

Handling form variables from an applet

The `cfapplet` tag `name` attribute corresponds to a variable in the action page, Form.*appletname*, which holds any value that the applet method returns when it is executed in the `cfform` tag.

Not all Java applets return values. For instance, graphical widgets might not return a specific value. For this type of applet, the method field in the ColdFusion MX Administrator remains empty, but you must still provide a `cfapplet name` attribute.

You can only use one method for each applet that you register. If an applet includes more than one method that you want to access, you can register the applet with a unique name for each additional method you want to use.

To reference a Java applet return value in your application page:

1. Specify the name of the method in the Add/Registered Java Applet page of the ColdFusion MX Administrator.
2. Specify the method name in the `name` attribute of the `cfapplet` tag.

When your page executes the applet, ColdFusion creates a form variable with the name that you specified. If you do not specify a method, ColdFusion does not create a form variable.

CHAPTER 28
Validating Data

This chapter explains how to validate data in Macromedia ColdFusion MX. It focuses on validating form data, but also contains information on how you can validate variable data and function parameters.

Contents

About ColdFusion MX validation 659
Validating form fields 667
Handling invalid data 669
Masking form input values 670
Validating form data with regular expressions 671
Validating form data using hidden fields 675
Validating form input and handling errors with JavaScript 680
Validating data with the IsValid function and the cfparam tag 683

About ColdFusion MX validation

Data validation lets you control data that is entered into an application by ensuring that the data conforms to specific type or formatting rules. Validation techniques have the following features:

- They let you provide feedback to users so that they can immediately correct information they provide. For example, a form can provide immediate feedback when a user enters a name in a telephone number field, or the form could force the user to enter the number in the correct format.
- They help prevent application errors that might arise when processing invalid data. For example, a validation test can prevent a variable that is used in a calculation from having nonnumeric data.
- They can help enhance security by preventing malicious users from providing data that takes advantage of system security weaknesses, such as buffer overrun attacks.

ColdFusion provides several techniques to ensure that data is valid. These include techniques for validating form data and for validating ColdFusion variables. They also include techniques for validating form data before the user submits it to ColdFusion, or on the ColdFusion server.

When you design data validation you consider the following factors:

The validation technique Whether to validate on the client's browser or on the server, and the specific server- or client-side validation technique, such as whether to validate when a field loses focus or when the user submits the form.

The validation type The specific method that you use to validate the data, including the rules that you apply to test the data validity, such as testing for a valid telephone number.

The following sections describe the ColdFusion MX validation techniques and provide information on selecting a technique that is appropriate for your application. They also describe the validation types that ColdFusion MX supports. Later sections describe particular techniques in detail.

Validation techniques

Different validation techniques apply to different ColdFusion tags or coding environments; for example, you can use masking only in HTML and Macromedia Flash format `cfinput` tags. Validation techniques also vary in where and when they execute; for example, on the client browser when the user submits form data, or on the server when processing data.

The following table describes the ColdFusion MX validation techniques:

Validation technique	Applies to	Where and when performed	Description
mask (`mask` attribute)	HTML and Flash format `cfinput` tags	On the client as the user enters data	ColdFusion MX generates JavaScript or ActionScript to directly control the data a user enters by specifying a pattern. For example, 999-999-9999 requires a user to enter ten digits, and automatically fills in the dash (-) separators to create a formatted telephone number. For detailed information on using masks, see "Handling invalid data" on page 669.
onBlur (`validateat="onBlur"` attribute)	`cfinput` and `cftextarea` tags	On the client when the data field loses focus	In HTML and XML format, ColdFusion MX generates JavaScript that runs on the browser to check whether entered data is valid and provide immediate feedback, if the entry is invalid. In Flash format, uses Flash built-in validation routines.

Validation technique	Applies to	Where and when performed	Description
onSubmit (`validateat="onSubmit"` attribute)	`cfinput` and `cftextarea` tags	On the client when the user clicks Submit	In HTML or XML format, the validation logic is identical to onBlur validation, but the test is not done until the user submits the form. In Flash format, this validation type is identical to onBlur Validation. Flash checks do not differentiate between the two events for validation.
onServer (`validateat="onServer"` attribute)	`cfinput` and cftextarea tags	On the server when ColdFusion gets the submitted form	ColdFusion MX checks submitted data for validity and runs a validation error page if the data is not valid. You can use the `cferror` tag to specify the validation error page.
hidden field	All Forms, including HTML-only forms	On the server when ColdFusion gets the submitted form	ColdFusion MX uses the same validation logic as with onServer validation, but you must create additional, hidden, fields and you can use this technique with HTML tags or CFML tags. For detailed information on using hidden fields, see "Validating form data using hidden fields" on page 675.
JavaScript (`onValidate="`*`function`*`"` attribute)	`cfgrid`, `cfinput`, `cfslider`, `cftextarea`, and `cftree` tags in HTML and XML format forms	On the client, when the user clicks Submit, before field-specific onSubmit validation	ColdFusion MX includes the specified JavaScript function in the HTML page it sends to the browser, and the browser calls it. For detailed information on using JavaScript for validation, see "Validating form input and handling errors with JavaScript" on page 680.
`IsValid` function	ColdFusion variables	On the server, when the function executes	ColdFusion MX tests the variable to determine whether it follows a specified validation rule and the function returns true or false. For more information on using the `IsValid` function for validation, see "Validating data with the IsValid function and the cfparam tag" on page 683.

Validation technique	Applies to	Where and when performed	Description
`cfparam` tag	ColdFusion variables	On the server, when the tag executes	ColdFusion MX checks the specified variable. If the value does not meet the validation criteria, ColdFusion generates an expression exception. For more information on using the `cfparam` tag for validation, see, "Validating data with the IsValid function and the cfparam tag" on page 683.
`cfargument` tag	UDF and CFC function arguments	On the server, when a function is called or invoked	ColdFusion MX checks the argument value when it is passed to the function. If the value does not meet the validation criteria, ColdFusion generates an application exception. For more information on using the `cfargument` tag, see Chapter 9, "Writing and Calling User-Defined Functions," on page 171.

Note: For more information on ColdFusion error handling, see Chapter 14, "Handling Errors," on page 307

Selecting a validation technique

The following considerations affect the validation technique that you select:

- If you are validating form data, the techniques you use can vary depending on whether you are using HTML, Flash, or XML forms; for example, different form types have different validation limitations.
- Different validation techniques are appropriate for different form controls and data types.
- Available techniques vary depending on when and where you want the data validated; on the client or the server, when the user enters data or submits a form, or when ColdFusion processes a variable or function argument.
- Each technique has specific features and considerations, such as the form of user feedback, feature limitations, and so on.
- Security issues or concerns that apply to your environment or application can affect the technique you select.

The preceding table in the "Validation techniques" section described some of the considerations. The following table describes additional considerations for selecting a validation technique. For additional considerations that are specific to form fields, see "Validation type considerations" on page 667.

Validation technique	Features	Considerations	Security issues
mask (`mask` attribute)	Directly controls user input.	Limited to `cfinput` tags. Provides limited control over user input patterns.	In HTML and XML format, can be circumvented because JavaScript runs directly in the browser.
onBlur (`validateat="onBlur"` attribute)	Provides immediate feedback if a user enters invalid data.	Limited to `cfinput` and `cftextarea` tags. In HTML or XML format, requires the browser to enable JavaScript.	In HTML and XML format, can be circumvented because JavaScript runs directly in the browser.
onSubmit (`validateat="onSubmit"` attribute)	All entered data is available to the user; only the invalid data needs reentering.	Limited to `cfinput` and `cftextarea` tags. In Flash format, is identical to onBlur. In HTML or XML format, validates after all fields have been entered, and requires the browser to enable JavaScript.	In HTML and XML format, can be circumvented because JavaScript runs directly in the browser.
onServer (`validateat="onServer"` attribute)	Does not require browser support.	Limited to `cfinput` and `cftextarea` tags.	Can be circumvented because validation rules are submitted with the form.
Hidden form field	Does not require browser support. Can be used with HTML or CFML form elements.	Limited to forms.	Can be circumvented because validation rules are submitted with the form.
JavaScript (`onValidate = "function"` attribute)	Allows all on-client processing supported by the browser. Can be used with HTML or CFML form elements.	Limited to specific ColdFusion form tags. Calls a single JavaScript function. JavaScript levels of support can vary among browsers, and users can disable JavaScript in their browsers.	Can be circumvented because JavaScript runs directly in the browser.
`IsValid` function	Can be used for any variable, not just form fields. Returns a Yes or No result that you use to determine further processing.	When used with a form field, runs after the data is submitted. Must be used each time a variable needs to be validated. Provides some data type checks not available in forms validation techniques.	None

Validation technique	Features	Considerations	Security issues
`cfparam` tag	Can be used for any variable, not just form fields. The tag can set a default value in addition to validating data.	When used with a form field, the tag runs after the data is submitted. You respond to validation failures using error-handling code.	None
`cfargument` tag	Used for arguments to functions written using the `cffunction` tag.	Runs when the function is called on the server. You respond to validation failures using error-handling code.	None

Security considerations

Although form-specific validation techniques provide good methods for preventing users from submitting invalid or badly formatted data, they cannot prevent users from submitting maliciously formatted data from HTML forms. Malicious users can circumvent validation techniques that require validation on the browser using JavaScript or submission of validation rules in hidden fields. If you must use a technique for preventing malicious data submissions, consider using the following techniques:

- The `onSubmit` or `OnBlur` validation in Flash forms, which use Flash built-in validation.
- The `IsValid` function and the `cfparam`, and `cfargument` tags, which let you test variables and arguments in your CFML code.
- The `cfqueryparam` tag in `cfquery` tags, described in "Enhancing security with cfqueryparam" on page 472, which can help protect databases from malicious query input.
- The script protection option, which helps prevent cross-site scripting attacks. You can set this option on the ColdFusion MX Administrator Server Settings > Settings page or by using the Application.cfc This.scriptProtect variable or the `cfapplication` tag `scriptprotect` attribute. For more information on cross-site scripting attacks and this option, see the `cfapplication` tag page in the *CFML Reference*.

Data validation types

The following table lists the types of data you can validate when you use most ColdFusion validation techniques. It does not include mask validation. Some validation types are not available for all techniques; in these cases the table indicates the limitations. The `onBlur` and `onSubmit` validation algorithms for Flash format forms might vary from those described in the following table, and most commonly have less functionality. For more detailed descriptions of the onServer validation algorithms, see the table in "Validating form data using hidden fields" on page 675.

Type field	Description
date	When validating on the server, allows any date/time format that returns true in the `IsDate` function, including a time value. When validating on the client, same as USdate.
USdate *	A U.S. date of the format mm/dd/yy, with 1- or 2-digit days and months, and 1- through 4-digit years. The separators can be slash (/), hyphen (-), or period (.) characters
eurodate *	A date of the format dd/mm/yy, with 1- or 2-digit days and months, and 1- through 4-digit years. The separators can be slash (/), hyphen (-), or period (.) characters.
time *	When validating on the server, allows any date/time format that returns True in the `IsDate` function, including a date value. When validating on the client, allows a time of format hh:mm[:ss] [A/PM].
float *	A number; allows integers. When validating form fields on the server, integer values are converted to real numbers.
numeric	A number; allows integers. When validating form fields on the server, integer values are unchanged.
integer *	An integer.
range *	A numeric range specified by a `range` attribute or `max` and `min` attributes.
boolean	A value that can be converted to a Boolean value: Yes, No, True, or False, (all case-independent), or a number.
telephone *	Standard U.S. telephone formats. Allows an initial 1 long-distance designator and up to 5-digit extensions, optionally starting with x.
zipcode *	U.S. 5- or 9-digit ZIP code format #####-####. The separator can be a hyphen (-) or a space.
creditcard *	Strips blanks and dashes; verifies number using mod10 algorithm. The number must have 13-16 digits.
ssn * *or* social_security_number *	US. Social Security number format, ###-##-####. The separator can be a dash (-) or a space.
email *	A valid e-mail address of the form name@server.domain. ColdFusion validates the format only; it does not check that entry is a valid active e-mail address.
URL *	A valid URL pattern; supports http, https, ftp file, mailto, and news URLs.
guid *	A unique identifier that follows the Microsoft/DCE format, xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx, where x is a hexadecimal number.

Type field	Description
uuid *	A universally unique identifier (UUID) that follows the ColdFusion format, xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxxxxxx, where x is a hexadecimal number.
regex * *or* regular_expression *	Matches the value against a regular expression specified in a `pattern` attribute. Valid in HTML and XML format only; ignored in Flash format.

* Indicates attribute values that are not valid for the `cfargument` tag.

Note: For more details on how ColdFusion handles data when it does onServer and hidden field validation, see "Validating form data using hidden fields" on page 675

The following validation types can only be used in `cfinput` tags:

Type	Description
maxlength	Limits the input to a maximum number of characters specified by a `maxlength` attribute.
noblanks	Does not allow fields that consist only of blanks. ColdFusion uses this validation only if the `required` attribute is True.
SubmitOnce	Used only with `cfform submit` and `image` types; prevents the user from submitting the same form multiple times before until the next page loads, Use this attribute, for example, to prevent a user from submitting an order form a second time before getting the confirmation for the initial order, and thereby making a duplicate order, Valid in HTML and XML format only; ignored in Flash format.

You can use the following validation types in `cfparam` and `cfargument` tags and the `IsValid` function only:

Type	Description
any	Any type of value
array	An array of values
binary	A binary value
query	A query object
string	A string value or single character
struct	A structure
variableName *	A string formatted according to ColdFusion variable naming conventions.

* This value is not valid for the `cfargument` tag `type` attribute. The `cfargument` tag does support component names as `type` attribute values. For more information, see `cfargument` in *CFML Reference*.

Validating form fields

This section describes basic validation of form fields. Later sections in this chapter describe other validation types and techniques that you can use, including regular expression validation, masking, hidden field validation, JavaScript validation, and validation using CFML tags and functions.

In basic form field validation, you do the following:

- Use a `cfinput` or `cftextarea` tag.
- Specify a validation type, such as numeric, or multiple types.
- Optionally, specify an error message.
- Optionally, specify a validation technique. (By default, ColdFusion uses onSubmit validation.)

The following example specifies onBlur validation of a telephone number:

```
Phone: <cfinput type="text" name="HPhone"
     validateat="onBlur"
     validate="required,telephone"
     message="Please enter a standard U.S. telephone number with an optional
       extension, such as x12345">
```

The following sections describe considerations for validation in `cfinput` and `cftextarea` tags, and show a more complete example.

Validation type considerations

General considerations Consider the following issues when you determine how to validate form data:

- When you validate form data using onBlur, onSubmit, onServer, or hidden form field validation, you can specify one or more validation types for each field that you validate. For example, you can specify that a field entry is required and that it must be numeric. To specify multiple validation types for onSubmit, onBlur, or onServer validation, specify the type values in a comma-delimited list.
- If you use onBlur, onSubmit, or onServer type validation, you can specify only one error message for each field that you validate. If you use hidden field validation, you can create a custom message for each validation rule (with the exception of range checking).
- In the `cfinput` tag, most validation type attributes apply only to text or password fields.

Validation algorithm differences The underlying validation code used when validating form data can differ depending on the validation technique and the form type. As a result, the algorithms used vary in some instances, including the following:

- The validation algorithms used for date/time values in onSubmit and OnBlur validation are different from those used for all server-side validation techniques.
- The algorithms used for onSubmit and OnBlur validation in Flash might vary from those used for HTML or XML format, and generally follow simpler rules.

For detailed information on the validation algorithms used for validation techniques used on the server, see "Validating form data using hidden fields" on page 675.

Validating data in XML skinnable forms

If you create an XML skinnable form and use any skin provided by Macromedia, such as the basic.xsl or silver.xsl skin, you can use all form validation techniques that are available for HTML forms.

If you use a custom skin (XSL file), the available validation techniques depend on the skin. The *cf_webroot*\CFIDE\scripts\xsl directory contains a _cfformvalidation.xsl file that implements all ColdFusion HTML form validation techniques and supports onBlur, onSubmit, onServer, and hidden form field validation. XML skin writers can include this file in their skin XSLT to implement ColdFusion validation for their skin.

Example: basic form validation

The following form asks for information that might be used when registering a new user. It checks to make sure that the user enters required information. (Only the telephone number is optional.) It also checks to make sure that the telephone number and e-mail address are properly formatted and that the number to be used in a challenge question is in the proper range. This example performs onSubmit validation. It posts back to itself, and dumps the submitted results.

```
<cfif IsDefined("form.fieldnames")>
  <cfdump var="#form#"><br>
</cfif>

<cfform name="myform" preservedata="Yes" >
  First Name: <cfinput type="text" size="15" name="firstname"
    required="yes" message="You must enter a first name."><br>
  Last Name: <cfinput type="text" size="25" name="lastname"
    required="yes" message="You must enter a last name."><br>
  Telephone: <cfinput type="text" size="20" name="telephone"
    validate="telephone" message="You must enter your telephone
    number, for example 617-555-1212 x1234"><br>
  E-mail: <cfinput type="text" size="25" name="email"
    validate="email" required="Yes"
    message="You must enter a valid e-mail address."><br>
  Password:<cfinput type="password" size="12" name="password1"
    required="yes" maxlength="12"
    message="You must enter a password."><br>
  Reenter password:<cfinput type="password" size="12" name="password2"
    required="yes" maxlength="12"
    message="You must enter your password twice."><br>
  We will ask you for the following number, in the range 100-999 if you forget
    your password.<br>
  Number: <cfinput type="text" size="5" name="chalenge"
    validate="range" range="100,999" required="Yes"
    message="You must enter a reminder number in the range 100-999."><br>
  <cfinput type="submit" name="submitit">
</cfform>
```

Handling invalid data

How you handle invalid data depends on the validation type. This section describes validation error-handling rules and considerations. For detailed information on error handling in ColdFusion, including invalid data handling, see Chapter 14, "Handling Errors," on page 307.

- For onBlur, onSubmit, or onServer validation, you can use the `cfinput` or `cftextarea` tag's `message` attribute to specify a text-only error message to display. Otherwise, ColdFusion MX uses a default message that includes the name of the form field that was invalid. (For OnServer validation, you can customize this message, as described in "Handling form field validation errors" on page 315.) The following example displays an error message when the user enters an invalid e-mail address:

```
E-mail: <cfinput type="text" size="25" name="email"
  validate="email" message="You must enter a valid e-mail address.">
```

- For hidden form validation, you can specify a text-only error message in the hidden field's `value` attribute. Otherwise, ColdFusion MX uses a default message that includes the name of the form field that was invalid. (You can customize this message, as described in "Handling form field validation errors" on page 315.) The following `cfinput` tag, for example, uses a hidden field validation to display an error message if the user enters an invalid address. (It uses onServer validation to display a different error message if the user fails to enter a number.)

```
Telephone: <cfinput type="text" size="20" name="telephone"
  validateat="onServer" required="Yes"
  message="You must enter a telephone number">
  <cfinput type="hidden" name="telephone_cfformtelephone"
  value="The number you entered is not in the correct format.<br>Use a
  number such as (617) 555-1212, 617-555-1212, or 617-555-1212 x12345">
```

- For HTML and XML format forms (using ColdFusion MX skins), most ColdFusion form tags have an `onError` attribute that lets you specify a Javascript function to run if an onSubmit error occurs.
- For the `IsValid` function, you write separate code paths to handle valid and invalid data. The following example shows a simplified case that displays an error message if the user entered an invalid e-mail address, or a different message if the address is valid:

```
<cfif IsValid("email", custEmail)>
  Thank you for entering a valid address.
  <!--- More processing would go here. --->
<cfelse>
  You must enter a valid e-mail address.<br>
  Click the Back button and try again.
</cfif>
```

- For cfparam and cfargument tags, you use standard ColdFusion MX error-handling techniques. You can include the tag in a try block and use a catch block to handle the error, or you can use a custom error-handling page. The following example form action page code uses a custom error page, expresserr.cfm, to handle the error that the cfparam tag generates if a user submits a form with an invalid e-mail address:

```
<cferror type="EXCEPTION" exception="expression" template="expresserr.cfm">
<cfif IsDefined("form.fieldnames")>
  <cfparam name="form.custEmail" type="email">
  <!--- Normal form processing code goes here. --->
</cfif>
```

Masking form input values

The cfinput tag mask attribute controls the format of data that can be entered into a text or datefield input field. You can also use a mask attribute in the cfcalendar tag. You can combine masking and validation on a field.

- In HTML and Flash form format, a mask can control the format of data entered into a text field.
- In the cfcalendar tag, and, for Flash format forms, the datefield type cfinput field, a mask can control the format of the date that ColdFusion MX uses for the date a user chooses in the displayed calendar.

Note: The standard ColdFusion MX XML skins do not support masking.

Masking text input

In text fields, ColdFusion automatically inserts literal mask characters, such as - characters in telephone numbers. Users type only the variable part of the field. You can use the following characters to mask data:

Mask character	Effect
A	Allows an uppercase or lowercase character: A–Z and a–z.
X	Allows an uppercase or lowercase character or number: A–Z, a–z, and 0–9.
9	Allows a number: 0–9.
?	Allows any character.
All other characters	Automatically inserts the literal character.

The following pattern enforces entry of a part number of the format EB-1234-c1-098765, where the user starts the entry by typing the first numeric character, such as 1. ColdFusion fills in the preceding EB prefix and all hyphen (-) characters. The user must enter four numbers, followed by two alphanumeric characters, followed by six numbers.

```
<cfinput type="text" name="newPart" mask="EB-9999-XX-999999" />
```

Note: You cannot force a user to type an A, X, 9, or question mark (?) character. To ensure that a pattern is all-uppercase or all-lowercase, use the ColdFusion MX UCase or LCase functions in the action page.

Masking cfcalendar and datefield input

In the `cfcalendar` tag and the Flash format `datefield` input control, you use the following masks to determine the format of the output. You can use uppercase or lowercase characters in the mask:

Mask	Pattern
D	Single- or double-digit day of month, such as 1 or 28
DD	Double-digit day of month, such as 01 or 28
M	Single- or double-digit month, such as 1 or 12
MM	Double-digit month, such as 01 or 12
MMM	Abbreviated month name, such as Jan or Dec
MMMM	Full month name, such as January or December
YY	Two-character year, such as 05
YYYY	Four-character year, such as 2005
E	Single-digit day of week, in the range 0 (Sunday)-6 (Saturday)
EEE	Abbreviated day of week name, such as Mon or Sun
EEEE	Full month day of week name, such as Monday or Sunday

The following pattern specifies that the Flash forms sends the date selected using a `datefield` input control to ColdFusion as text in the format 04/29/2004:

```
<cfinput name="startDate" type="datefield" label="date:" mask="mm/dd/yyyy"/>
```

Validating form data with regular expressions

You can use *regular expressions* to match and validate the text that users enter in `cfinput` and `cftextinput` tags. Ordinary characters are combined with special characters to define the match pattern. The validation succeeds only if the user input matches the pattern.

Regular expressions let you check input text for a wide variety of custom conditions for which the input must follow a specific pattern. You can concatenate simple regular expressions into complex search criteria to validate against complex patterns, such as any of several words with different endings.

You can use ColdFusion variables and functions in regular expressions. The ColdFusion server evaluates the variables and functions before the regular expression is evaluated. For example, you can validate against a value that you generate dynamically from other input data or database values.

Note: The rules listed in this section are for JavaScript regular expressions, and apply to the regular expressions used in `cfinput` and `cftextinput` tags only. These rules differ from those used by the ColdFusion functions `REFind`, `REReplace`, `REFindNoCase`, and `REReplaceNoCase`. For information on regular expressions used in ColdFusion functions, see Chapter 7, "Using Regular Expressions in Functions," on page 141.

Special characters

Because special characters are the operators in regular expressions, in order to represent a special character as an ordinary one, you must escape it by preceding it with a backslash. For example, use two backslash characters (\\) to represent a backslash character.

Single-character regular expressions

The following rules govern regular expressions that match a single character:

- Special characters are: `+ * ? . [ ^ $ ( ) { | \`
- Any character that is not a special character or escaped by being preceded by a backslash (\) matches itself.
- A backslash (\) followed by any special character matches the literal character itself; that is, the backslash escapes the special character.
- A period (.) matches any character except newline.
- A set of characters enclosed in brackets ([]) is a one-character regular expression that matches any of the characters in that set. For example, "[akm]" matches an *a*, *k*, or *m*. If you include] (closing square bracket) in square brackets, it must be the first character. Otherwise, it does not work, even if you use \].
- A dash can indicate a range of characters. For example, [a-z] matches any lowercase letter.
- If the first character of a set of characters in brackets is the caret (^), the expression matches any character except those in the set. It does not match the empty string. For example: "[^akm]" matches any character except *a*, *k*, or *m*. The caret loses its special meaning if it is not the first character of the set.
- You can make regular expressions case insensitive by substituting individual characters with character sets; for example, "[Nn][Ii][Cc][Kk]" is a case-insensitive pattern for the name Nick (or NICK, or nick, or even nIcK).
- You can use the following escape sequences to match specific characters or character classes:

Escape seq	Matches	Escape seq	Meaning
[\b]	Backspace.	\s	Any of the following white space characters: space, tab, form feed, and line feed.
\b	A word boundary, such as a space.	\S	Any character except the white space characters matched by \s.
\B	A nonword boundary.	\t	Tab.
\cX	The control character Ctrl-x. For example, \cv matches Ctrl-v, the usual control character for pasting text.	\v	Vertical tab.
\d	A digit character [0-9].	\w	An alphanumeric character or underscore. The equivalent of [A-Za-z0-9_].

Escape seq	Matches	Escape seq	Meaning
\D	Any character except a digit.	\W	Any character not matched by \w. The equivalent of [^A-Za-z0-9_].
\f	Form feed.	\n	Backreference to the nth expression in parentheses. See "Backreferences" on page 674.
\n	Line feed.	\o*octal*	The character represented in the ASII character table by the specified octal number.
\r	Carriage return.	\x*hex*	The character represented in the ASCII character table by the specified hexadecimal number.

Multicharacter regular expressions

Use the following rules to build a multicharacter regular expression:

- Parentheses group parts of regular expressions together into a subexpression that can be treated as a single unit. For example, "(ha)+" matches one or more instances of *ha*.
- A one-character regular expression or grouped subexpression followed by an asterisk (*) matches zero or more occurrences of the regular expression. For example, "[a-z]*" matches zero or more lowercase characters.
- A one-character regular expression or grouped subexpression followed by a plus sign (+) matches one or more occurrences of the regular expression. For example, "[a-z]+" matches one or more lowercase characters.
- A one-character regular expression or grouped subexpression followed by a question mark (?) matches zero or one occurrences of the regular expression. For example, "xy?z" matches either *xyz* or *xz*.
- The carat (^) at the beginning of a regular expression matches the beginning of the field.
- The dollar sign ($) at the end of a regular expression matches the end of the field.
- The concatenation of regular expressions creates a regular expression that matches the corresponding concatenation of strings. For example, "[A-Z][a-z]*" matches any capitalized word.
- The OR character (|) allows a choice between two regular expressions. For example, "jell(y|ies)" matches either *jelly* or *jellies*.
- Braces ({}) indicate a range of occurrences of a regular expression. You use them in the form "{m, n}" where *m* is a positive integer equal to or greater than zero indicating the start of the range and *n* is equal to or greater than *m*, indicating the end of the range. For example, "(ba){0,3}" matches up to three pairs of the expression *ba*. The form "{m,}" requires at least *m* occurrences of the preceding regular expression. The form "{m}" requires exactly *m* occurrences of the preceding regular expression. The form "{,n}" is not allowed.

Backreferences

Backreferencing lets you match text in previously matched sets of parentheses. A slash followed by a digit n (\n) refers to the *n*th parenthesized subexpression.

One example of how you can use backreferencing is searching for doubled words; for example, to find instances of "the the" or "is is" in text. The following example shows backreferencing in a regular expression:

```
(\b[A-Za-z]+)[ ]+\1
```

This code matches text that contains a word that is repeated twice; that is, it matches a word, (specified by the \b word boundary special character and the "[A-Za-z]+)" followed by one or more spaces (specified by "[ ]+"), followed by the first matched subexpression, the first word, in parentheses. For example, it would match "is is", but not "This is".

Exact and partial matches

ColdFusion validation normally considers a value to be valid if any of it matches the regular expression pattern. Often you might want to ensure that the entire entry matches the pattern. If so, you must "anchor" it to the beginning and end of the field, as follows:

- If a caret (^) is at the beginning of a pattern, the field must begin with a string that matches the pattern.
- If a dollar sign ($) is at the end of a pattern, the field must end with a string that matches the pattern.
- If the expression starts with a caret and ends with a dollar sign, the field must exactly match the pattern.

Expression examples

The following examples show some regular expressions and describe what they match:

Expression	Description
[\?&]value=	Any string containing a URL parameter value.
^[A-Z]:(\\[A-Z0-9_]+)+$	An uppercase Windows directory path that is not the root of a drive and has only letters, numbers, and underscores in its text.
^(\+|-)?[1-9][0-9]*$	An integer that does not begin with a zero and has an optional sign.
^(\+|-)?[1-9][0-9]*(\.[0-9]*)?$	A real number.
^(\+|-)?[1-9]\.[0-9]*E(\+|-)?[0-9]+$	A real number in engineering notation.
a{2,4}	A string containing two to four occurrences of *a*: aa, aaa, aaaa; for example, aardvark, but not automatic.
(ba){2,}	A string containing least two *ba* pairs; for example, Ali baba, but not Ali Baba.

Note: An excellent reference on regular expressions is *Mastering Regular Expressions* by Jeffrey E.F. Friedl, published by O'Reilly & Associates, Inc.

Validating form data using hidden fields

ColdFusion lets you specify form field validation on the server by using hidden form fields whose names consist of the name of the field to validate and the validation type. Hidden field validation uses the same underlying techniques and algorithms as onServer validation of ColdFusion form fields.

Hidden field validation has the following features:

- You can use it with standard HTML tags. For example, you can validate data in an HTML `input` tag. This feature was particularly useful in releases prior to ColdFusion MX 7, because the `cfinput` tag did not support all HTML `type` attributes.
- It is backward-compatible with validation prior to ColdFusion MX 7, when hidden field validation was the only way to do validation on the server.
- Because you use a separate tag for each validation type, if you specify multiple validation rules for a field, you can specify a different error message for each rule.
- You can use hidden field validation with any form field type that submits a data value, not just `input`, `cfinput`, `textarea`, or `cftextarea`.

Specifying hidden form field validation

To specify hidden field validation, you do the following:

- Create one HTML `input` element or CFML `cfinput` tag of `type="hidden"` for each validation rule.
- Specify the name of the field to validate as the first part of the hidden field name.
- Specify the type of validation, starting with an underscore character (_), as the second part of the hidden field name.
- You can specify multiple rules for each form data field. For example, to specify range and required validation for a field named myValue, create hidden myValue_cfformrange and myValue_cfformrequired fields.
- For most types of validation, specify the error message as the field `value` attribute.
- For range, maximum length, or regular expression validation, specify the rule, such as the maximum length, in the `value` attribute. For these validation types, you cannot specify a custom error message.

The following example uses hidden fields to require data in a date field and ensure that the field contains a date. It consists only of HTML tags.

```
<input type="text" name="StartDate" size="16" maxlength="16"><br>
<input type="hidden" name="StartDate_required"
    value="You must enter a start date.">
<input type="hidden" name="StartDate_date"
  value="Please enter a valid date as the start date.">
```

Use the following suffixes at the end of hidden form field names to specify the validation type. The type identifier always starts with an underscore. Several validation rules have two names you can use. The names that do not start with "_cf" were used in earlier releases and are retained for backward compatibility. For consistency and clarity, Macromedia recommends using the names that start with "_cf" in new forms.

Field name suffix	Verifies
_integer, _cfforminteger	An integer of the range -2,147,483,648 – 2,147,483,647. Treats the initial characters "$ € ¥ £ +" as valid input, but removes them from the number.
_cfformnumeric	Any numeric value. Treats the initial characters "$ € ¥ £ +"as valid input, but does NOT remove them from the number.
_float, _cfformfloat	Any value (including an integer) that can be represented as a floating point number with up to 12 significant digits. Treats the initial characters "$ € ¥ £ +" as valid input, but removes them from the number. Converts input data to a real number; for example a dump of an integer value on the action page includes a decimal point followed by a 0.
_range, _cfformrange	A numeric value within boundaries specified by the `value` attribute. Specify the range in the value attribute using the format "min=*minvalue* max=*maxvalue*." You cannot specify a custom error message for this validation.
_date, _cfformdate	A date in any format that ColdFusion can understand; converts the input to ODBC date format. Allows entry of a time part, but removes it from the ODBC value.
_cfformusdate	A date in the form m/d/y, m-d-y , or m.d.y, The m and d format can be 1 or 2 digits; y can be 2 or 4 digits. Does not convert the string to an ODBC value and does not allow a time part.
_eurodate, _cfformeurodate	A date in the form d/m/y, d-m-y, or d.m.y. The m and d format can be 1 or 2 digits; y can be 2 or 4 digits. Converts the input to ODBC date format. Allows entry of a time part, but removes it from the ODBC value.
_time, _cfformtime	A time. Can be in 12-hour or 24-hour clock format, and can include seconds in the form hh:mm:ss or a case-independent am or pm indicator. Converts the input to ODBC time format. Allows entry of a date part, but removes it from the ODBC value.
_cfformcreditcard	After stripping blanks and dashes, a number that conforms to the mod10 algorithm. Number must have 13-16 digits.
_cfformSSN, _cfformsocial_security_number	A nine-digit Social Security number. Can be of the form xxx-xx-xxxx or xxx xx xxxx.

Field name suffix	Verifies
_cfformtelephone	Standard U.S. telephone formats. Does not support international telephone numbers. Allows area codes with or without parentheses, and hyphens (-), spaces, periods, or no separators between standard number groups. Can be preceded by a 1 long-distance designator, and can end with an up-to-5 digit extension, optionally starting with x. The area code and exchange must begin with a digit in the range 1 - 9.
_cfformzipcode	A 5-digit or 9-digit U.S. ZIP code. In 9-digit codes, the final four digits must be preceded by a hyphen (-) or space.
_cfformemail	A valid e-mail address. Valid address characters are a-zA-Z0-9_- and the period and separator. There must be a single at sign (@) and the text after the @ character must include a period, as in my_address@MyCo.com or b-b.jones27@hisco.co.uk.
_cfformURL	A valid URL. Must start with http:\\, https:\\, ftp:\\, file:\\, mailto:, or news:. Can include, as appropriate, username and password designators and query strings. The main part of the address can only have the characters A-Za-z0-9 and -.
_cfformboolean	A value that can be treated as a Boolean: Yes, No, True, False, 0, 1.
_cfformUUID	A universally unique identifier (UUID) that follows the ColdFusion format, xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxxxxxx, where x is a hexadecimal number.
_cfformGUID	A unique identifier that follows the Microsoft/DCE format, xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx, where x is a hexadecimal number.
_cfformnoblanks	The field must not include blanks. ColdFusion uses this validation only if you also specify a `_required` hidden field.
_cfformmaxlength	The number of characters must not exceed the number specified by the tag `value` attribute.
_cfformregex, _cfformregular_expression	The data must match a JavaScript regular expression specified by the tag `value` attribute.
_required, _cfformrequired	Data must be entered or selected in the form field.

Hidden form field considerations

Consider the following rules and recommendations when determining whether and how to use hidden form field validation:

- Use hidden field validation if you want to validate data from standard HTML `input` tags. The `cfinput` and `cftextarea` tags include a `validateAt` attribute that provides a simpler method for specifying server-side validation.
- Consider using hidden field validation with the `cfinput` and `cftextarea` tags if you specify multiple validation rules for a single field and want to provide a separate error message for each validation.
- Do not use the suffixes listed in the table as field names.
- Adding a validation rule to a field does not make it a required field. You must add a separate `_required` hidden field to ensure user entry.

Hidden form field example

The following procedure creates a simple form for entering a start date and a salary. It uses hidden fields to ensure that you enter data and that the data is in the right format.

This example uses a self-submitting form; the same ColdFusion page is both the form page and its action page. The page uses an `IsDefined` function to check that form data has been submitted. This way, the pages does not show any results until you submit the input. The form uses HTML tags only; you can substitute these tags with the CFML equivalents.

```
<html>
<head>
  <title>Simple Data Form</title>
</head>
<body>
<h2>Simple Data Form</h2>

<!--- Form part. --->
<form action="datatest.cfm" method="Post">
  <input type="hidden"
    name="StartDate_cfformrequired"
    value="You must enter a start date.">
  <input type="hidden"
    name="StartDate_cfformdate"
    value="Enter a valid date as the start date.">
  <input type="hidden"
    name="Salary_cfformrequired"
    value="You must enter a salary.">
  <input type="hidden"
    name="Salary_cfformfloat"
    value="The salary must be a number.">
  Start Date:
  <input type="text"
    name="StartDate" size="16"
    maxlength="16"><br>
  Salary:
  <input type="text"
    name="Salary"
    size="10"
    maxlength="10"><br>
  <input type="reset"
    name="ResetForm"
    value="Clear Form">
  <input type="submit"
    name="SubmitForm"
    value="Insert Data">
</form>
<br>

<!--- Action part. --->
<cfif isdefined("Form.StartDate")>
  <cfoutput>
    Start Date is: #DateFormat(Form.StartDate)#<br>
    Salary is: #DollarFormat(Form.Salary)#
  </cfoutput>
```

```
</cfif>
</html>
```

When the user submits this form, ColdFusion scans the form fields to find any validation rules. It then uses the rules to analyze the user's input. If any of the input rules are violated, ColdFusion displays an error page with the error message that you specified in the hidden field's `value` attribute. The user must go back to the form, correct the problem, and resubmit the form. ColdFusion does not accept form submission until the user enters the entire form correctly.

Because numeric values often contain commas and currency symbols, ColdFusion automatically deletes these characters from fields with _cfforminteger and _cfformfloat rules before it validates the form field and passes the data to the form's action page. ColdFusion does not delete these characters from fields with _cfformnumeric rules.

Reviewing the code

The following table describes the code and its function:

Code	Description
`<form action="datatest.cfm"` `  method="post">`	Gathers the information from this form sends it to the dataform.cfm page (this page) using the Post method.

`<input type="hidden"`
`  name="StartDate_cfformrequired"`
`  value="You must enter a start date.">`
`<input type="hidden"`
`  name="StartDate_cfformdate"`
`  value="Enter a valid date as the`
`  start date.">`
	Requires input into the StartDate input field. If there is no input, displays the error information "You must enter a start date." Requires the input to be in a valid date format. If the input is not valid, displays the error information "Enter a valid date as the start date."

`<input type="hidden"`
`  name="Salary_required"`
`  value="You must enter a salary.">`
`<input type="cfformhidden"`
`  name="Salary_cfformfloat"`
`  value="The salary must be a number.">`
	Requires input into the Salary input field. If there is no input, displays the error information "You must enter a salary." Requires the input to be in a valid number. If it is not valid, displays the error information "The salary must be a number."

`Start Date:` `<input type="text"` `  name="StartDate" size="16"` `  maxlength="16"> `	Creates a text box called StartDate in which users can enter their starting date. Makes it 16-characters wide.
`Salary:` `<input type="text"` `  name="Salary"` `  size="10"` `  maxlength="10"> `	Creates a text box called Salary in which users can enter their salary. Makes it ten-characters wide.
`<cfif isdefined("Form.StartDate")>` `  <cfoutput>` `  Start Date is:` `  #DateFormat(Form.StartDate)# ` `  Salary is:` `  #DollarFormat(Form.Salary)#` `  </cfoutput>` `</cfif>`	Displays the values of the StartDate and Salary form fields only if they are defined. They are not defined until you submit the form, so they do not appear on the initial form. Uses the `DateFormat` function to display the start date in the default date format. Uses the `DollarFormat` function to display the salary with a dollar sign and commas.

Validating form input and handling errors with JavaScript

ColdFusion MX lets you write your own validation routines in JavaScript, and lets you create JavaScript error handlers.

Validating input with JavaScript

In addition to native ColdFusion input validation using the `validate` attribute of the `cfinput` and `cftextarea` tags, the following tags support the `onValidate` attribute, which lets you specify a JavaScript function to handle your `cfform` input validation:

- `cfgrid`
- `cfinput`
- `cfslider`
- `cftextarea`
- `cftree`

ColdFusion passes the following arguments to the JavaScript function that you specify in the `onValidate` attribute:

- The form JavaScript DOM object
- The `name` attribute of the form element
- The value of the control to validate

For example, if you write the `cfinput` tag as the following:

```
<cfinput type="text"
  ...
<!--- Do not include () in JavaScript function name. --->
  onvalidate="handleValidation"
  ...
>
```

You define the JavaScript function as the following:

```
<script>
<!--
function handleValidation(form_object, input_object, object_value) {
...
}
//-->
</script>
```

Example: validating a password

The following example validates a password. The password must have at least one of each of the following: an uppercase letter, a lowercase letter, and a number. It must be between 8 and 12 characters long. If the password is invalid, the browser displays a message box. If the password is valid, it redisplays the page with a brief success message.

To use JavaScript to validate form data:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>JavaScript Validation</title>

<!--- JavaScript used for validation. --->
<script>
<!--
  // Regular expressions used for pattern matching.
  var anUpperCase = /[A-Z]/;
  var aLowerCase = /[a-z]/;
  var aNumber = /[0-9]/;
  /* The function specified by the onValidate attribute.
    Tests for existence of at least one uppercase, lowercase, and numeric
    character, and checks the length for a minimum.
    A maximum length test is not needed because of the cfinput maxlength
    attribute. */
  function testpasswd(form, ctrl, value) {
    if (value.length < 8 || value.search(anUpperCase) == -1 ||
      value.search (anUpperCase) == -1 || value.search (aNumber) == -1)
    {
      return (false);
    }
    else
    {
      return (true);
    }
  }
//-->
</script>
</head>

<body>
<h2>JavaScript validation test</h2>
<!--- Form is submitted only if the password is valid. --->
<cfif IsDefined("Form.passwd1")>
  <p>Your Password if valid.</p>
</cfif>
<p>Please enter your new password:</p>

<cfform name="UpdateForm" preservedata="Yes" >
  <!--- The onValidate attribute specifies the JavaScript validation
        function. The message attribute is the message that appears
        if the validation function returns False. --->
  <cfinput type="password" name="passwd1" required="YES"
    onValidate="testpasswd"
    message="Your password must have 8-12 characters and include uppercase
and lowercase letters and at least one number."
    size="13" maxlength="12">

<input type="Submit" value=" Update... ">
</cfform>
```

```
</body>
</html>
```

2. Save the page as validjs.cfm.
3. View the validjs.cfm page in your browser.

Handling failed validation

The `onError` attribute lets you specify a JavaScript function to execute if an onValidate, onBlur or onSubmit validation fails. For example, if you use the `onValidate` attribute to specify a JavaScript function to handle input validation, you can also use the `onError` attribute to specify a JavaScript function to handle a failed validation (that is, when `onValidate` returns a False value). If you use the `onValidate` attribute, you can also use the `onError` attribute to specify a JavaScript function that handles the validation errors. The following `cfform` tags support the `onerror` attribute:

- `cfgrid`
- `cfinput`
- `cfselect`
- `cfslider`
- `cftextinput`
- `cftree`

ColdFusion passes the following JavaScript objects to the function in the `onerror` attribute:

- The JavaScript form object
- The `name` attribute of the form element
- The value that failed validation
- The error message text specified by the CFML tag's `message` attribute

The following example shows a form that uses an `onError` attribute to tell ColdFusion to call a showErrorMessage JavaScript function that uses the `alert` method to display an error message. The function assembles the message from the invalid value and the contents of the `cfinput` tag's `message` attribute.

```
<!--- The JavaScript function to handle errors.
      Puts a message, including the field name and value, in an alert box. --->
<script>
  <!--
  function showErrorMessage(form, ctrl, value, message) {
    alert("The value " + value +" of the " + ctrl + " field " + message);
  }
  //-->
</script>

<!--- The form.
  The cfinput tags use the onError attribute to override the ColdFusion
  default error message mechanism. --->
<cfform>
  <!--- A minimum quantity is required and must be a number. --->
```

```
  Minimum Quantity: <cfinput type="Text" name="MinQuantity"
    onError="showErrorMessage" validate="numeric" required="Yes"
    message="is not a number." ><br>
  <!--- A maximum quantity is optional, but must be a number if supplied. --->
  Maximum Quantity: <cfinput type="Text" name="MaxQuantity"
    onError="showErrorMessage" validate="numeric"
      message="is not a number." ><br>
  <cfinput type="submit" name="submitit">
</cfform>
```

Validating data with the IsValid function and the cfparam tag

The `IsValid` function and `cfparam` tag validate any ColdFusion MX variable value, not just forms variables. Because they reside entirely on the ColdFusion server, they can provide a secure mechanism for ensuring that the required validation steps get performed. Users cannot evade any of the checks by modifying the form data that gets submitted. These techniques also provide greater flexibility in how you respond to user errors, because you can use full CFML syntax in your error-handling code.

These two validation techniques operate as follows:

- The `IsValid` function tests the value of a ColdFusion variable. If the value is valid, it returns True; if the value is invalid, it returns False.
- The `cfparam` tag with a `type` attribute tests the value of a ColdFusion value for validity. If the value is valid, it does nothing; if the value is invalid, it throws a ColdFusion expression exception.

You can use either technique interchangeably. The one you choose should depend on your coding style and programming practices. It can also depend on the specific information that you want to display if an error occurs.

Example: IsValid function validation

The following example checks whether a user has submitted a numeric ID and a valid e-mail address and phone number. If any of the submitted values does not meet the validation test, the page displays an error message.

```
<!--- Action code. First make sure the form was submitted. --->
<cfif isDefined("form.saveSubmit")>
  <cfif isValid("integer", form.UserID) and isValid("email", form.emailAddr)
      and isValid("telephone", form.phoneNo)>
    <cfoutput>
      <!--- Application code to update the database goes here --->
      <h3>The e-mail address and phone number for user #Form.UserID#
        have been added</h3>
    </cfoutput>
  <cfelse>
    <H3>Please enter a valid user ID, phone number, and e-mail address.</H2>
  </cfif>
  <cfelse>
</cfif>

<!--- The form. --->
```

```
<cfform action="#CGI.SCRIPT_NAME#">
  User ID:<cfinput type="Text" name="UserID"><br>
  Phone: <cfinput type="Text" name="phoneNo"><br>
  E-mail: <cfinput type="Text" name="emailAddr"><br>
  <cfinput type="submit" name="saveSubmit" value="Save Data"><br>
</cfform>
```

Examples: cfparam tag validation

The following two examples use `cfparam` tags to do the same tests as in the "Example: IsValid function validation" on page 683. They check whether a user has submitted a numeric ID and a valid e-mail address and phone number. If any of the submitted values does not meet the validation test, ColdFusion throws an expression exception.

In the first example, the error is handled by the exprerr.cfm page specified in the `cferror` tag. In this case, if the user made multiple errors, ColdFusion lists only one.

In the second example, each invalid field is handled in a separate try/catch block. In this case, the user gets information about each error.

Using an error-handling page

The self-posting form and action page looks as follows:

```
<!--- Action part of the page. --->
<!--- If an expression exception occurs, run the expresser.cfm page. --->
<cferror type="EXCEPTION" exception="expression" template="expresserr.cfm">
<!--- Make sure the form was submitted. --->
<cfif isDefined("form.saveSubmit")>
<!--- Use cfparam tags to check the form field data types. --->
     <cfparam name="form.emailAddr" type="email">
     <cfparam name="form.UserID" type="integer">
     <cfparam name="form.phoneNo" type="telephone">
  <!--- Application code to update the database goes here. --->
  <cfoutput>
    <h3>The e-mail address and phone number for user #Form.UserID#
      have been added</h3>
  </cfoutput>
</cfif>

<!--- The form. --->
<cfform action="#CGI.SCRIPT_NAME#">
  User ID:<cfinput type="Text" name="UserID"><br>
  Phone: <cfinput type="Text" name="phoneNo"><br>
  E-mail: <cfinput type="Text" name="emailAddr"><br>
  <cfinput type="submit" name="saveSubmit" value="Save Data"><br>
</cfform>
```

The expresserr.cfm page looks as follows:

```
<cfoutput>
  You entered invalid data.<br>
  Please click the browser Back button and try again<br>
  #cferror.RootCause.detailMessage#
</cfoutput>
```

Using cftry and cfcatch tags

The self-posting form and action page looks as follows:

```
<!--- Use a Boolean variable to indicate whether all fields are good. --->
<cfset goodData="Yes">
<!--- Make sure the form was submitted. --->
<cfif isDefined("form.saveSubmit")>
<!--- The cftry block for testing the User ID value. --->
  <cftry>
<!--- The cfparam tag checks the field data types. --->
    <cfparam name="form.UserID" type="integer">
<!--- If the data is invalid, ColdFusion throws an expression exception. --->
<!--- Catch and handle the exception. --->
    <cfcatch type="expression">
      <!--- Set the data validity indicator to False. --->
      <cfset goodData="No">
      <cfoutput>
        Invalid user ID.<br>
        #cfcatch.detail#<br><br>
      </cfoutput>
    </cfcatch>
  </cftry>
<!--- The cftry block for testing the e-mail address value. --->
  <cftry>
    <cfparam name="form.emailAddr" type="email">
    <cfcatch type="expression">
      <cfset goodData="No">
      <cfoutput>
        Invalid e-mail address.<br>
        #cfcatch.detail#<br><br>
      </cfoutput>
    </cfcatch>
  </cftry>
<!--- The cftry block for testing the telephone number value. --->
  <cftry>
    <cfparam name="form.phoneNo" type="telephone">
    <cfcatch type="expression">
      <cfset goodData="No">
      <cfoutput>
        Invalid telephone number.<br>
        #cfcatch.detail#<br><br>
      </cfoutput>
    </cfcatch>
  </cftry>
  <!--- Do this only if the validity indicator was not set to False in a
      validation catch block. --->
  <cfif goodData>
    <!--- Application code to update the database goes here. --->
    <cfoutput>
      <h3>The e-mail address and phone number for user #Form.UserID#
        have been added</h3>
    </cfoutput>
  </cfif> <!--- goodData is True--->
</cfif> <!--- Form was submitted. --->
```

```
<cfform action="#CGI.SCRIPT_NAME#" preservedata="Yes">
  User ID:<cfinput type="Text" name="UserID"><br>
  Phone: <cfinput type="Text" name="phoneNo"><br>
  E-mail: <cfinput type="Text" name="emailAddr"><br>
  <cfinput type="submit" name="saveSubmit" value="Save Data"><br>
</cfform>
```

CHAPTER 29
Creating Forms in Macromedia Flash

This chapter describes how to create effective forms in Macromedia Flash format. In this format, Macromedia ColdFusion MX 7 displays forms using Flash, not HTML. This chapter describes Flash forms and how they differ from HTML forms, and discusses how to create Flash forms.

Contents

About Flash forms 687

Building Flash forms 689

Binding data in Flash forms 699

Setting styles and skins in Flash forms 700

Using ActionScript in Flash forms 704

Best practices for Flash forms 707

About Flash forms

ColdFusion MX can deliver forms to the client in Macromedia Flash (SWF) file format. ColdFusion automatically generates the Flash binary from your CFML code and displays it on the client. Flash forms have the following advantages over HTML forms:

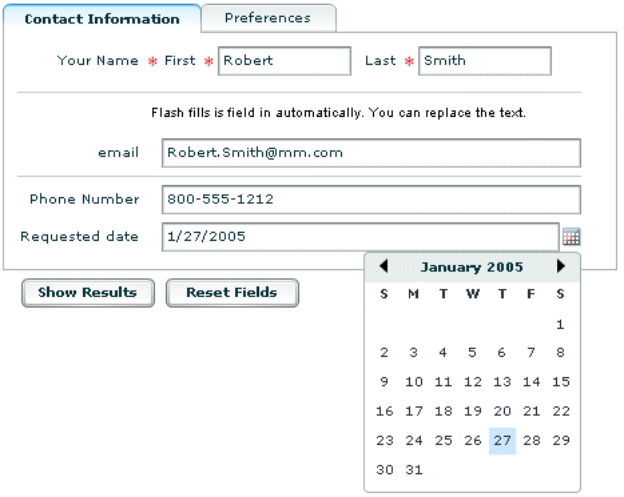
- They are browser-independent. Flash Player works in all commonly used browsers on Windows and Macintosh systems, and in Netscape and Mozilla on Linux.
- By default, they present a modern, visually pleasing appearance, and you can apply predefined color skins or customize the appearance with specifications similar to those in a Cascading Style Sheet (CSS).
- They let you develop complex, multipart forms that do no require multiple pages, by using tabbed or accordion-style dialog boxes.
- They automatically do much of the layout work for you.

Note: Flash form configuration requirements differ from ColdFusion MX requirements. For example, Flash forms might not work with all J2EE servers supported by ColdFusion MX. For more information, see *Installing and Using ColdFusion MX*.

In addition to creating Flash forms, ColdFusion MX lets you specify Flash format for `cfcalendar`, `cftree`, and `cfgrid` tags. Use these tags to embed Flash calendar choosers, trees, and grids in HTML forms, to eliminate the need to use Java applets. This chapter does not specifically discuss using Flash grids and trees in HTML forms; however, the information in this chapter about grids and trees also applies to these elements.

A Flash form example

Flash forms provide many features that help you quickly create easy-to-use, professional-looking, complex forms. The following figure contains a two-tab form that shows many of these features:



This form includes the following features:

- Each tab contains a different section of the overall form, and users can enter data on both tabs before submitting the form. This technique can eliminate the need for multiple forms on multiple HTML pages.
- The first and last names are required fields, indicated by the red asterisks.
- The Flash form automatically fills the e-mail field with data from the name fields, but the user can override this information.
- When the user selects the date field, a calendar automatically opens for picking the date.

Flash form CFML differences from HTML forms

Because ColdFusion MX sends a Flash form to the client in SWF format, everything inside a Flash form is rendered by Flash. Rendering the form in Flash has several effects:

- Plain text and HTML tags in the body of a Flash Form have no effect.
- You must specify all form content inside CFML tags that support Flash forms.

- ColdFusion provides two tags that let you take advantage of Flash features and perform tasks that you would otherwise do in HTML: you use the `cfformitem` tag to add text blocks and horizontal and vertical rules to your form, and you use the `cfformgroup` tag to structure your form.
- Standard ColdFusion forms tags, such as `cfinput` and `cftree`, include attributes that work only with Flash forms, and attribute values that let you specify form style and behavior. These include the `skin` attribute with many Flash-specific `style` attribute values for appearance, and the `bind` attribute for filling a field value with data from other fields.

The reference pages for the individual tags in *CFML Reference* describe the form tags and their features, indicating which attributes and values work with Flash forms. This chapter describes how you can use CFML tags to build effective Flash forms.

Building Flash forms

You build Flash forms using standard ColdFusion form tags, plus the `cfformgroup` and `cfformitem` tags. These tags create the *elements* of the form, as follows:

- The `cfcalendar`, `cfgrid`, `cfinput`, `cfselect`, `cftextarea`, and `cftree` tags create *controls* for data display and user input.
- The `cfformitem` tag lets you add formatted or unformatted text, spacers, and horizontal and vertical rules without using HTML.
- The `cfformgroup` tag creates *containers*, such as horizontally aligned boxes or tabbed navigators, that let you group, organize, and structure the form contents.

Flash forms follow a hierarchical structure of containers and *children*.

1. The `cfform` tag is the master container, and its contents are child containers and controls.
2. The `cfformgroup` tag defines a container that organizes its child elements.
3. All other tags create individual controls, including display elements such as rules.

For example, the figure in the "About Flash forms" section has the following hierarchical structure of containers and children. (This outline only shows the structure of the page that is visible in the figure. It omits the structure of the Preferences tab.)

```
1 cfform
2   cfformgroup type="tabnavigator" -- Tab navigator container
3     cfformgroup type="page" -- Tabbed page container, child of tabnavigator
4       cfformgroup type="horizontal" -- Aligns its two children horizontally
5         cfinput type="text" -- First name input control
5         cfinput type="spacer" -- Space between the name input controls
5         cfinput type="text" -- Last name input control
4       cfformitem type="hrule" -- Displays a rule
4       cfformitem type="html" -- Displays formatted text
4       cffinput type="text" -- E-mail input control
4       cfformitem type="hrule" -- Displays a rule
4       cfinput type="text" -- Phone number input control
4       cfinput type="spacer" -- Space between the phone and date controls
4       cfinput type="datefield" -- Date input control
3     cfinput type="page" -- Second tabbed page container for preferences
      .
```

```
      .
2    cfformgroup type="horizontal" -- Follows the tabnavigator in the form
3    cfinput type="submit" -- Submit button control
3    cfinput type="reset" -- Reset button control
```

The following sections describe how you use the various Flash form elements to build a Flash form.

Adding text, images, rules, and space with the cfformitem tag

Because Flash forms do not support inline HTML, you use the `cfformitem` tag to add text blocks and horizontal and vertical rules to your form. (Flash form controls, such as `cfinput`, use the `label` or `value` attribute to specify text labels.) You can also use the `cfformitem` tag to add spacers between visual form elements.

The `cfformitem type="hrule"` and `cfformitem type="vrule"` tags put horizontal and vertical rules in the form. You can specify the rule thickness, color, and shadow characteristics by using style specifications. For more information on specifying rule styles, see "Styles for cfformitem with hrule or vrule type attributes" in Chapter 4, "ColdFusion MX Flash Form Style Reference" in *CFML Reference*.

The `cfformitem type="spacer"` tag puts a blank space of the specified height and width on the form. This tag type does not use styles; it can be useful in improving the form appearance by helping you control the form layout.

The `cfformitem type="text"` tag lets you insert plain text in the form You can use the `style` attribute to apply a consistent format to the text.

The `cfformitem type="html"` tag lets you insert formatted text and images in the form. You can include basic HTML-style formatting tags in the body of this tag to add images and to format and style the text.

You can use the following formatting tags and attributes in the body of a `cfformitem type="html"` tag:

Tag	Valid attributes
`a`	`href` URL to link to. `target` window name; can be a standard HTML window name such as _blank.
`b`	None.
`br`	None.
`font`	`color` Must be in hexadecimal format, such as #FF00AA. Use a single number sign (#) character. `face` Can be a comma-delimited list of font face names; Flash uses the first font that is available on the client system. `size` In pixels; + and -relative values are allowed.
`i`	None.
`img`	See the attribute table for the `img` tag. ***Note:*** You must close this tag with /> or an </img> tag.

Tag	Valid attributes
li	None.
p	align Must be one of the following: left, right, center.
textformat	See the attribute table for the textformat tag.
u	None.

The img tag supports the following attributes:

Attribute	Description
src	(Required) URL or pathname to a JPEG or SWF file. Images are not displayed until they have downloaded completely. Flash Player does not support progressive JPEG files.
width	Width of the image, in pixels.
height	Height of the image in pixels.
align	Horizontal alignment of the embedded image within the text field. Valid values are left and right. The default value is left.
hspace	Number of pixels of horizontal space that surround the image where no text appears. The default value is 8.
vspace	Number of pixels of vertical space that surround the image where no text appears. The default value is 8.

Note: Because of the Flash dynamic sizing rules, to ensure that the image displays properly, you might have to specify the formitem tag height attribute and the width and height attributes for the form or the containing cfformgroup tag. Also, the image always displays on a new line, not inline with text, and text that follows the image in your code occupies any available space to the right of the image.

The textformat tag is specific to Flash, and has the following attributes:

Attribute	Description
blockindent	Block indentation, in points.
indent	Indentation from the left margin to the first character in the paragraph.
leading	Amount of leading (vertical space) between lines.
leftmargin	Left margin of the paragraph, in points.
rightmargin	Right margin of the paragraph, in points.
tabstops	Custom tab stops as an array of nonnegative integers. To specify tabs in text, use the \t escape character.

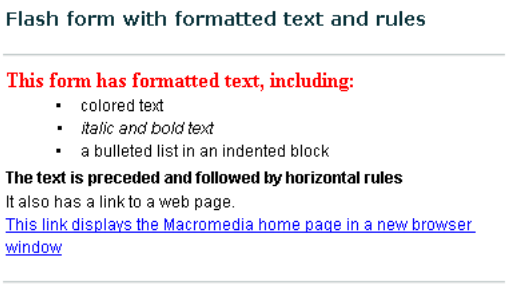
For detailed descriptions of these tags, see the Flash documentation.

The following code creates a simple Flash form that consists of a formatted text area surrounded by horizontal rules:

```
<cfform name="myform" height="220" width="400" format="Flash" >
  <!--- Use text formitem tag with style specifications for the heading. --->
```

```
  <cfformitem type="text" style="fontWeight:bold; fontSize:14;">
  Flash form with formatted text and rules
  </cfformitem>
  <!--- The spacer adds space between the text and the rule --->
  <cfformitem type="spacer" height="2" />
  <cfformitem type="hrule" />
  <cfformitem type="html">
    <b><font color="#FF0000" size="+4" face="serif">
    This form has formatted text, including:</font></b><br>
    <textformat blockindent="20" leading="2">
      <li>colored text</li>
      <li><i>italic and bold text</i></li>
      <li>a bulleted list in an indented block</li>
    </textformat>
    <p><b>The text is preceded and followed by horizontal rules</b></p>
    It also has a link to a web page.</b><br>
    <a href="http://www.macromedia.com/" target="_blank">
    <font color="#0000FF"><u>
    This link displays the Macromedia home page in a new browser window
    </u></font></a>
  </cfformitem>
  <cfformitem type="spacer" height="2" />
  <cfformitem type="hrule" />
</cfform>
```

This form appears as follows:



Using the cfformgroup tag to structure forms

ColdFusion provides form group container types that provide basic structure to a Flash form. You specify these types in the `type` attribute of the `cfformgroup` tag. Use the following container types to control the layout of controls and groups of controls:

Type	Description
`horizontal`	Arranges individual controls horizontally, and optionally applies a label to the left of the controls. Use only for arranging ColdFusion form controls, including `cfformitem` controls. As a general rule, do not use to organize `cfformgroup` containers; use the `hbox` attribute instead. If you put other `cfformgroup` tags inside a `horizontal` form group, the controls inside the included `cfformgroup` tag do not align with other controls in the horizontal group.
`vertical`	Arranges individual controls vertically, and optionally applies a label to the left (*not* top) of the controls. Use only for groups of ColdFusion form controls, including `cfformitem` controls. As a general rule, do not use to organize `cfformgroup` containers; use the `vbox` attribute instead. If you put other `cfformgroup` tags inside a `vertical` form group, the controls inside the included `cfformgroup` tag do not align with other controls in the vertical group.
`hbox`	Arranges groups of controls horizontally. Does not apply a label. Use this attribute value to arrange other `cfformgroup` containers. This tag does not enforce alignment of child controls that have labels, so you should not use it to align individual controls.
`vbox`	Arranges groups of controls vertically. Does not apply a label. Use this attribute value to arrange other `cfformgroup` containers. This tag does not enforce alignment of child controls that have labels, so you should not use it to align individual controls.
`hdividedbox`	Arranges two or more children horizontally, and puts divider handles between the children that users can drag to change the relative sizes of the children. Does not apply a label. The direct children of an `hdividedbox` container must be `cfformgroup` tags with `type` attributes other than `horizontal` or `vertical`.
`vdividedbox`	Arranges two or more children vertically, and puts divider handles between the children that users can drag to change the relative sizes of the children. Does not apply a label. The direct children of a `vdividedbox` container must be `cfformgroup` tags with `type` attributes other than `horizontal` or `vertical`.
`tile`	Arranges its children in a rectangular grid in row-first order. Does not apply a label.
`panel`	Consists of a title bar containing the `label` attribute text, a border, and a content area with vertically arranged children.
`accordion`	Puts each of its child pages in an accordion pleat with a label bar. Displays the contents of one pleat at a time. Users click the labels to expand or contract the pleat pages. Does not apply a label.
`tabnavigator`	Puts each of its children on a tabbed page. Users click the tabs to display a selected page. Does not apply a label.
`page`	The immediate child of an accordion or tab navigator container. Specifies the label on the pleat bar or tab, and arranges its child containers and controls vertically.

For more information on using the `accordion`, `tabnavigator`, and `page cfformgroup` types, see "Creating complex forms with accordion and tab navigator containers" on page 696.

Example: structuring with the cfformgroup tag

The following example shows a form with an `hdividedbox` container with two `vbox` containers. The left box uses a `horizontal` container to arrange two radio buttons. The right box uses a `tile` container to lay out its check boxes. You can drag the divider handle to resize the two boxes. When you submit the form, the ColdFusion page dumps the Form scope to show the submitted data.

```
<cfif Isdefined("Form.fieldnames")>
<cfdump var="#form#" label="form scope">
<br><br>
</cfif>

<cfform name="myform" height="200" width="460" format="Flash" skin="HaloBlue">
  <cfformitem type="html" height="20">
    <b>Tell us your preferences</b>
  </cfformitem>
  <!--- Put the pet selectors to the left of the fruit selectors. --->
  <cfformgroup type="hdividedbox" >
  <!--- Group the pet selector box contents, aligned vertically. --->
    <cfformgroup type="VBox"  height="130">
      <cfformitem type="text" height="20">
        Pets:
      </cfformitem>
      <cfformgroup type="vertical" height="80">
        <cfinput type="Radio" name="pets" label="Dogs" value="Dogs"
          checked>
        <cfinput type="Radio" name="pets" label="Cats" value="Cats">
      </cfformgroup>
    </cfformgroup>

  <!--- Group the fruit selector box contents, aligned vertically. --->
    <cfformgroup type="VBox" height="130">
      <cfformitem type="text" height="20">
        Fruits:
      </cfformitem>
      <cfformgroup type="tile" height="80" width="190" label="Tile box">
        <--- Flash requires unique names for all controls --->
        <cfinput type = "Checkbox" name="chk1" Label="Apples"
          value="Apples">
        <cfinput type="Checkbox" name="chk2" Label="Bananas"
          value="Bananas">
        <cfinput type="Checkbox" name="chk3" Label="Pears"
          value="Pears">
        <cfinput type="Checkbox" name="chk4" Label="Oranges"
        value="Oranges">
        <cfinput type="Checkbox" name="chk5" Label="Grapes"
          value="Grapes">
        <cfinput type="Checkbox" name="chk6" Label="Cumquats"
          value="Cumquats">
      </cfformgroup>
```

```
    </cfformgroup>
  </cfformgroup>
  <cfformgroup type="horizontal">
    <cfinput type="submit" name="submit" width="100" value="Show Results">
    <cfinput type="reset" name="reset" width="100" value="Reset Fields">
  </cfformgroup>
</cfform>
```

Controlling sizes in Flash forms

Sizing elements in a Flash form is something of an art, rather than a science. As a general rule, if you don't specify the `height` and `width` attributes, Flash tends to do a good job of laying out the form. However, keep in mind the following considerations:

- If you do not specify the `height` and `width` attributes in the `cfform` tag, Flash reserves the full dimensions of the visible browser window, if the form is not in a table, or the table cell, if the form is in a table, even if they are not required for the form contents. Any HTML output that precedes or follows the form causes the output page to exceed the size of the browser window.
- If you do not specify the height or width of a control, including a form group, Flash adjusts the dimensions, trying to fit the controls in the available space. For example, Flash often extends input boxes to the width of the containing control, if not otherwise specified.

In general, it is best to use the following process when you design your Flash form.

To determine the sizes of a Flash form and its controls:

1. When you first create the form, don't specify any `height` and `width` attributes on the form or its child tags. Run the form and examine the results to determine height and width values to use.
2. Specify `height` and `width` attributes in the `cfform` tag for the desired dimensions of the form. You can specify absolute pixel values, or percentage values relative to the size of the containing window.
3. Specify any `height` or `width` attributes on individual tags. These values must be in pixels.
4. Repeat step 3 for various tags, and possibly step 2, until your form has a pleasing appearance.

Repeating Flash form elements based on query data

The `repeater cfformgroup` type tells Flash Player to iterate over a query and create a set of the `cfformgroup` tag's child controls for each row in the query. For each set of child controls, `bind` attributes in the child tags can access fields in the current query row. This `cfformgroup` type lets you create Flash forms where the number of controls can change based on a query, without requiring ColdFusion to recompile the Flash SWF file for the form. This significantly enhances server performance.

Note: For more information on binding data, see "Binding data in Flash forms" on page 699.

Optionally, you can specify a start row and a maximum number of rows to use in the `repeater`. Unlike most ColdFusion tags, `cfrepeater` index values start at 0, not 1. To specify a repeater that starts on the first line of the query object and uses no more than 15 rows, use a tag such as the following:

```
<cfformgroup type="repeater" query="q1" startrow="0" maxrows="15">
```

One example that might use a repeater is a form that lets a teacher select a specific class and update the student grades. Each class can have a different number of students, so the form must have a varying number of input lines. Another example is a shopping cart application that displays the product name and quantity ordered and lets users change the quantity.

The following example uses the `cfformgroup` tag with a `repeater type` attribute value to populate a form. It creates a query, and uses the repeater to iterate over a query and create a firstname and lastname input box for each row in the query.

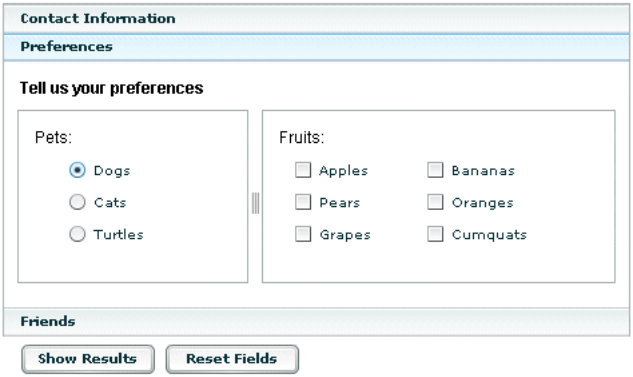
```
<cfif IsDefined("Form.fieldnames")>
  <cfdump var="#form#" label="form scope">
  <br><br>
</cfif>
<cfscript>
  q1 = queryNew("id,firstname,lastname");
  queryAddRow(q1);
  querySetCell(q1, "id", "1");
  querySetCell(q1, "firstname", "Rob");
  querySetCell(q1, "lastname", "Smith");
  queryAddRow(q1);
  querySetCell(q1, "id", "2");
  querySetCell(q1, "firstname", "John");
  querySetCell(q1, "lastname", "Doe");
  queryAddRow(q1);
  querySetCell(q1, "id", "3");
  querySetCell(q1, "firstname", "Jane");
  querySetCell(q1, "lastname", "Doe");
  queryAddRow(q1);
  querySetCell(q1, "id", "4");
  querySetCell(q1, "firstname", "Erik");
  querySetCell(q1, "lastname", "Pramenter");
</cfscript>

<cfform name="form1" format="flash" height="220" width="450">
  <cfselect label="select a teacher" name="sel1" query="q1" value="id"
    display="firstname" width="100" />
  <cfformgroup type="repeater" query="q1">
    <cfformgroup type="horizontal" label="name">
      <cfinput type="Text" name="fname"  bind="{q1.currentItem.firstname}">
      <cfinput type="Text" name="lname" bind="{q1.currentItem.lastname}">
    </cfformgroup>
  </cfformgroup>
  <cfinput type="submit" name="submitBtn" value="Send Data" width="100">
</cfform>
```

Creating complex forms with accordion and tab navigator containers

The `accordion` and `tabnavigator` attributes of the `cfformgroup` tag let you construct complex forms that would otherwise require multiple HTML pages. With accordions and tab navigator containers, users can switch among multiple entry areas without submitting intermediate forms. All data that they enter is available until they submit the form, and all form elements load at one time.

An accordion container puts each logical form page on an accordion pleat. Each pleat has a label bar; when the user clicks a bar, the current page collapses and the selected page expands to fill the available form space. The following figure shows a three-pleat accordion, open to the middle pleat, Preferences:



A tab navigator container puts each logical form page on a tabbed frame. When the user clicks a tab, the selected page replaces the previous page. The figure in "About Flash forms" shows a tab navigator container.

The following example generates a two-tab tab navigator container that gets contact information and preferences. You can change it to an accordion container by changing the `type` attribute of the first `cfformgroup` tag from `accordion` to `tabnavigator`. To prevent the accordion from having scroll bars, you must also increase the `cfform` tag `height` attribute to 310 and the `tabnavigator` tag `height` attribute to 260.

```
<cfif IsDefined("Form.fieldnames")>
  <cfdump var="#form#" label="form scope">
  <br><br>
</cfif>
<br>
<cfform name="myform" height="285" width="480" format="Flash" skin="HaloBlue">
  <cfformgroup type="tabnavigator" height="240" style="marginTop: 0">
    <cfformgroup type="page" label="Contact Information">
      <!--- Align the first and last name fields horizontally. --->
      <cfformgroup type="horizontal" label="Your Name">
        <cfinput type="text" required="Yes" name="firstName" label="First"
          value="" width="100"/>
        <cfinput type="text" required="Yes" name="lastName" label="Last"
          value="" width="100"/>
      </cfformgroup>
      <cfformitem type="hrule" />
      <cfformitem type="HTML"><textformat indent="95"><font size="-2">
        Flash fills this field in automatically.
        You can replace the text.
        </font></textformat>
      </cfformitem>
      <!--- The bind attribute gets the field contents from the firstName
          and lastName fields as they get filled in. --->
      <cfinput type="text" name="email" label="email"
```

```
        bind="{firstName.text}.{lastName.text}@mm.com">
      <cfformitem type="spacer" height="3" />
      <cfformitem type="hrule" />
      <cfformitem type="spacer" height="3" />

      <cfinput type="text" name="phone" validate="telephone" required="no"
        label="Phone Number">
      <cfinput type="datefield" name="mydate1" label="Requested date">
    </cfformgroup>

    <cfformgroup type="page" label="Preferences" style="marginTop: 0">
      <cfformitem type="html" height="20">
        <b>Tell us your preferences</b>
      </cfformitem>
      <!--- Put the pet selectors to the left of the fruit selectors. --->
      <cfformgroup type="hdividedbox" >
      <!--- Group the pet selector box contents, aligned vertically. --->
        <cfformgroup type="VBox"  height="130">
          <cfformitem type="text" height="20">
            Pets:
          </cfformitem>
          <cfformgroup type="vertical" height="80">
            <cfinput type="Radio" name="pets" label="Dogs" value="Dogs"
              checked>
            <cfinput type="Radio" name="pets" label="Cats" value="Cats">
          </cfformgroup>
        </cfformgroup>

      <!--- Group the fruit selector box contents, aligned vertically. --->
        <cfformgroup type="VBox" height="130">
          <cfformitem type="text" height="20">
            Fruits:
          </cfformitem>
          <cfformgroup type="tile" height="80" width="190" label="Tile box">
            <--- Flash requires unique names for all controls. --->
            <cfinput type = "Checkbox" name="chk1" Label="Apples"
              value="Apples">
            <cfinput type="Checkbox" name="chk2" Label="Bananas"
              value="Bananas">
            <cfinput type="Checkbox" name="chk3" Label="Pears"
              value="Pears">
            <cfinput type="Checkbox" name="chk4" Label="Oranges"
              value="Oranges">
            <cfinput type="Checkbox" name="chk5" Label="Grapes"
              value="Grapes">
            <cfinput type="Checkbox" name="chk6" Label="Kumquats"
              value="Cumquats">
          </cfformgroup>
        </cfformgroup>
      </cfformgroup>
    </cfformgroup>
  </cfformgroup>
```

```
  <cfformgroup type="horizontal">
    <cfinput type = "submit" name="submit" width="100" value = "Show Results">
    <cfinput type = "reset" name="reset" width="100" value = "Reset Fields">
  </cfformgroup>
</cfform>
```

Binding data in Flash forms

The `bind` attribute lets you set the value of the fields using the contents of other form fields. You can use the `bind` attribute with the `cftextarea` tag and any `cfinput` type that takes a value, including `hidden`. This data binding occurs dynamically as the user enters data within Flash on the client system. Flash does not send any information to ColdFusion until the user submits the form. To use the `bind` attribute to specify the field value, use the following formats:

Data source	bind attribute format
`cfinput type = "text"` or `cftextarea` text	`bind="{`*sourceName*`.text}"`
`cfinput` selected radio button	`bind="{`*sourceName*`.selectedData}"`
`cftree` selected item	`bind="{`*sourceName*`.selectedNode.getProperty('data').value}`
`cfgrid` selected item	`bind="{`*sourceName*`.selectedItem.`*COLUMNAME*`}"`
`cfselect` selected item	`bind="{`*sourceName*`.selectedItem.data}"`

Note: If you use the `bind` attribute, you cannot use the `value` attribute.

The following rules and techniques apply to the binding formats:

- The *sourceName* value in these formats is the name attribute of the tag that contains the element that you are binding to.
- You can bind to additional information about a selected item in a tree. Replace `value` with `display` to get the displayed value, or with `path` to get the path to the node in the tree.
- You can bind to the displayed value of a `cfselect` item by replacing `data` with `label`.
- If the user selects multiple items in a `cfselect` control, the selectedItem object contains the most recent selection, and a selectedItems array contains all selected items. You can access the individual values in the array, as in myTree.selectedItems[1].data. The selectedItems array exists only if the user selects multiple items; otherwise, it is undefined.
- You can use ActionScript expressions in Flash bind statements.

The following example shows how to use the values from the firstName and lastName fields to construct an e-mail address. The user can change or replace this value with a typed entry.

```
<cfformgroup type="horizontal" label="Your Name">
  <cfinput type="text" required="Yes" name="firstName" label="First"
    value="" width="100"/>
  <cfinput type="text" required="Yes" name="lastName" label="Last"
    value="" width="100"/>
</cfformgroup>
```

```
<cfinput type="text" name="email" label="email"
  bind="{firstName.text}.{lastName.text}@mm.com">
```

Setting styles and skins in Flash forms

ColdFusion provides the following methods for controlling the style and appearance of Flash forms and their elements:

Skins provide a simple method for controlling the overall appearance of your form. A single skin controls the entire form.

Styles provide a finer-grained level of control than skins. Each style specifies a particular characteristic for a single control. Many styles are also inherited by a control's children.

You can use both techniques in combination: you can specify a skin for your form and use styles to specify the appearance (such as input text font) of individual controls.

The following sections describe these methods and how you can use them. For detailed information on the style names and values that you can use, see Chapter 4, "ColdFusion MX Flash Form Style Reference" in *CFML Reference*.

Controlling form appearance with Flash skins

The `cfform` tag takes a `skin` attribute, which lets you select an overall appearance for your form. The skin determines the color used for highlighted and selected elements.

You can select the following Flash skins:

- haloBlue
- haloGreen (the default)
- haloOrange
- haloSilver

About Flash form styles

The ColdFusion Flash form tags have a `style` attribute that lets you specify control characteristics using CSS syntax. You can specify a `style` attribute in the following tags:

- `cfform`
- `cfformgroup`
- `cfcalendar`
- `cfformitem`, types `hrule` and `vrule`
- `cfgrid`
- `cfinput`
- `cfselect`
- `cftextarea`
- `cftree`

The attributes for the `cfform` and `cfformgroup` generally apply to all the form or form group's children.

Flash supports many, but not all, standard CSS styles. ColdFusion MX Flash forms only support applying specific CSS style specifications to individual CFML tags and their corresponding Flash controls and groups. You cannot use an external style sheet or define a document-level style sheet, as you can for HTML format forms.

Flash form style specification syntax

To specify a Flash style, use the following format:

```
style="stylename1: value; stylename2: value; ..."
```

For example, the following code specifies three style values for a text input control:

```
<cfinput type="text" name="text2" label="Last"
  style="borderSyle:inset; fontSize:12; backgroundColor:##FFEEFF">
```

About Flash form style value formats

Style properties can be Boolean values, strings, numbers, or arrays of these values. The following sections describe the formats for length, time, and color values.

Length format

You specify styles that take length or dimension values, including font sizes, in pixels.

The `fontSize` style property lets you use a set of keywords in addition to numbered units. You can use the following keywords when you set the `fontSize` style property. The exact sizes are defined by the client browser.

- `xx-small`
- `x-small`
- `small`
- `medium`
- `large`
- `x-large`
- `xx-large`

The following `cfinput` tag uses the `style` attribute with a `fontSize` keyword to specify the size of the text in the input box:

```
<cfinput type="text" name="text1" style="fontSize:X-large" label="Name">
```

Time format

You specify styles that take time values, such as the `openDuration` style that specifies how fast an accordion pleat opens, in milliseconds. The following example shows an `accordion` tag that takes one-half second to change between accordion pleats:

```
<cfformgroup type="accordion" height="260" style="openDuration: 500">
```

Color format

You define color values, such as those for the `backgroundColor` style, in the following formats:

Format	Description
hexadecimal	Hexadecimal colors are represented by a six-digit code preceded by two number sign characters (##). Two # characters are required to prevent ColdFusion from interpreting the character. The range of possible values is `##000000` to `##FFFFFF`.
VGA color names	VGA color names are a set of 16 basic colors supported by all browsers that support CSS. The available color names are `Aqua`, `Black`, `Blue`, `Fuchsia`, `Gray`, `Green`, `Lime`, `Maroon`, `Navy`, `Olive`, `Purple`, `Red`, `Silver`, `Teal`, `White`, and `Yellow`. Some browsers support a larger list of color names. VGA color names are not case-sensitive.

Some styles support only the hexadecimal color format.

Some controls accept multiple colors. For example, the tree control's `depthColors` style property can use a different background color for each level in the tree. To assign multiple colors, use a comma-delimited list, as the following example shows:

```
style="depthColors: ##EAEAEA, ##FF22CC, ##FFFFFF"
```

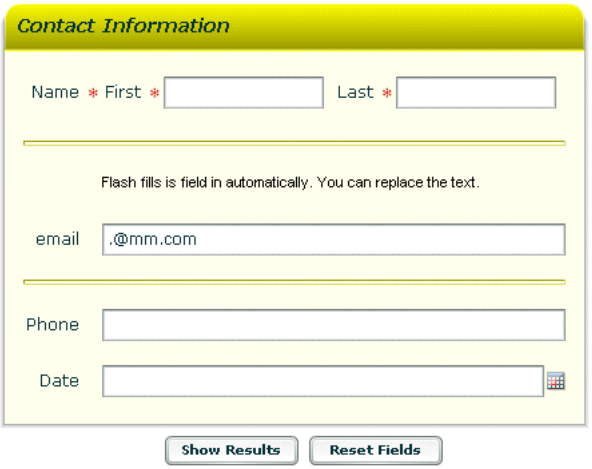
About Flash form style applicability and inheritance

Because of the way Flash control styles are implemented, some common styles are valid, but have no effect, in some tags. Therefore, in the table in "Styles valid for all controls" in Chapter 4, "ColdFusion MX Flash Form Style Reference" in *CFML Reference*, the listed styles do not cause errors when used in controls, but might not have any effect.

Styles can be inheritable or noninheritable. If a style is noninheritable, it only affects the tag, and does not affect any of its children. For example, to maintain a consistent background color in an `hbox` form group and its children tags, you must specify the color in all tags. If a style is inheritable, it applies to the tag and its children.

Example: applying styles to a Flash form

The following form uses a skin and styles to control its appearance:



The code for the form is as follows. Comments in the code explain how formatting controls and styles determine the appearance.

```
<!--- Specify the form height and width, use the HaloBlue skin.
      Note: Flash ignores a backgroundColor style set in cfform. --->
<cfform name="myform" height="390" width="440" format="Flash" skin="HaloBlue">
  <!--- The input area is a panel. Styles to specify panel characteristics.
        Child controls inherit the background color and font settings. --->
  <cfformgroup type="Panel" label="Contact Information"
      style="marginTop:20; marginBottom:20; fontSize:14; fontStyle:italic;
      headerColors:##FFFF00, ##999900; backgroundColor:##FFFFEE;
      headerHeight:35; cornerRadius:12">
    <!--- This vbox sets the font size and style, and spacing between and
          around its child controls. --->
    <cfformgroup type="vbox" style="fontSize:12; fontStyle:normal;
        verticalGap:18; marginLeft:10; marginRight:10">
      <!--- Use a horizontal group to align the first and last name fields
            and set a common label. --->
      <cfformgroup type="horizontal" label="Name" >
        <!--- Use text styles to highlight the entered names. --->
        <cfinput type="text" required="Yes" name="firstName" label="First"
          value="" width="120" style="color:##006090; fontSize:12;
          fontWeight:bold" />
        <cfinput type="text" required="Yes" name="lastName" label="Last"
          value="" width="120" style="color:##006090; fontSize:12;
          fontWeight:bold"/>
      </cfformgroup>
      <!--- Horizontal rules surround the e-mail address.
            Styles specify the rule characteristics. --->
      <cfformitem type="hrule" style="color:##999900; shadowColor:##DDDD66;
        strokeWidth:4"/>
      <cfformitem type="HTML" Style="marginTop:0; marginBottom:0">
        <textformat indent="57"> <font size="-1">Flash fills this field in
```

```
        automatically. You can replace the text.</font></textformat>
      </cfformitem>
      <cfinput type="text" name="email" label="email"
        bind="{firstName.text}.{lastName.text}@mm.com">
      <cfformitem type="hrule" style="color:##999900; shadowColor:##DDDD66;
        strokeWidth:4"/>
      <cfinput type="text" name="phone" validate="telephone" label="Phone">
      <!--- Styles control the colors of the current, selected, and
        rolled-over dates. --->
      <cfinput type="datefield" name="mydate1" label="Date"
        style="rollOverColor:##DDDDFF; selectionColor:##0000FF;
        todayColor:##AAAAFF">
    </cfformgroup> <!--- vbox --->
  </cfformgroup> <!--- panel --->
  <!--- A style centers the buttons at the bottom of the form. --->
  <cfformgroup type="horizontal"style="horizontalAlign:center">
    <cfinput type = "submit" name="submit" width="100" value = "Show Results">
    <cfinput type = "reset" name="reset" width="100" value = "Reset Fields">
  </cfformgroup>
</cfform>
```

Using ActionScript in Flash forms

ActionScript 2 is a powerful scripting language that is used in Macromedia Flash and other related products and is similar to JavaScript. You can use a subset of ActionScript 2 code in your Flash forms.

The following sections tells you how to include ActionScript in your Flash forms, and describes restrictions and additions to ActionScript that apply to ColdFusion MX Flash forms. It does not provide information on writing ActionScript. For details on ActionScript and how you can use it, see the Flash ActionScript 2 documentation, including the documents available in the Flash MX 2004 and Flex sections of Macromedia Livedocs.

Using ActionScript code in CFML

You can use ActionScript in the following attribute of tags in CFML Flash format forms:

- Form and control events, such as the `onSubmit` attribute of the `cfform` tag, or the `onChange` and `onClick` attributes of the `cfinput` tag. The attribute description on the tag reference pages in *CFML Reference* list the event attributes.
- Bind expressions, which you can use to set field values. For more information on binding data, see "Binding data in Flash forms" on page 699.

Your ActionScript code must be inline in the form attribute specification. You cannot create your own ActionScript function as a separate script and use it in the attribute value.

The following example shows a simple Fahrenheit to Celsius converter that does the conversion directly on the client, without requiring the user to submit a form to the ColdFusion server.

```
<cfform format="flash" width="200" height="150">
  <cfinput type="text" name="fahrenheit" label="Fahrenheit" width="100"
    value="68">
  <cfinput type="text" name="celsius" label="Celsius" width="100">
```

```
  <cfinput type="button" name="convert" value="Convert" width="100"
    onClick="celsius.text = Math.round((farenheit.text-32)/1.8*10)/10">
</cfform>
```

Note: You do not use the text property (for example, *fieldname*.`text`) to access hidden fields. To access a hidden field, use the format *formname.fieldname* = '*value*'.

Restricted ActionScript keywords

The following ActionScript keywords are restricted in ColdFusion MX Flash forms. If you use any of them, ColdFusion MX generates an error:

`__proto__`	`createTextField`	`loadMovie`
`attachMovie`	`Delete`	`New`
`createChild`	`duplicateMovieClip`	`registerClass`

Custom ActionScript functions

Custom ActionScript functions are the equivalent of CFML UDFs. ColdFusion does not fully support your writing or using custom ActionScript functions; however, it does include a small number of predefined custom ActionScript functions that you can use in your Flash form controls.

You can use the following functions in the ActionScript for all form controls to reset or submit the form:

- `resetForm()`
- `submitForm()`

You can use the following functions in `cfgrid` tags only to insert and delete rows in the grid:

- `GridData.insertRow(`*gridName*`)`
- `GridData.deleteRow(`*gridName*`)`

The following example shows how you can use the two GridData functions to add custom buttons that add and delete rows from a Flash form. These buttons are equivalent to the buttons that ColdFusion creates if you specify `insert="yes"` and `delete="yes"` in the `cfgrid` tag, but they allow you to specify you own button text and placement. This example puts the buttons on the side of the grid, instead of below it and uses longer than standard button labels..

```
<cfform format="flash" height="265" width="400">
  <cfformitem type="html">
    You can edit this grid as follows:
    <ul>
    <li>To change an item, click the field and type.</li>
    <li>To add a row, click the Insert Row button and type in the fields
      in the highlighted row.</li>
    <li>To delete a row, click anywhere in the row and click the
      Delete Row button</li>
    </ul>
    <p><b>When you are done, click the submit button.</b></p>
  </cfformitem>
  <!--- The hbox aligns the grid and the button vbox horizontally --->
```

```
  <cfformgroup type="hbox" style="verticalAlign:bottom;
      horizontalAlign:center">
    <!--- To make all elements align properly, all of the hbox children must
      be containers, so we must put the cfgrid tag in a vbox tag. --->
    <cfformgroup type="vbox">
      <!-- An editable grid with hard coded data (for simplicity).
        By default, this grid does not have insert or delete buttons. --->
      <cfgrid name="mygrid" height="120" width="250" selectmode="edit">
        <cfgridcolumn name="city">
        <cfgridcolumn name="state">
        <cfgridrow data="Rockville,MD">
        <cfgridrow data="Washington,DC">
        <cfgridrow data="Arlington,VA">
      </cfgrid>
    </cfformgroup>
    <!--- Group the Insert and Delete buttons vertically;
        use a vbox to ensure correct alignment. --->
    <cfformgroup type="vbox" name="buttons"  style="verticalAlign:bottom;
        horizontalAlign:center">
      <!--- Use a spacer to position the buttons. --->
      <cfformitem type="spacer" height="18" />
      <!--- Use the insertRow method in the onClick event to add a row. --->
      <cfinput type="button" name="ins" value="Insert a new row" width="125"
        onClick="GridData.insertRow(mygrid);">
      <!--- Use the deleteRow method in the onClick event to delete
        the selected row --->
      <cfinput type="button" name="del" value="Delete selected row"
          width="125" onClick="GridData.deleteRow(mygrid)">
      <cfinput type="submit" name="f1" value="Submit" width="125">
    </cfformgroup>
  </cfformgroup>
</cfform>

<!--- Dump the form if it has been submitted. --->
<cfif IsDefined("form.fieldnames")>
<cfdump var="#form#"><br>
</cfif>
```

Best practices for Flash forms

The following sections describe best practices that can help you increase the performance of Flash forms.

Minimizing form recompilation

Flash forms are sent to the client as SWF files, which ColdFusion must compile from your CFML code. The following techniques can help limit how frequently ColdFusion must recompile a Flash form.

- Only data should be dynamic. Whenever a variable name changes, or a form characteristic, such as an element width or a label changes, the Flash output must be recompiled. If a data value changes, the output does not need to be recompiled.
- Use `cfformgroup type="repeater"` if you must loop no more than ten times over no more than ten elements. This tag does not require recompiling when the number of elements changes. It does have a processing overhead that increases with the number of loops and elements, however, so for large data sets or many elements, it is often more efficient not to use the repeater.

Caching data in Flash forms

The `cfform` tag `timeout` attribute specifies how many seconds ColdFusion retains Flash form data on the server. When a Flash form is generated, the values for the form are stored in memory on the server. When the Flash form is loaded on the client, it requests these form values from the server. If this attribute is 0, the default, the data on the server is immediately deleted after the data has been requested from the Flash form.

A Flash form can be reloaded multiple times if a user displays a page with a Flash form, goes to another page, and uses the browser Back button to return to the page with the form. This kind of behavior is common with search forms, login forms, and the like. When the user returns to the original page:

- If the `timeout` value is 0, or the timeout period has expired, the data is no longer available, and ColdFusion returns a data-expired exception to the browser; in this case, the browser typically tells the user to reload the page.
- If the timeout has not expired, the browser displays the original data.

If your form data contains sensitive information, such as credit card numbers or social security numbers, you should leave the timeout set to 0. Otherwise, consider setting a timeout value that corresponds to a small number of minutes.

CHAPTER 30
Creating Skinnable XML Forms

This chapter describes how to create XML skinnable forms, which are forms that generate XForms-compliant XML and are normally formatted using an XSLT (extensible stylesheet language transformations) skin. The chapter also provides information on the format of the XML that Macromedia ColdFusion MX 7 generates for transformation by XSLT skins.

You can use XML skinnable forms with the skins that ColdFusion MX provides without having any knowledge of either XML or XSLT. Therefore, the first two sections in this chapter do not require that you have any knowledge of these technologies. The rest of this chapter, however, assumes that you have some familiarity with XML and XSLT. The chapter does not provide an introduction to, or detailed information on, writing XSLT, and it does not document XML. For information on using XML with ColdFusion MX, see Chapter 35, "Using XML and WDDX," on page 843.

Contents

About XML skinnable forms 709
Building XML skinnable forms 712
ColdFusion XML format 715
Creating XSLT skins 728

About XML skinnable forms

A ColdFusion MX form with a `format="XML"` attribute is an *XML skinnable form*. When ColdFusion MX processes an XML skinnable form, it generates the form as XML. By default, it applies an XML Stylesheet Language Transform (XSLT) skin to the XML and generates a formatted HTML page for display on the user's browser. Optionally, you can specify an XSLT file, or you can process the raw XML in your ColdFusion page.

By using XML skinnable forms, you can control the type and appearance of the forms that ColdFusion MX generates and displays. ColdFusion MX provides a set of standard skins, including a default skin that it uses if you do not specify another skin (or tell it not to apply a skin). You can also create your own XSLT skin and have ColdFusion use it to give your forms a specific style or appearance.

ColdFusion forms and XForms

ColdFusion skinnable forms conform to and extend the W3C XForms specification. This specification provides an XML syntax for defining interactive forms using a syntax that is independent of form appearance. ColdFusion MX forms tags include attributes that provide information that does not correspond directly to the XForms model, such as appearance information, validation rules, and standard HTML attributes. ColdFusion skinnable forms retain this information in XForms extensions so that an XSL transformation can use the values to determine appearance or do other processing.

For more information on XML structure of ColdFusion skinnable forms, see "ColdFusion XML format" on page 715.

The role of the XSLT skin

An XSLT skin and associated cascading style sheet (CSS) determine how an XML skinnable form is processed and displayed, as follows:

- The XSLT skin tells ColdFusion MX how to process the XML, and typically converts it to HTML for display. The skin specifies the CSS style sheet to use to format the output.
- The CSS style sheet specifies style definitions that determine the appearance of the generated output.

XSLT skins give you extensive freedom in the generated output. They let you create a custom appearance for your forms, or even different appearances for different purposes. For example, you could use the same form in an intranet and on the Internet, and change only the skin to give a different appearance (or even select different subsets of the form for display). You can also create skins that process your form for devices, such as wireless mobile devices.

How ColdFusion processes XML skinnable forms

When ColdFusion processes a `cfform` tag that specifies XML format and an XSLT skin, it does the following to the form:

1. Converts the CFML form tags into an XForms-compliant XML text format and makes it available in a variable with the same name as the form. ColdFusion ignores in-line text or HTML tags in the form, and does not pass them to the XML. (It does process HTML `option` tags that are children of a `cfselect` tag.)
2. Applies an XSLT skin to the XML; for example, to convert the form into HTML. The XSLT file specifies the CSS style sheet.
3. Returns the resulting, styled, form to the client, such as a user's browser.

If you omit the `cfform` tag `skin` attribute, ColdFusion MX uses a default skin.

If you specify `skin="none"`, ColdFusion MX performs the first step, but omits the remaining steps. Your application must handle the XML version of the form as needed. This technique lets you specify your own XSL engine, or incorporate the form as part of a larger form.

ColdFusion MX XSL skins

ColdFusion MX provides the following XSLT skins:

- basic
- basiccss
- basiccss_top
- beige
- blue
- bluegray
- default
- lightgray
- red
- silver

The XSLT skin files are located in the *cf_webroot*\CFIDE\scripts\xsl directory, and the CSS files that they use for style definitions are located in the *cf_webroot*\CFIDE\scripts\css directory.

The default skin and the basic skin format forms identically. ColdFusion uses the default skin if you do not specify a `skin` attribute in the `cfform` tag. The default and basic skins are simple skins that use tables for arranging the form contents. The basic skin uses `div` and `span` tags to arrange the elements. The skins with names of colors are similar to the basic skin, but make more use of color.

Example: a simple skinnable form

The following figure shows a simple XML skinnable form that uses the default skin to format the output:



Later sections in this chapter use this form in their examples and description.

Building XML skinnable forms

You build ColdFusion XML skinnable forms using standard ColdFusion forms tags, including `cfformgroup` and `cfformitem` tags. These tags create the *elements* of the form, the building blocks of the form.

ColdFusion converts the following tags to XML for processing by the XSLT:

Standard ColdFusion form data control tags The `cfgrid`, `cfinput`, `cfselect`, `cfslider`, `cftextarea`, and `tree` tags specify the controls that the form displays.

cfformitem tags Add individual items to your form, such as text or rules. The valid types depend on the skin.

cfformgroup tags Group, organize, and structure the form contents. The valid types depend on the skin.

These tags are designed so you can develop forms in a hierarchical structure of *containers* and *children*. Using this model, the `cfform` tag is the master container, and its contents are *children* containers and controls. Each `cfformgroup` tag defines a container that organizes its child elements.

The specific tags and attributes that you use in your form depend on the capabilities of the XSLT skin. You use only the tag and attribute combinations that the skin supports. If you are using a skin provided by a third party, make sure that the supplier provides information on the supported attributes.

Using standard ColdFusion MX form tags

You use standard ColdFusion MX form tags, such as `cfinput` or `cftree`, as you normally do in standard CFML forms to generate form input elements. ColdFusion MX maps most of these tags and their subtags (such as `option` tags in the `cfselect` tag) to equivalent XForms elements. ColdFusion maps applet and Macromedia Flash format `cfgrid` and `cftree` tags to ColdFusion XML extensions that contain Java applet or Flash objects. It converts XML format `cfgrid` and `cftree` tags to ColdFusion XML extension.

The specific attributes you can use and their meanings can depend on the skins.

Using ColdFusion MX skins The skins that are supplied with ColdFusion MX 7 support the attributes that you can use with HTML forms. You can also use `label` attributes to provide labels for the following tags:

- `cfinput` with `type` attribute values of `text`, `button`, `password`, and `file`
- `cfselect`
- `cfslider`
- `cftextarea`

Using other skins If you use any other skin, some attributes might not be supported, or the skin might support custom attributes. Get the information about the supported attributes from the XSLT skin developer.

Using cfformitem tags

ColdFusion MX does not process inline text or standard HTML tags when it generates an XML form; therefore, you use the `cfformitem` tag to add formatted HTML or plain text blocks and any other display elements, such as horizontal and vertical rules, to your form.

ColdFusion converts all `cfformitem type` attribute values to all-lowercase. For example, if you specify `type="MyType"` ColdFusion MX converts the type name to `"mytype"`.

ColdFusion MX makes no other limitations on the `cfformitem type` attributes that you can use in a form, but the XSLT skin must process the attributes to display the items.

Using ColdFusion skins The skins provided in ColdFusion MX support the following `cfformitem` types:

- `hrule`
- `text`
- `html`

The `hrule` type inserts an HTML `hr` tag, and the `text` type displays unformatted plain text.

The `html` type displays HTML-formatted text. You can include standard HTML text markup tags, such as `strong`, `p`, `ul`, or `li`, and their attributes. For example, the following text from the "Example: a simple skinnable form" section shows how you could use a `cfformitem` tag to insert descriptive text in a form:

```
<cfformitem type="html">
  <b>We value your input</b>.<br>
  <em>Please tell us a little about yourself and your thoughts.</em>
</cfformitem>
```

Using other skins If you use any other skin, the supported attributes and attribute values depend on the skin implementation. Get the information about the supported attributes and attribute values from the XSLT skin developer.

Using cfformgroup tags

The `cfformgroup` tag lets you structure forms by organizing its child tags, for example, to align them horizontally or vertically. Some skins might use `cfformgroup` tags for more complex formatting, such as tabbed navigator or accordion containers. ColdFusion MX makes no limitations on the `type` attributes that you can use in a form, but the XSLT must process the resulting XML to affect the display.

Using ColdFusion skins The skins provided in ColdFusion MX support the following `type` attribute values:

- `horizontal`
- `vertical`
- `fieldset`

The `horizontal` and `vertical` types arrange their child tags in the specified direction and place a label to the left of the group of children. The following text from the "Example: a simple skinnable form" section shows how you could use a `cfformgroup` tag to apply a *Name* label and align first and last name fields horizontally:

```
<cfformgroup type="horizontal" label="Name">
  <cfinput type="text" name="firstname" label="First" required="yes">
  <cfinput type="text" name="lastname" label="Last" required="yes">
</cfformgroup>
```

The `fieldset` type corresponds to the HTML `fieldset` tag, and groups its children by drawing a box around them and replacing part of the top line with legend text. To specify the legend, use the `label` attribute. To specify the box dimensions, use the `style` attribute with `height` and `width` values.

The following code shows a simple form group with three text controls. The `cfformgroup type="vertical"` tag ensures that the contents of the form is consistently aligned. The `cfformgroup type="horizontal"` aligns the firstname and lastname fields horizontally.

```
<cfform name="comments" format="xml" skin="basiccss" width="400"
    preservedata="Yes" >
  <cfformgroup type="fieldset" label="Contact Information">
    <cfformgroup type="vertical">
      <cfformgroup type="horizontal" label="Name">
        <cfinput type="text" size="20" name="firstname" required="yes">
        <cfinput type="text" size="25" name="lastname" required="yes">
      </cfformgroup>
      <cfinput type="text" name="email" label="E-mail" validation="email">
    </cfformgroup>
  </cfformgroup>
</cfform>
```

Note: Because XML is case-sensitive, but ColdFusion is not, ColdFusion MX converts `cfformgroup` and `cfformitem` attributes to all-lowercase letters. For example, if you specify `cfformgroup type="Random"`, ColdFusion converts the type to `random` in the XML.

Using other skins If you use any other skin, the supported attributes and attribute values depend on the skin implementation. Get the information about the supported attributes and attribute values from the skin developer.

Example: CFML for a skinnable XML form

The following CFML code creates the form shown in the figure in "About XML skinnable forms" on page 709. It shows how you can use CFML to structure your form.

```
<cfform name="comments" format="xml" skin="basiccss" width="400"
  preservedata="Yes" >
  <cfinput type="hidden" name="revision" value="12a">
  <cfformgroup type="fieldset" label="Basic Information">
    <cfformgroup type="vertical">
    <cfformgroup type="horizontal" label="Name">
      <cfinput type="text" size="20" name="firstname" required="yes">
      <cfinput type="text" size="25" name="lastname" required="yes">
    </cfformgroup>
```

```
    <cfinput type="text" name="email" label="E-mail" validate="email"
  maxlength="35">
    <cfselect name="satisfaction" style="width:120px" multiple="false"
  label="Satisfaction">
      <option selected>very satisfied</option>
      <option>somewhat satisfied</option>
      <option>somewhat dissatisfied</option>
      <option>very dissatisfied</option>
      <option>no opinion</option>
    </cfselect>
    </cfformgroup>
  </cfformgroup>
  <cfformitem name="html1" type="html">
  <p><b>We value your input</b>.<br>
  <em>Please tell us a little about yourself and your thoughts.</em></p>
  </cfformitem>
  <cftextarea name="thoughts" label="Additional Comments" rows="5"
  cols="66">We really want to hear from you!</cftextarea>
  <cfformgroup type="horizontal">
    <cfinput type="submit" name="submit" style="width:80" value="Tell Us">
    <cfinput type="reset" name="reset" style="width:80" value="Clear Fields">
  </cfformgroup>
</cfform>
```

ColdFusion XML format

This section describes the XML generated from a ColdFusion `cfform` tag and its children. It provides a building block toward creating your own XSL skins.

XML namespace use

The XML that ColdFusion generates for forms uses elements and attributes in several XML *namespaces*. Namespaces are named collections of names that help ensure that XML names are unique. They often correspond to a web standard, a specific document type definition (DTD), or a schema. In XML, the namespace name and a colon (:) precede the name of the tag that is defined in that namespace; for example `xf:model` for the XForms namespace `model` tag.

ColdFusion uses several standard XML namespaces defined by the World Wid Web Consortium (W3C). These namespaces correspond to specifications for standard XML dialects such as XHTML, XForms and XML Events. ColdFusion XML forms also use a custom namespace for skinnable forms XML extensions. The following table lists the namespaces in the XML that ColdFusion generates.

Prefix	URL	Used for
html	http://www.w3.org/1999/xhtml	Form tag information, including action, height, width, and name. XHTML compliant.
xf	http://www.w3.org/2002/xforms	XForms model (including initial field values) and XForms elements that correspond to cfform tags.

Prefix	URL	Used for
ev	http://www.w3.org/2001/xml-events	System events. Used for the `cfinput type="reset"`.
cf	http://www.macromedia.com/2004/cfform	All ColdFusion extensions, including passthrough of attributes that do not correspond to XForms elements or attributes.

XML structure

For each CFML tag, ColdFusion MX converts attributes and element values to XML in the XForms `xf:model` element, or in individual control elements, such as the XForms `xf:input`, `xf:secret`, or `xf:group` elements.

ColdFusion MX generates XForms XML in the following format. The numbers on each line indicate the level of nesting of the tags.

```
1 form tag
2   XForms model element
3     XForms instance element
4       cf:data element
3     XForms submission element
3     XForms bind element
3     XForms bind element
3   .
3   .
3   .
2   (end of model element)
2   XForms or ColdFusion extension control element
2   XForms or ColdFusion extension control element
      .
      .
      .
1 (end of form)
```

The following sections describe the data model and the elements that make up the XForms document.

Data model

The XForms data model specifies the data that the form submits. It includes information on each displayed control that can submit data, including initial values and validation information. It does not contain information about `cfformgroup` or `cfformitem` tags. The data model consists of the following elements and their children:

- One `xf:instance` element
- One `xf:submission` element
- One `xf:bind` element for each form control that can submit data

xf:instance element

The XForms `xf:instance` element contains information about the form data controls. Any control that can submit data has a corresponding instance element. If the control has an initial value, the instance element contains that value.

The `xf:instance` element contains a single `cf:data` element that contains an element for each data control: `cfgrid`, most `cfinput` tag types, `cfselect`, `cfslider`, `cftextarea`, and `cftree`. Each element name is the corresponding CFML tag's `name` attribute. For applet and Flash format `cfgrid` and `cftree` tags, the element name is the value of the `cf_param_name` parameter of the tree or grid's Java applet object. Only `cfinput` tags of types `submit`, `image`, `reset` and `button` do not have instance data, because they cannot submit data.

Each element's body contains the initial control data from the CFML tag's `value` attribute or its equivalent. For example, for a `cfselect` tag, the `xf:instance` element body is a comma-delimited list that contains the `name` attributes of all the `option` tags with a `selected` attribute. For `submit` and `image` buttons, the body contains the `name` attribute value.

The following example shows the `xf:instance` element for the form shown in the figure in "About XML skinnable forms" on page 709:

```
<xf:instance>
  <cf:data>
       <firstname/>
       <lastname/>
       <email/>
       <revision>Comment Form revision 12a</revision>
       <satisfaction>very satisfied</satisfaction>
       <thoughts>We really want to hear from you!</thoughts>
  </cf:data>
```

xf:submission element

The `xf:submission` element specifies the action when the form is submitted, and contains the values of the `cfform action` and `method` attributes.:

The following example shows the XML for the form shown in the figure in "About XML skinnable forms" on page 709:

```
<xf:submission
  action="/_MyStuff/phase1/forms/XForms/FrameExamples/Figure1.cfm"
  method="post"/>
```

xf:bind elements

The `xf:bind` elements provide information about the input control behavior, including the control type and any data validation rules. The XML has one bind element for each instance element that can submit data. It does not have bind elements for controls such as `cfformitem` tags, or `cfinput` tags with submit, input, reset, or image types. Each element has the following attributes:

Attribute	Description
id	CFML tag `name` attribute value
nodeset	XPath expression with the path in the XML to the instance element for the control
required	CFML tag `required` attribute value

Each `xf:bind` element has an `xf:extension` element with ColdFusion specific information, including type and validation values. The following table lists the cf namespace elements that are used in this section:

Element	Description
cf:attribute name="type"	Control type. One of the following: CHECKBOX, FILE, IMAGE, PASSWORD, RADIO, SELECT, SUBMIT TEXT, CFSLIDER. The TEXT type is used for `cfinput type="text"` and `cftextinput`.
cf:attribute name="onerror"	JavaScript function specified by the control's `onError` attribute, if any.
cfargument name="maxlength	Value of the control's `maxlength` attribute, if any.
cf:validate type="*valiadationtype"*	Data validation information. Has one attribute, `type`, the validation type, and one or more `cf:argument` and `cf:trigger` children. ColdFusion MX generates a `cf:validate` element for each of the following: • `cfinput` or `cftextarea validation` attribute • `cfinput` or `cftextarea range` attribute • `cfslider`: the `range` and `message` attributes are specified by a `cf:validate type="range"` element

Element	Description
cf:argument (in the body of a cf:validate element)	Data validation specification. Has one attribute, `name`, and body text. Each `cf:validate` element can have multiple `cf:argument` children, corresponding to the validation-related CFML tag attribute values, such as maximum length, and maximum and minimum range values. The element body contains the CFML attribute value. Valid `name` values are as follows. Unless specified otherwise, the name is identical to the corresponding CFML tag attribute name. • max • message • min • pattern
cf:trigger (in the body of a cf:validate element)	When to do the validation; specifies a form element `validateAt` attribute value. Has one attribute, `event`, which can be one of the following: • onBlur • onSubmit • onServer If a `validateAt` attribute specifies multiple validation triggers, the XML has one `cf:trigger` element for each entry in the list.

The following example shows the `xf:bind` element of the form shown in the figure in "About XML skinnable forms" on page 709:

```
<xf:bind id="firstname"
     nodeset="//xf:model/xf:instance/cf:data/firstname"
     required="true()">
  <xf:extension>
    <cf:attribute name="type">TEXT</cf:attribute>
    <cf:attribute name="onerror">_CF_onError</cf:attribute>
  </xf:extension>
</xf:bind>
<xf:bind id="lastname"
     nodeset="//xf:model/xf:instance/cf:data/lastname"
     required="true()">
  <xf:extension>
    <cf:attribute name="type">TEXT</cf:attribute>
    <cf:attribute name="onerror">_CF_onError</cf:attribute>
  </xf:extension>
</xf:bind>
<xf:bind id="email"
     nodeset="//xf:model/xf:instance/cf:data/email" required="false()">
  <xf:extension>
    <cf:attribute name="type">TEXT</cf:attribute>
    <cf:attribute name="onerror">_CF_onError</cf:attribute>
  </xf:extension>
</xf:bind>
<xf:bind id="satisfaction"
     nodeset="//xf:model/xf:instance/cf:data/satisfaction"
     required="false()">
  <xf:extension>
```

```
    <cf:attribute name="type">SELECT</cf:attribute>
    <cf:attribute name="onerror">_CF_onError</cf:attribute>
  </xf:extension>
</xf:bind>
<xf:bind id="thoughts"
    nodeset="//xf:model/xf:instance/cf:data/thoughts" required="false()">
  <xf:extension>
    <cf:attribute name="type">TEXT</cf:attribute>
    <cf:attribute name="onerror">_CF_onError</cf:attribute>
  </xf:extension>
</xf:bind>
```

Control elements

The XML tags that follow the `xf:bind` element specify the form controls and their layout. The XML includes one element for each form control and `cfformitem` or `cfformgroup` tag.

CFML to XML tag mapping

ColdFusion MX maps CFML tags to XForms elements and ColdFusion extensions as the following table shows:

CFML tag	XML tag
cfinput type="text"	xf:input
cfinput type="password"	xf:secret
cfinput type="hidden"	None: instance data only
cfinput type="file"	xf:upload
cfinput type="radio"	xf:select1
cfinput type="checkbox"	xf:select
cfinput type="button"	xf:trigger
cfinput type="image"	xf:submit
cfinput type="reset"	xf:submit
cfinput type="submit"	xf:submit
cfselect multiple="false"	xf:select1
cfselect multiple="true"	xf:select
cftextarea	xf:textarea
cfslider	xf:range
cfgrid	cf:grid
cftree	cf:tree
cfformitem type="text"	xf:output
cfformitem type="html"	xf:output

CFML tag	XML tag
cfformitem type="*" (all but text, html)	xf:group appearance="*"
cfformgroup type="*"	xf:group appearance="*"

ColdFusion converts `cfformitem` tags with text and html `type` attributes to XForms `output` elements with the tag body in a <![CDATA[section. It converts all other `cfformitem` tags to XForms `group` elements, and sets each element's `appearance` attribute to the `cfformitem` tag's `type` attribute. The XSLT must process these elements to produce meaningful output. For example, the ColdFusion default skin transform displays the `xf:output` text blocks and processes the `xf:group appearance="hrule"` element, but it ignores all other `xf:group` elements.

General control element structure

Each control element that can be represented by a standard XForms control element has the following general structure. (For information on XML element structure for `cfformitem`, `cfformgroup`, `cfgrid`, and `cftree` tags, see the following sections.)

```
<xf:tagname bind="bindid" id="bindid">
  <xf:label>label</xf:label>
  <xf:extension>
    <cf:attribute name="type">controltype</cf:attribute>
    <cf:attribute name="attribname>attribvalue</cf:attribute>
    <cf:attribute name="attribname>attribvalue</cf:attribute>
    .
    .
    .
  </xf:extension>
</xf:tagname>
```

The following table describes the variable parts of this structure:

Part	Description
tagname	The xf or cf namespace element name, as identified in the table in "CFML to XML tag mapping" on page 720.
bindid	ID attribute of the model `xf:bind` element for this control. Specified by the control's CFML tag `name` attribute.
label	Control label text. Specified by one of the following: • The CFML tag `label` attribute • The `value` attribute of the `radiobutton`, `submit`, and `reset` `cfinput` tags • The tag body content of `cfselect` `option` subtags, • Not used for `cfgrid` and `cftree` tags.
controltype	Type of control. One of the following: • The `cfinput` `type` attribute • `Select`, `slider`, or `textarea`, for the `cfselect`, `cfslider`, or `cftextarea` tags, respectively. • Not used for `cfgrid` and `cftree` tags.

Part	Description
attribname	Name of a CFML tag attribute. There is a `cf:attribute` tag for each attribute specified in the CFML code that does not otherwise have an entry in the XML.
attribvalue	Value of a CFML tag attribute.

Tag-specific element structure

The following sections describe tag-specific features of the XML for several types of input tags. It is not all-inclusive. For the specific structure of any ColdFusion form tag, see the XML generated from the tag by ColdFusion MX.

Selection tags

Tags that are used for selection, `cfselect`, `cfinput type="radio"`, and `cfinput type="checkbox"` are converted to XForms `select` and `select1` elements. These elements include an `xf:choices` element, which in turn has an `xf:item` element for each item a user can choose. Each item normally has an `xf:label` element and an `xf:value` element. Check boxes have a single item; select and radio button controls have more than one.

The following example shows the CFML code for a group of two radio buttons, followed by the generated XML control elements. This example also shows the use of a `cfformgroup` tag to arrange and label the radio button group.

CFML

```
<cfformgroup type="horizontal" label="Accept?">
  <cfinput type = "Radio" name = "YesNo" value = "Yes" checked>
  <cfinput type = "Radio" name = "YesNo" value = "No">
</cfformgroup>
```

XML

```
<xf:group appearance="horizontal">
  <xf:label>Accept?</xf:label>
  <xf:extension/>
  <xf:select1 appearance="full" bind="YesNo" id="YesNo">
    <xf:extension>
      <cf:attribute name="type">radio</cf:attribute>
    </xf:extension>
    <xf:choices>
      <xf:item>
        <xf:label>Yes</xf:label>
        <xf:value>Yes</xf:value>
        <xf:extension>
          <cf:attribute name="checked">checked</cf:attribute>
        </xf:extension>
      </xf:item>
      <xf:item>
        <xf:label>No</xf:label>
        <xf:value>No</xf:value>
        <xf:extension/>
      </xf:item>
```

```
    </xf:choices>
  </xf:select1>
</xf:group>
```

cfgrid tags

ColdFusion MX represents a `cfgrid` tag using the `cf:grid` XML tag. This tag has four attributes: `format`, which can be Flash, Applet, or XML; and the `id`, `name`, and `bind` attributes, which all have the value of the `cfgrid` tag `name` attribute.

For applet and Flash format grids, ColdFusion inserts `cfgrid` controls in the XML as HTML embed objects in <![CDATA[sections in the body of a `cf:grid` tag. The controls can be Java applets or in SWF file format.

For XML format grids, ColdFusion converts the CFML to XML in the following format:

```
cf:grid bind="gridname" name="gridname" format="xml" id="gridname>
  <metadata>
    <cfgridAttribute1>attributeValue</cfgridAttribute1>
    ...
    (There are an entry for attributes with a specified or default value.)
  </metadata>
  <columns>
    <column cfgridcolumnAttribute1="value" ... />
    ...
  </columns>
  <rows>
    <row>
      <column1Name>row1Column1Value</column1Name>
      <column2Name>row1Column2Value</column2Name>
      ...
    </row>
    <row>
      <column1Name>row2Column1Value</column1Name>
      <column2Name>row2Column2Value</column2Name>
    </row>
    ...
  </rows>
</cf:grid>
```

The following example shows a minimal grid with two nodes.

CFML

```
<cfgrid name="mygrid" Format="xml" selectmode="Edit" width="350">
  <cfgridcolumn name="CorName" header="Course Name" >
  <cfgridcolumn name="Course_ID" header="ID">
  <cfgridrow data="one0,two0">
  <cfgridrow data="one1,two1">
</cfgrid>
```

XML

Most metadata lines are omitted for brevity:

```
<cf:grid bind="mygrid" format="XML" id="mygrid" name="mygrid">
  <metadata>
    <autowidth>false</autowidth>
    <insert>false</insert>
    <delete>false</delete>
    <sort>false</sort>
    <italic>false</italic>
    <bold>false</bold>
    <appendkey>true</appendkey>
    <highlughthref>true</highlughthref>
    <griddatalines>Left</griddatalines>
    <gridlines>true</gridlines>
    <rowheaders>true</rowheaders>
    <rowheaderalign>Left</rowheaderalign>
    <rowheaderitalic>false</rowheaderitalic>
    <rowheaderbold>false</rowheaderbold>
    <colheaders>true</colheaders>
    <colheaderalign>Left</colheaderalign>
    <colheaderitalic>false</colheaderitalic>
    <colheaderbold>false</colheaderbold>
    <selectmode>Edit</selectmode>
    <notsupported>&lt;b&gt; Browser must support Java to view ColdFusion Java
Applets&lt;/b&gt;</notsupported>
    <picturebar>false</picturebar>
    <insertbutton>insert</insertbutton>
    <deletebutton>delete</deletebutton>
    <sortAscendingButton>SortAsc</sortAscendingButton>
    <sortDescendingButton>SortDesc</sortDescendingButton>
  </metadata>
  <columns>
    <column bold="false" display="true" header="Course Name"
      headerBold="false" headerItalic="false" italic="false"
      name="CorName" select="true"/>
    <column bold="false" display="true" header="ID"
      headerBold="false" headerItalic="false" italic="false"
      name="Course_ID" select="true"/>
  </columns>
  <rows>
    <row>
      <CorName>one0</CorName>
      <Course_ID>two0</Course_ID>
    </row>
    <row>
      <CorName>one1</CorName>
      <Course_ID>two1</Course_ID>
    </row>
  </rows>
</cf:grid>
```

The cftree tags

For applet and Flash format trees, ColdFusion inserts `cftree` controls in the XML as HTML embed objects in <![CDATA[sections in the tag body. The controls can be Java applets or in Flash SWF format. The `cf:tree` XML tag has two attributes: `format`, which can be Flash or Applet, and `id`.

For XML format trees, ColdFusion converts the CFML to XML in the following format:

```
cf:tree format="XML" id="treename>
  <metadata>
    <cftreeAttribute1>attributeValue</cftreeAttribute1>
    ...
  </metadata>
  <node cfml tree item attributes>
    <node //nested node with no children
      cfml tree item attributes />
    ...
  </node>
  ...
</cf:tree>
```

The following example shows a minimal tree with two nodes:

CFML

```
<cfform name="form2" Format="XML" >
<cftree name="tree1" hscroll="No" vscroll="No" format="xml"
  border="No">
  <cftreeitem value="Divisions">
  <cftreeitem value="Development"
    parent="Divisions" img="folder">
</cftree>
</cfform>
```

XML

The following code shows only the XML that is related to the tree appearance:

```
<cf:tree format="xml" id="tree1">
  <metadata>
    <fontWeight/>
    <align/>
    <lookAndFeel>windows</lookAndFeel>
    <delimiter>\</delimiter>
    <completePath>false</completePath>
    <border>false</border>
    <hScroll>false</hScroll>
    <vScroll>false</vScroll>
    <appendKey>true</appendKey>
    <highlightHref>true</highlightHref>
    <italic>false</italic>
    <bold>false</bold>
  </metadata>
  <node display="Divisions" expand="true" href="" img=""
    imgOpen="" parent="" path="Divisions" queryAsRoot="true"
          value="Divisions">
```

```
    <node display="Development" expand="true" href=""
      img="folder" imgOpen="" parent="Divisions"
      path="Divisions\Development" queryAsRoot="true"
            value="Development"/>
  </node>
</cf:tree>
```

The cfformgroup and cfformitem tags

All `cfformgroup` tags and all `cfformitem` tags, except `type="html"` and `type="text"`, generate `xf:group` elements. The following rules determine the element structure:

- The CFML tag `type` attribute determines the `xf:group appearance` attribute.
- ColdFusion MX converts `type` attribute values to all-lowercase characters.
- For `cfformgroup` tags only, the CFML `label` attribute determines the `xf:group label` attribute.
- All other CFML attributes are put in `cf:attribute` elements in a `xf:extension` element.
- The `cfformitem` tags generate an `xf:output` element with the body text in a <![CDATA[section.

The following example shows two `cformitem` tags, and the resulting XML:

CFML

```
<cfformitem name="text1" type="text" style="color:green">
  Please tell us a little about yourself and your thoughts.
</cfformitem>
<cfformitem type="hrule" height="3" width="200" testattribute="testvalue" />
```

XML

```
<xf:output><![CDATA[Please tell us a little about yourself and your
thoughts.]]>
  <xf:extension>
    <cf:attribute name="style">color:green</cf:attribute>
  </xf:extension>
</xf:output>
<xf:group appearance="hrule">
  <xf:extension>
    <cf:attribute name="width">200</cf:attribute>
    <cf:attribute name="height">3</cf:attribute>
    <cf:attribute name="testattribute">testvalue</cf:attribute>
  </xf:extension>
</xf:group>
```

Example: control element XML

The following code shows the XML for the input controls for the form shown in the figure in "About XML skinnable forms" on page 709. This code immediately follows the end of the `xf:model` element.

```
<xf:group appearance="horizontal">
  <xf:label>name</xf:label>
  <xf:extension/>
```

```
  <xf:input bind="firstname" id="firstname">
    <xf:label>First</xf:label>
    <xf:extension>
      <cf:attribute name="type">text</cf:attribute>
      <cf:attribute name="size">20</cf:attribute>
    </xf:extension>
  </xf:input>
  <xf:input bind="lastname" id="lastname">
    <xf:label>Last</xf:label>
    <xf:extension>
      <cf:attribute name="type">text</cf:attribute>
      <cf:attribute name="size">25</cf:attribute>
    </xf:extension>
  </xf:input>
</xf:group>
<xf:input bind="email" id="email">
  <xf:label>Email</xf:label>
  <xf:extension>
    <cf:attribute name="type">text</cf:attribute>
    <cf:attribute name="validation">email</cf:attribute>
  </xf:extension>
</xf:input>
<xf:output><![CDATA[<b>We value your input</b>.<br>
  <em>Please tell us a little about yourself and your thoughts.</em>]]>
  <xf:extension/>
</xf:output>
<xf:group appearance="vertical">
  <xf:extension/>
  <xf:select1 appearance="minimal" bind="satisfaction" id="satisfaction">
    <xf:label>Satisfaction</xf:label>
    <xf:extension>
      <cf:attribute name="type">select</cf:attribute>
      <cf:attribute name="style">width:200</cf:attribute>
    </xf:extension>
    <xf:choices>
      <xf:item>
        <xf:label>very satisfied</xf:label>
        <xf:value>very satisfied</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>somewhat satisfied</xf:label>
        <xf:value>somewhat satisfied</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>somewhat dissatisfied</xf:label>
        <xf:value>somewhat dissatisfied</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>very dissatisfied</xf:label>
        <xf:value>very dissatisfied</xf:value>
      </xf:item>
      <xf:item>
        <xf:label>no opinion</xf:label>
        <xf:value>no opinion</xf:value>
      </xf:item>
```

```
      </xf:choices>
    </xf:select1>
    <xf:textarea bind="thoughts" id="thoughts">
      <xf:label>Additional Comments</xf:label>
      <xf:extension>
        <cf:attribute name="type">textarea</cf:attribute>
        <cf:attribute name="rows">5</cf:attribute>
        <cf:attribute name="cols">40</cf:attribute>
      </xf:extension>
    </xf:textarea>
  </xf:group>
  <xf:group appearance="horizontal">
    <xf:extension/>
    <xf:submit id="submit" submission="comments">
      <xf:label>Tell Us</xf:label>
      <xf:extension>
        <cf:attribute name="type">submit</cf:attribute>
        <cf:attribute name="name">submit</cf:attribute>
      </xf:extension>
    </xf:submit>
    <xf:submit id="reset">
      <xf:label>Clear Fields</xf:label>
      <reset ev:event="DOMActivate"/>
      <xf:extension>
        <cf:attribute name="name">reset</cf:attribute>
      </xf:extension>
    </xf:submit>
  </xf:group>
```

Creating XSLT skins

This section provides information to help you create your own XSLT skins to process the XML that ColdFusion MX generates. It assumes that you are familiar with XSLT and CSS programming, and does not provide general information on writing XSLT transformations or CSS styles. It does provide information about the following areas:

- How ColdFusion passes form attribute values to the XML file
- How extend XSLT skins that ColdFusion MX provides as templates
- Basic techniques for extending the basic.xsl file to support additional `cfformgroup` and `cfformitem` tag `type` attributes
- How to extend the ColdFusion CSS files to enhance form appearance.

XSLT skin file locations

If you specify an XSLT skin by name and omit the .xsl suffix, ColdFusion MX looks for the file in the `cfform` script source directory and its subdirectories. You can specify the script source directory in your `cfform` tag `scriptsrc` attribute, and you can set a default location on the Settings page in the ColdFusion MX Administrator. When you install ColdFusion MX, the default location is set to /CFIDE/scripts/ (relative to the web root).

You can also use a relative or absolute file path, or a URL, to specify the XSLT skin location. ColdFusion MX uses the CFML page's directory as the root of relative paths. The following formats are valid:

Format	Location
<cfform format="xml" skin="basic">	Searches for XML/CSS in the default directory and its subdirectories.
<cfform format="xml" skin="c:\foo\bar\basic.xsl">	Uses the absolute path.
<cfform format="xml" skin="basic.xsl">	Searches in the current directory.
<cfform format="xml" skin="..\basic.xsl:">	Searches the parent of the current directory.
<cfform format="xml" skin="http://anywhereOnTheWeb/basic.xsl">	Uses the specified URL.

Note: Hosting companies might move the default skin location folder out of CFIDE; this lets them secure the CFIDE while giving site developers access to the files that you need for `cfform`.

Attribute and value passthrough

ColdFusion passes form tag attributes or attribute values that it does not specifically process directly to the XML, as follows:

- It converts `cfformitem` and `cfformgroup type` attributes to `xf:group` element `appearance` attributes.
- It passes the name and value of tag attributes that it does not recognize or process in `cf:attribute` elements.

This passthrough feature lets you create custom versions of any of the following items for your XSLT to process:

- The `cfformitem` types, such as rules, spacers, or other display elements
- The `cfgroup` types, such as divided boxes or tabbed dialog boxes
- The custom `cfinput` types, such as a custom year chooser element
- ColdFusion tag attributes, such as those used to control validation

Extending ColdFusion XSLT skins

ColdFusion MX 7 provides basic XSLT transforms that you can use as templates and extend for making your own skin. Each skin has a base XSL file, which include several utility XSL files. Utility file names start with an underscore (_), and the files are shared by multiple base skins. The following tables describes the XSL files, which are located in the *cf_webroot*\CFIDE\scripts\xsl directory:

File	Description
default.xsl	The default transform that ColdFusion MX uses if you do not specify a `skin` attribute for an XML format form. Identical to the basic.xsl file.
basic.xsl	A basic form format that arranges form elements using a table.

File	Description
basiccss.xsl	A basic form format that arranges form elements using HTML `div` and `span` tags.
colorname.xsl	A basic form format that arranges form elements using a table and applies a color scheme determined by the *colorname* to the form. Based on the basic.xsl file.
_cfformvalidation.xsl	Applies ColdFusion MX validation rules. Used by all skins.
_formelements.xsl	Transformation rules for form elements except for those defined using `cfformgroup` tags. Used by all skins
_group_*type*.xsl _group_*type*_table.xsl _group_*type*_css.xsl	Transformation rules for `cfformgroup` tags. The tag *`type`* attribute is part of the file name. Files with *table* in the name are used by basic.xsl and its derivatives. Files with *css* in the name are used by basiccss.xsl.

All skins support the same set of CFML tags and tag types, and do a relatively simple transformation from XML to HTML. For example, they do not support horizontal or vertical rules.

The ColdFusion MX skin XSL files have several features that you can use when designing and developing your own transformation. They do the following:

- Provide an overall structure and initial templates for implementing custom transformations.
- Show how you can handle the various elements in the ColdFusion-generated XML.
- Use a structure of included files that can form a template for your XSLT code.
- The base XSL files include a separate file, _cfformvalidation.xsl, with complete code for generating the hidden fields required for ColdFusion onServer validation and the JavaScript for performing ColdFusion onSubmit and onBlur validation. You can include this file without modification to do ColdFusion validation in your XSLT template, or you can change it to add other forms of validation or to change the validation rules.
- The base XSL files include files, that implement several form groups, laying out the child tags and applying a label to the group. These files can serve as templates for implementing additional form group types or you can expand them to provide more sophisticated horizontal and vertical form groups.
- You can add custom `cfformgroup` and `cfformitem type` attributes by including additional XSL files.

Extending basic.xsl cfformgroup and cfformitem support

The following procedure describes the steps for extending the basic.xsl file to support additional `cfformgroup` and `cfformitem` types. You can use similar procedures to extend other xsl files.

To add support for cfformgroup and cfformitem types to the basic.xsl:

1. Create an XSL file.
2. For each `type` attribute that you want to support, create an `xsl:template` element to do the formatting. The element's `match` attribute must have the following format:

   `match="xf:group[@appearance='`*`type_attribute_name`*`']"`

For example, to add a panel `cfformgroup` type, add an element with a start tag such as the following:

```
<xsl:template match="xf:group[@appearance='panel']">
```

3. Deploy your XSL file or files to the *cf_webroot*\CFIDE\scripts\xsl directory.
4. Add an include statement to the basic.xsl file at the end of the Supported groups section; for example, if you create a my_group_panel.xsl file to handle a panel `cfformgroup` type, your basic.xsl file would include the following lines:

```
<!-- include groups that will be supported for this skin-->
<xsl:include href="_group_vertical_table.xsl" />
<xsl:include href="_group_horizontal_table.xsl" />
<xsl:include href="_group_fieldset.xsl"/>
<xsl:include href="my_group_panel.xsl" />
```

Styling forms by extending the ColdFusion MX CSS files

Each ColdFusion MX skinnable form XSL file uses a corresponding CSS style sheet to specify the form style and layout characteristics. The following CSS files are located in the *cf_webroot*\CFIDE\scripts\css directory:

File	Description
basic_style.css default_style.css	Provides a plain style for ColdFusion XSL files that use table-based formatting. These files are identical and are used by the basic.xsl and default.xsl transforms. ColdFusion uses the default_style.css if you do not specify a skin in your `cfform` tag.
basic2_style.css	The basic_style with limited positioning changes for use with XSL files that have div-based formatting. Used by the basiccss.xsl transform.
css_layout.css	Style specifications for laying out forms that use div-based formatting. Used by the basiccss.xsl transform.
*colorname*_style.css	Used by the color-formatted ColdFusion MX skins. Defines the same classes as basic_style.css, with additional property specifications.

The ColdFusion XSL files and their corresponding CSS style sheets use classes extensively to format the form. The basic.xsl file, for example, has only one element style; all other styles are class-based. Although the CSS files contain specifications for all classes used in the XSL files, they do not always contain formatting information. The horizontal class definition in basic_style.css, which is used for horizontal form groups, for example, is empty.

You can enhance the style of XML skinnable forms without changing the XSL transform by enhancing the style sheets that ColdFusion MX provides.

CHAPTER 31
Creating Charts and Graphs

This chapter explains how to use the `cfchart` tag to display charts and graphs. It describes ways that you can chart data and the tools you need to create effective charts.

Contents

About charts 733
Creating a basic chart 734
Charting data 735
Controlling chart appearance 743
Creating charts: examples 751
Administering charts 756
Writing a chart to a variable 756
Linking charts to URLs 758

About charts

The ability to display data in a chart or graph can make data interpretation much easier. Rather than present a simple table of numeric data, you can display a bar, pie, line, or other applicable type of chart using colors, captions, and a two-dimensional or three-dimensional representation of your data.

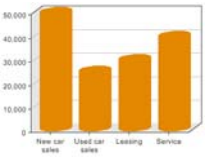
The `cfchart` tag, along with the `cfchartseries` and `cfchartdata` tags, provide many different chart types. The attributes to these tags let you customize your chart appearance.

You can create 11 types of charts in Macromedia ColdFusion MX 7 in two and three dimensions. The following figure shows a sample of each type of chart.

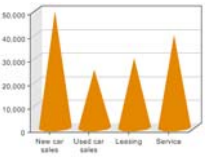
Note: In two dimensions, bar and cylinder charts appear the same, as do cone and pyramid charts.



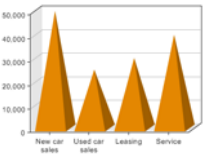
bar



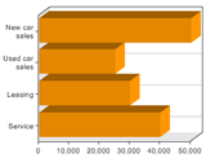
cylinder



cone



pyramid



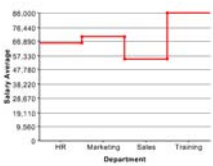
horizontal bar



line



curve



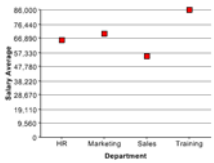
step



area



pie



scatter

Creating a basic chart

You can create a chart in either of the following ways:

- Using the `cfchart`, `cfchartseries`, and `cfchartdata` tags in a ColdFusion page.
- Using the chart wizard that is included with the ColdFusion Report Builder. For more information, see Chapter 32, "Creating Reports for Printing."

Creating a chart with ColdFusion tags

To create a chart with ColdFusion tags, you use the `cfchart` tag along with at least one `cfchartseries` tag. You can optionally include one or more `cfchartdata` tags within a `cfchartseries` tag. The following table describes these tags:

Tag	Description
`cfchart`	Specifies the container in which the chart appears. This container defines the height, width, background color, labels, fonts, and other characteristics of the chart. You must include at least one `cfchartseries` tag within the `cfchart` tag.
`cfchartseries`	Specifies a database query that supplies the data to the chart and one or more `cfchartdata` tags that specify individual data points. Specifies the chart type, colors for the chart, and other optional attributes.
`cfchartdata`	Optionally specifies an individual data point to the `cfchartseries` tag.

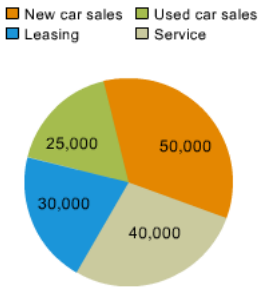
The following example shows an outline of the basic code that you use to create a chart:

```
<cfchart>
  <cfchartseries type="type">
       <cfchartdata item=”something” value="number">
  </cfchartseries>
</chart>
```

The following example displays a simple pie chart that illustrates four values:

```
<cfchart>
  <cfchartseries type="pie">
    <cfchartdata item="New car sales" value="50000">
    <cfchartdata item="Used car sales" value="25000">
    <cfchartdata item="Leasing" value="30000">
    <cfchartdata item="Service" value="40000">
  </cfchartseries>
</cfchart>
```

The following figure shows the resulting chart:



Creating a chart with the Report Builder wizard

The ColdFusion Report Builder includes a wizard that lets you create charts easily. The wizard lets you specify all of the chart characteristics that you can specify using the `cfchart`, `cfchartseries`, and `cfchartdata` tags. For information about using the Report Builder chart wizard, see Chapter 32, “Creating Reports for Printing,” on page 763.

Charting data

One of the most important considerations when you chart data is the way that you supply the data to the `cfchart` tag. You can supply data in the following ways:

- Specify individual data points using `cfchartdata` tags.
- Provide all the data in a single query using `cfchartseries` tags.
- Combine data from a query with additional data points from `cfchartdata` tags.
- Provide all the data in a report created with Report Builder. For more information, see Chapter 32, “Creating Reports for Printing,” on page 763.

Note: The `cfchart` tag charts numeric data only. As a result, you must convert any dates, times, or preformatted currency values, such as $3,000.53, to integers or real numbers.

Charting individual data points

When you chart individual data points, you specify each data point by inserting a `cfchartdata` tag in the `cfchartseries` tag body. For example, the following code creates a simple pie chart:

```
<cfchart>
  <cfchartseries type="pie">
    <cfchartdata item="New Vehicle Sales" value=500000>
    <cfchartdata item="Used Vehicle Sales" value=250000>
    <cfchartdata item="Leasing" value=300000>
    <cfchartdata item="Service" value=400000>
  </cfchartseries>
</cfchart>
```

This pie chart displays four types of revenue for a car dealership. Each `cfchartdata` tag specifies a department's income and a description for the legend.

Note: If two data points have the same item name, ColdFusion creates a graph of the value for the last one specified within the `cfchart` tag.

The `cfchartdata` tag lets you specify the following information about a data point:

Attribute	Description
value	The data value to be charted. This attribute is required.
item	(Optional) The description for this data point. The item appears on the horizontal axis of bar and line charts, on the vertical axis of horizontalbar charts, and in the legend of pie charts.

Charting a query

Each bar, dot, line, or slice of a chart represents data from one row/column coordinate in your result set. A related group of data is called a chart series.

Because each bar, dot, line, or slice represents the intersection of two axes, you must craft the query result set such that the row and column values have meaning when displayed in a chart. This often requires that you aggregate data in the query. You typically aggregate data in a query using one of the following:

- Specify a SQL aggregate function (SUM, AVG, MAX, and so on) using a GROUP BY clause in the SELECT statement.
- Use a Query of Queries.
- Retrieve data from a view, instead of a table.

When you chart a query, you specify the query name using the `query` attribute of the `cfchartseries` tag. For example, the code for a simple bar chart might be as follows:

```
<cfchart
    xAxisTitle="Department"
    yAxisTitle="Salary Average"
  >
  <cfchartseries
    type="bar"
    query="DataTable"
```

```
    valueColumn="AvgByDept"
    itemColumn="Dept_Name"
    />
</cfchart>
```

This example displays the values in the AvgByDept column of the DataTable query. It displays the Dept_Name column value as the item label by each bar.

The following table lists the attributes of the `cfchartseries` tag that you use when working with queries:

Attribute	Description
`query`	The query that contains the data. You must also specify the `valueColumn` and `itemColumn`.
`valueColumn`	The query column that contains the values to be charted.
`itemColumn`	The query column that contains the description for this data point. The item normally appears on the horizontal axis of bar and line charts, on the vertical axis of horizontalbar charts, and in the legend in pie charts.

Charting a query of queries

In addition to charting the results of a query, you can also chart the results of a queries of queries. For more information about using query of queries, see Chapter 22, "Using Query of Queries," on page 489. Query of queries provides significant power in generating the data for the chart. For example, you can use aggregating functions such as SUM, AVG, and GROUP BY to create a query of queries with statistical data based on a raw database query. For more information, see Chapter 22, "Using Query of Queries," on page 489.

You can also take advantage of the ability to dynamically reference and modify query data. For example, you can loop through the entries in a query column and reformat the data to show whole dollar values.

The example in the following procedure analyzes the salary data in the cfdocexamples database using a query of queries, and displays the data as a bar chart.

To chart a query of queries:

1. Create a new ColdFusion page with the following content:

```
<!-- Get the raw data from the database. -->
<cfquery name="GetSalaries" datasource="cfdocexamples">
  SELECT Departmt.Dept_Name,
    Employee.Salary
  FROM Departmt, Employee
  WHERE Departmt.Dept_ID = Employee.Dept_ID
</cfquery>

<!-- Generate a query with statistical data for each department. -->
<cfquery dbtype = "query" name = "DeptSalaries">
  SELECT
    Dept_Name,
    AVG(Salary) AS AvgByDept
  FROM GetSalaries
```

```
  GROUP BY Dept_Name
</cfquery>

<!--- Reformat the generated numbers to show only thousands. --->
<cfloop index="i" from="1" to="#DeptSalaries.RecordCount#">
  <cfset DeptSalaries.AvgByDept[i]=Round(DeptSalaries.AvgByDept[i]/
  1000)*1000>
</cfloop>

<html>
<head>
  <title>Employee Salary Analysis</title>
</head>

<body>
<h1>Employee Salary Analysis</h1>

<!--- Bar chart, from DeptSalaries Query of Queries. --->
<cfchart
     xAxisTitle="Department"
     yAxisTitle="Salary Average"
     font="Arial"
     gridlines=6
     showXGridlines="yes"
     showYGridlines="yes"
     showborder="yes"
     show3d="yes"
  >

  <cfchartseries
    type="bar"
    query="DeptSalaries"
    valueColumn="AvgByDept"
    itemColumn="Dept_Name"
    seriesColor="olive"
    paintStyle="plain"
  />
</cfchart>

<br>
</body>
</html>
```

2. Save the page as chartdata.cfm in the myapps directory under the web root directory. For example, the directory path in Windows might be C:\Inetpub\wwwroot\myapps.

3. Return to your browser and enter the following URL to view the chartdata.cfm page:

   http://localhost/myapps/chartdata.cfm

   The following figure appears:



Note: If a query contains two rows with the same value for the `itemColumn` attribute, ColdFusion graphs the last row in the query for that value. For the preceding example, if the query contains two rows for the Sales department, ColdFusion graphs the value for the last row in the query for Sales.

Reviewing the code

The following table describes the code and its function:

Code	Description
`<cfquery name="GetSalaries" datasource="cfdocexamples"> SELECT Departmt.Dept_Name, Employee.Salary FROM Departmt, Employee WHERE Departmt.Dept_ID = Employee.Dept_ID </cfquery>`	Query the cfdocexamples database to get the Dept_Name and Salary for each employee. Because the Dept_Name is in the Departmt table and the Salary is in the Employee table, you need a table join in the WHERE clause. You can use the raw results of this query elsewhere on the page.
`<cfquery dbtype = "query" name = "DeptSalaries"> SELECT Dept_Name, AVG(Salary) AS AvgByDept FROM GetSalaries GROUP BY Dept_Name </cfquery>`	Generate a new query from the GetSalaries query. Use the AVG aggregating function to get statistical data on the employees. Use the GROUP BY statement to ensure that there is only one row for each department.

Code	Description
<cfloop index="i" from="1" to="#DeptSalaries.RecordCount#"> <cfset DeptSalaries.AvgByDept[i]= Round(DeptSalaries.AvgByDept[i] /1000)*1000> </cfloop>	Loop through all the rows in the DeptSalaries query and round the salary data to the nearest thousand. This loop uses the RecordCount query variable to get the number of rows, and directly changes the contents of the query object.
<cfchart xAxisTitle="Department" yAxisTitle="Salary Average" font="Arial" gridlines=6 showXGridlines="yes" showYGridlines="yes" showborder="yes" show3d="yes" > <cfchartseries type="bar" query="DeptSalaries" valueColumn="AvgByDept" itemColumn="Dept_Name" seriesColor="olive" paintStyle="plain"/> </cfchart>	Create a bar chart using the data from the AvgByDept column of the DeptSalaries query. Label the bars with the department names.

You can also rewrite this example to use the `cfoutput` and `cfchartdata` tags within the `cfchartseries` tag, instead of using the loop, to round the salary data, as the following code shows:

```
<cfchartseries
  type="bar"
  seriesColor="olive"
  paintStyle="plain">

  <cfoutput query="deptSalaries">
    <cfchartdata item="#dept_name#" value=#Round(AvgByDept/1000)*1000#>
  </cfoutput>

</cfchartseries>
```

Combining a query and data points

To chart data from both query and individual data values, you specify the query name and related attributes in the `cfchartseries` tag, and provide additional data points using the `cfchartdata` tag.

ColdFusion displays the chart data specified by a `cfchartdata` tag before the data from a query; for example, to the left on a bar chart. You can use the `sortXAxis` attribute of the `cfchart` tag to sort data alphabetically along the x-axis.

One use of combining queries and data points is to provide data that is missing from the database; for example, to provide the data for one department if the data for that department is missing. The example in the following procedure adds data for the Facilities and Documentation departments to the salary data obtained from the query shown in the previous section:

To chart a query and individual data points:

1. Open the chartdata.cfm file in your editor.
2. Edit the `cfchart` tag so that it appears as follows:

```
<cfchart chartwidth="600">
  <cfchartseries
    type="bar"
    query="DeptSalaries"
    itemColumn ="Dept_Name"
    valueColumn="AvgByDept"
    >
    <cfchartdata item="Facilities" value="35000">
    <cfchartdata item="Documentation" value="45000">
  </cfchartseries>
</cfchart>
```

3. Save the page as chartqueryanddata.cfm in the myapps directory under the web root directory. For example, the directory path in Windows might be C:\Inetpub\wwwroot\myapps.
4. Return to your browser and enter the following URL to view the chartqueryanddata.cfm page:

   http://localhost/myapps/chartqueryanddata.cfm

Charting multiple data collections

Sometimes, you might have more than one series of data to display on a single chart, or you want to compare two sets of data on the same chart. In some cases, you might want to use different charting types on the same chart. For example, you might want to include a line chart on a bar chart.

To combine multiple data series into a single chart, insert multiple `cfchartseries` tags within a single `cfchart` tag. You control how the multiple data collections are charted using the `seriesPlacement` attribute of the `cfchart` tag. Using this attribute, you can specify the following options:

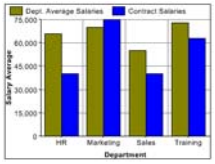
default Let ColdFusion determine the best method for combining the data.

cluster Place corresponding chart elements from each series next to each other.

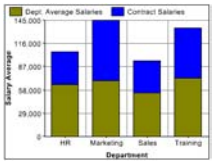
stacked Combine the corresponding elements of each series.

percent Show the elements of each series as a percentage of the total of all corresponding elements.

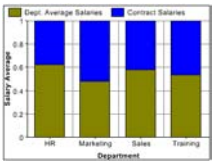
The following figure shows these options for combining two bar charts:



Cluster

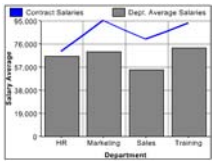


Stacked



Percent

You can also combine chart types. The following is a combination bar and line chart:



The only chart type that you cannot mix with others is the pie chart. If you define one of the data series to use a pie chart, no other chart will appear.

The example in the following procedure creates the chart in the previous figure, which shows a bar chart with a line chart added to it. In this example, you chart the salary of permanent employees (bar) against contract employees (line).

Note: The layering of multiple series depends on the order that you specify the `cfchartseries` tags. For example, if you specify a bar chart first and a line chart second, the bar chart appears in front of the line chart in the final chart.

To create a combination bar chart and line chart:

1. Open the chartdata.cfm file in your editor.
2. Edit the `cfchart` tag so that it appears as follows:

```
<cfchart
     backgroundColor="white"
     xAxisTitle="Department"
     yAxisTitle="Salary Average"
     font="Arial"
     gridlines=6
     showXGridlines="yes"
     showYGridlines="yes"
     showborder="yes"
   >
   <cfchartseries
     type="line"
     seriesColor="blue"
     paintStyle="plain"
     seriesLabel="Contract Salaries"
     >
       <cfchartdata item="HR" value=70000>
       <cfchartdata item="Marketing" value=95000>
       <cfchartdata item="Sales" value=80000>
       <cfchartdata item="Training" value=93000>
```

```
    </cfchartseries>
  <cfchartseries
    type="bar"
    query="DeptSalaries"
    valueColumn="AvgByDept"
    itemColumn="Dept_Name"
    seriesColor="gray"
    paintStyle="plain"
    seriesLabel="Dept. Average Salaries"
    />

</cfchart>
```

3. Save the file as chart2queries.cfm in the myapps directory under the web root directory.
4. Return to your browser and view the chart2queries.cfm page.

Controlling chart appearance

You can control the appearance of charts by doing any of the following:

- Using the default chart styles included with ColdFusion
- Using the attributes of the cfchart and cfchartseries tags
- Creating your own chart styles

Using the default chart styles included with ColdFusion

ColdFusion supplies a number of chart styles. These styles include:

- beige
- blue
- default
- red
- silver
- yellow

To use any of these styles, specify the style using the `style` attribute of the `cfchart` tag. The following example illustrates using the `beige` style:

```
<cfchart style="beige">
  <cfchartseries type="pie">
    <cfchartdata item="New car sales" value="50000">
    <cfchartdata item="Used car sales" value="25000">
    <cfchartdata item="Leasing" value="30000">
    <cfchartdata item="Service" value="40000">
  </cfchartseries>
</cfchart>
```

Using the attributes of the cfchart and cfchartseries tags

You can specify the appearance of charts by using the attributes of the `cfchart` and `cfchartseries` tags.

You can optionally specify the following characteristics to the `cfchart` tag on the types of charts indicated:

Chart characteristic	Attributes used	Description	Chart type
File type	`format`	Whether to send the chart to the user as a JPEG, PNG, or SWF file. The SWF file is the default format.	All
Size	`chartWidth` `chartHeight`	The width and height, in pixels, of the chart. This size defines the entire chart area, including the legend and background area around the chart. The default height is 240 pixels; the default width is 320 pixels.	All

Color	`foregroundColor`
	`dataBackgroundColor`
	`backgroundColor`
		The colors used for foreground and background objects.
		The default foreground color is black; the default background colors are white.
		You can specify 16 color names, use any valid HTML color format, or specify an 8-digit hexadecimal value to specify the RGB value and transparency. If you use numeric format, you must use double number signs; for example, blue or ##FF33CC. To specify the color and transparency, use the format ##*xx*FF33CC, where xx indicates the transparency. Opaque is indicated by the value FF; transparent is indicated by the value 00. For the complete list of colors, see *Configuring and Administering ColdFusion MX*.
			All

Chart characteristic	Attributes used	Description	Chart type
Labels	font fontSize fontBold frontItalic labelFormat xAxisTitle yAxisTitle	The font attribute specifies the font for all text. The default value is Arial. If you are using a double-byte character set on UNIX, or using a double-byte character set in Windows with a file type of Flash, you must specify ArialUnicodeMs as the font. ***Note:*** If a chart attempts to use a font that is not installed on the ColdFusion server, it uses a different font that is available. Also, if you do not specify the font, characters that are not ASCII, such as Japanese, Chinese, Korean and so on, may not display properly. The fontSize specifies an Integer font size used for all text. The default value is 11. The fontBold attribute specifies to display all text as bold. The default value is no. The fontItalic attribute specifies to display all text as italic. The default value is no. The labelFormat attribute specifies the format of the y-axis labels, number, currency, percent, or date. The default value is number. The xAxisTitle and yAxisTitle attributes specify the title for each axis.	All
Border	showBorder	Use the showBorder attribute to draw a border around the chart. The border color is specified by the foregroundColor attribute. The default value is no.	All
Grid lines	showXGridlines showYGridlines gridLines	Use the showXGridlines and showYGridlines attributes to display x-axis and y-axis grid lines. The default value No for x-axis gridlines, and Yes for y-axis gridlines. The gridLines attribute specifies the total number of grid lines on the value axis, including the axis itself. The value of each grid line appears along the value axis. The cfchart tag displays horizontal grid lines only. A value of 0 (the default) means no grid lines.	Area Bar Cone Curve Cylinder Horizontalbar Line Pyramid Scatter Step
Slice style	pieSliceStyle	Displays the pie chart as solid or sliced. The default value is sliced.	Pie

Chart characteristic	Attributes used	Description	Chart type
Markers	`showMarkers` `markerSize`	The `showMarkers` attribute displays markers at the data points for two dimensional line, curve, and scatter charts. The default value is `yes`. The `markerSize` attribute specifies an integer number of pixels for the marker size. ColdFusion determines the default value.	All
Value axis	`scaleFrom` `scaleTo`	The minimum and maximum points on the data axis. By default, the minimum is `0` or the lowest negative chart data value, and the maximum is the largest data value. ***Note:*** If you specify a `scaleFrom` or `scaleTo` attribute that would result in cropping the chart, `cfchart` uses a value that shows the entire chart without cropping.	Area Bar Cone Curve Cylinder Horizontalbar Line Pyramid Scatter Step
Axis type	`XAxisType` `sortXAxis`	Whether the x-axis corresponds to a numeric scale or identifies different categories, and how to sort the items on the axis. If the `XAxisType` attribute value is `scale`, the x-axis is numeric. All `cfchartdata item` attribute values must be numeric, and the axis is automatically sorted numerically. The `scale` value lets you create graphs of numeric relationships, such as population against age. If the attribute value is `category` (the default), the axis indicates the data category. The `sortXAxis` attribute determines the order of items when you specify the `cfchartdata item` attribute, whose values are treated as text. By default, the items are displayed in the order in which they are entered in the first chart series.	Area Bar Cone Curve Cylinder Horizontalbar Line Pyramid Scatter Step

3D appearance	`show3D`
	`xOffset`
	`yOffset`
		The `show3D` attribute displays the chart in three dimensions. The default value is `No`.
		The `xOffset` and `yOffset` attributes specify the amount to which the chart should be rotated on a horizontal axis (`xOffset`) or vertical axis (`yOffset`). The value `0` is flat (no rotation), `-1` and `1` are for a full 90 degree rotation left (`-1`) or right (`1`). The default value is `1`.
			All

Chart characteristic	Attributes used	Description	Chart type
Multiple series	showLegend seriesPlacement	The showLegend attribute lets you display the chart's legend when the chart contains more than one series of data. The default value is Yes. The seriesPlacement attribute specifies the location of each series relative to the others. By default, ColdFusion determines the best placement based on the graph type of each series.	All
Tips	tipStyle tipBGColor	The tipStyle attribute lets you display a small pop-up window that shows information about the chart element pointed to by the mouse pointer. Options are none, mousedown, or mouseover. The default value is mouseover. The tipBGColor attribute specifies the background color of the tip window for Flash format only. The default value is white.	All

You can also use the cfchartseries tag to specify attributes of chart appearance. The following table describes these attributes:

Chart characteristic	Attributes used	Description	Chart type
Multiple series	seriesLabel seriesColor	The seriesLabel attribute specifies the text that displays for the series label. The seriesColor attribute specifies a single color of the bar, line, pyramid, and so on. For pie charts, this is the first slice's color. Subsequent slices are automatically colored based on the specified initial color, or use the colorList attribute.	All
Paint	paintStyle	Specifies the way color is applied to a data series. You can specify solid color, buttonized look, linear gradient fill with a light center and darker outer edge, and gradient fill on lighter version of color. The default value is solid.	All

Chart characteristic	Attributes used	Description	Chart type
Data point colors	colorList	A comma-separated list of colors to use for each data point for bar, pyramid, area, horizontalbar, cone, cylinder, step, and pie charts. You can specify 16 color names, use any valid HTML color format, or specify an 8-digit hexadecimal value to specify the RGB value and transparency. If you use numeric format, you must use double number signs; for example, blue or ##FF33CC. To specify the color and transparency, use the format ##*xx*FF33CC, where xx indicates the transparency. Opaque is indicated by the value FF; transparent is indicated by the value 00. For the complete list of colors, see *Configuring and Administering ColdFusion MX.* If you specify fewer colors than data points, the colors repeat. If you specify more colors than data points, the extra colors are not used.	Pie
Data markers	markerStyle	Specifies the shape used to mark the data point. Shapes include `circle`, `diamond`, `letterx`, `mcross`, `rcross`, `rectangle`, `snow`, and `triangle`. Supported for two-dimensional charts. The default value is `rectangle`.	Curve Line Scatter
Labels	dataLabelStyle	Specifies the way in which the color is applied to the item in the series Styles include `None`, `Value`, `Rowlabel`, `Columnlabel`, and `Pattern`.	All

Creating your own chart styles

You can create your own chart styles by doing either of the following:

- Modifying the chart style XML files
- Using WebCharts3D to create chart styles

Modifying the chart style XML files

You can modify the chart styles included with ColdFusion to create your own chart styles. The files that contain the style information are XML files located in the *cf_root*\charting\styles directory. You should only modify attributes specified in the file. To specify additional attributes, follow the instructions in the section "Using WebCharts3D to create chart styles" on page 749.

Note: There are two XML files for each default chart style. For example, the beige style for pie charts is defined in the beige_pie.xml file; the beige style for all other types of charts is defined in the beige.xml file.

To modify a chart style XML file:

1. Open the XML file that you want to modify, for example beige.xml.
2. Modify the file contents.
3. Save the file with a different name; for example myBeige.xml.

Using WebCharts3D to create chart styles

ColdFusion MX 7 includes the WebCharts3D utility, which you can use to create chart style files.

To create your own chart style file with the WebCharts3D utility:

1. Start WebCharts3D by double-clicking the webcharts.bat file in the CFusionMX7\charting directory.
2. (Optional) Open an existing chart.
3. Make the changes you want to the chart's appearance.

   Note: To use the chart style file in the `cfchart` tag, you can only make the modifications indicated in the table that follows this procedure.

4. Click the XML style tab.
5. Click the Save button in the bottom right corner.
6. Specify the name of the file; for example, mystyle.xml.
7. Specify the directory in which you want to save the chart style file.

   Note: ColdFusion uses the same rules to look for the chart style XML files as it does for files included using the `cfinclude` tag. For more information, see `cfinclude`.

8. Click Save.

The following table lists the attributes of the `cfchart` and `cfchartseries` tags and the associated WebCharts3D commands:

Attribute	WebCharts3D command
`chartHeight`	Drag the chart by handles.
`chartWidth`	Drag the chart by handles.
`dataBackgroundColor`	Background: minColor (type must be PlainColor)
`font`	font: Family (specify only supported fonts.)
`fontBold`	font: check Bold
`fontItalic`	font: check Italic
`fontSize`	font: Size
`foregroundColor`	foreground
`gridlines`	XAxis: labelcount

Attribute	WebCharts3D command
labelFormat	YAxis: LabelFormat: Number | Percent| Currency | Datetime
markerSize	Elements: markerSize
pieSliceStyle	style: solid | slice
rotated	Type Frame chart: Elements: Shape:
scaleFrom	Yaxis: isAbsolute; scaleMin(int)
scaleTo	Yaxis: isAbsolute; scaleMax(int)
seriesPlacement	Elements: place
show3D	is3D
showBorder	Decoration: style (none or simple)?
showLegend	Legend: isVisible
showMarkers	Elements: showMarkers
showXGridlines	Frame: isVGridVisible
showYGridlines	Frame: isHGridVisible
tipbgColor	Popup: background
tipStyle	Popup: showOn: MouseOver | MouseDown | Disabled
url	Elements: action | Series: action
xAxisTitle	Xaxis: TitleStyle: text (enter text)
xAxisType	xAxis: type: (category or scale)
xOffset	Frame: xDepth
yAxisTitle	Yaxis: TitleStyle: text (enter text)
yAxisType	Currently has no effect.
yOffset	Frame: yDepth

The following table lists the attributes of the `cfchartseries` tag and the associated WebCharts3D commands:

Attribute	WebCharts3D command
colorlist	Elements: series: Paint: color
markerStyle	Elements: series: Marker type: Rectangle | Triangle | Diamond | Circle | Letter | MCROSS | Snow | RCROSS
paintStyle	Paint: paint: Plain | Shade | Light
seriesColor	Elements: series: Paint: color

seriesLabel	Elements: series:
type	Type: Pie chart | Type Frame chart: Elements: Shape: Bar | Line | Pyramid | Area | Curve | Step | Scatter | Cone | Cylinder | Horizontalbar |

Creating charts: examples

This section contains examples of different types of charts.

Creating a bar chart

The example in the following procedure adds a title to the bar chart, specifies that the chart is three dimensional, adds grid lines, sets the minimum and maximum y-axis values, and uses a custom set of colors.

To enhance the bar chart:

1. Open the chartdata.cfm file in your editor.
2. Edit the `cfchart` tag so that it appears as follows:

```
<!--- Bar chart, from Query of Queries --->
<cfchart
     scaleFrom=40000
     scaleTo=100000
     font="arial"
     fontSize=16
     gridLines=4
     show3D="yes"
     foregroundcolor="##000066"
     databackgroundcolor="##FFFFCC"
     chartwidth="450"
   >

   <cfchartseries
     type="bar"
     query="DeptSalaries"
     valueColumn="AvgByDept"
     itemColumn="Dept_Name"
     seriescolor="##33CC99"
     paintstyle="shade"
     />

</cfchart>
```

3. Save the file as chartdatastyle1.cfm.
4. View the chartdatastyle1.cfm page in your browser.

Reviewing the code

The following table describes the code in the preceding example.

Code	Description
`scaleFrom=40000`	Set the minimum value of the vertical axis to 40000.
`scaleTo=100000`	Set the maximum value of the vertical axis to 100000. The minimum value is the default, `0`.
`font="arial"`	Displays text using the Arial font.
`fontSize=16`	Makes the point size of the labels 16 points.
`gridLines = 4`	Displays four grid lines between the top and bottom of the chart.
`show3D = "yes"`	Shows the chart in 3D.
`foregroundcolor="##000066"`	Sets the color of the text, gridlines, and labels.
`databackgroundcolor="##FFFFCC"`	Sets the color of the background behind the bars.
`seriescolor="##33CC99"`	Sets the color of the bars.
`paintstyle="shade"`	Sets the paint display style.

Creating a pie chart

The example in the following procedure adds a pie chart to the page.

To create a pie chart:

1. Open the chartdata.cfm file in your editor.
2. Edit the DeptSalaries query and the `cfloop` code so that it appears as follows:

```
<!--- A query to get statistical data for each department. --->
<cfquery dbtype = "query" name = "DeptSalaries">
  SELECT
    Dept_Name,
    SUM(Salary) AS SumByDept,
    AVG(Salary) AS AvgByDept
  FROM GetSalaries
  GROUP BY Dept_Name
</cfquery>

<!--- Reformat the generated numbers to show only thousands. --->
<cfloop index="i" from="1" to="#DeptSalaries.RecordCount#">
  <cfset DeptSalaries.SumByDept[i]=Round(DeptSalaries.SumByDept[i]/
  1000)*1000>
  <cfset DeptSalaries.AvgByDept[i]=Round(DeptSalaries.AvgByDept[i]/
  1000)*1000>
</cfloop>
```

3. Add the following `cfchart` tag:

```
<!--- Pie chart, from DeptSalaries Query of Queries. --->
<cfchart
    tipStyle="mousedown"
    font="Times"
```

```
      fontsize=14
      fontBold="yes"
      backgroundColor = "##CCFFFF"
      show3D="yes"
      >

    <cfchartseries
      type="pie"
      query="DeptSalaries"
      valueColumn="SumByDept"
      itemColumn="Dept_Name"
      colorlist="##6666FF,##66FF66,##FF6666,##66CCCC"
      />
  </cfchart>
  <br>
```

4. Save the file as chartdatapie1.cfm.
5. View the chartdatapie1.cfm page in your browser:

Reviewing the code

The following table describes the code and its function:

Code	Description
`SUM(Salary) AS SumByDept,`	In the DeptSalaries query, add a SUM aggregation function to get the sum of all salaries per department.
`<cfset DeptSalaries.SumByDept[i]= Round(DeptSalaries.SumByDept[i]/ 1000)*1000>`	In the `cfloop` tag, round the salary sums to the nearest thousand.
`<cfchart tipStyle="mousedown" font="Times" fontBold="yes" backgroundColor = "##CCFFFF" show3D="yes" >`	Show a tip only when a user clicks on the chart, display text in Times bold font, set the background color to light blue, and display the chart in three dimensions.
`<cfchartseries type="pie" query="DeptSalaries" valueColumn="SumByDept" itemColumn="Dept_Name" colorlist= "##6666FF,##66FF66,##FF6666,##66CCCC" />`	Create a pie chart using the SumByDept salary sum values from the DeptSalaries query. Use the contents of the Dept_Name column for the item labels displayed in the chart legend. Get the pie slice colors from a custom list, which uses hexadecimal color numbers. The double number signs prevent ColdFusion from trying to interpret the color data as variable names.

Creating an area chart

The example in the following procedure adds an area chart to the salaries analysis page. The chart shows the average salary by start date to the salaries analysis page. It shows the use of a second query of queries to generate a new analysis of the raw data from the GetSalaries query. It also shows the use of additional `cfchart` attributes.

To create an area chart:

1. Open the chartdata.cfm file in your editor.
2. Edit the GetSalaries query so that it appears as follows:

```
<!-- Get the raw data from the database. -->
<cfquery name="GetSalaries" datasource="cfdocexamples">
  SELECT Departmt.Dept_Name,
    Employee.StartDate,
    Employee.Salary
  FROM Departmt, Employee
  WHERE Departmt.Dept_ID = Employee.Dept_ID
</cfquery>
```

3. Add the following code before the `html` tag:

```
<!--- Convert start date to start year. --->
<!--- You must convert the date to a number for the query to work --->
<cfloop index="i" from="1" to="#GetSalaries.RecordCount#">
<cfset GetSalaries.StartDate[i]=NumberFormat(DatePart("yyyy",
  GetSalaries.StartDate[i]) ,9999)>
</cfloop>

<!--- Query of Queries for average salary by start year. --->
<cfquery dbtype = "query" name = "HireSalaries">
  SELECT
    StartDate,
    AVG(Salary) AS AvgByStart
  FROM GetSalaries
  GROUP BY StartDate
</cfquery>

<!--- Round average salaries to thousands. --->
<cfloop index="i" from="1" to="#HireSalaries.RecordCount#">
  <cfset HireSalaries.AvgByStart[i]=Round(HireSalaries.AvgByStart[i]/
  1000)*1000>
</cfloop>
```

4. Add the following `cfchart` tag before the end of the `body` tag block:

```
<!--- Area-style Line chart, from HireSalaries Query of Queries. --->
<cfchart
     chartWidth=400
     BackgroundColor="##FFFF00"
     show3D="yes"
  >
  <cfchartseries
    type="area"
    query="HireSalaries"
    valueColumn="AvgByStart"
    itemColumn="StartDate"
   />
</cfchart>
<br>
```

5. Save the page.
6. View the chartdata.cfm page in your browser.

Reviewing the code

The following table describes the code and its function:

Code	Description
`Employee.StartDate,`	Add the employee start date to the data in the GetSalaries query.
`<cfloop index="i" from="1" to="#GetSalaries.RecordCount#"> <cfset GetSalaries.StartDate[i]= NumberFormat(DatePart("yyyy", GetSalaries.StartDate[i]) ,9999)> </cfloop>`	Use a `cfloop` tag to extract the year of hire from each employee's hire data, and convert the result to a four-digit number.
`<cfquery dbtype = "query" name = "HireSalaries"> SELECT StartDate, AVG(Salary) AS AvgByStart FROM GetSalaries GROUP BY StartDate </cfquery>`	Create a second query from the GetSalaries query. This query contains the average salary for each start year.
`<cfloop index="i" from="1" to="#HireSalaries.RecordCount#"> <cfset HireSalaries.AvgByStart[i] =Round(HireSalaries.AvgByStart[i] /1000)*1000> </cfloop>`	Round the salaries to the nearest thousand.
`<cfchart chartWidth=400 BackgroundColor="##FFFF00" show3D="yes" > <cfchartseries type="area" query="HireSalaries" valueColumn="AvgByStart" itemColumn="StartDate" /> </cfchart>`	Create a line chart using the HireSalaries query. Chart the average salaries against the start date. Limit the chart width to 400 pixels, show the chart in three dimensions, and set the background color to white.

Setting curve chart characteristics

Curves charts use the attributes already discussed. However, you should be aware that curve charts require a large amount of processing to render. For fastest performance, create them offline, write them to a file or variable, and then reference them in your application pages. For information on creating offline charts, see "Writing a chart to a variable" on page 756.

Administering charts

Use the ColdFusion MX Administrator to administer charts. In the Administrator, you can choose to save cached charts in memory or to disk. You can also specify the number of charts to cache, the number of charting threads, and the disk file for caching images to disk.

ColdFusion MX 7 caches charts as they are created. In that way, repeated requests of the same chart load the chart from the cache rather than having ColdFusion render the chart over and over again.

Note: You do not have to perform any special coding to reference a cached chart. Whenever you use the `cfchart` tag, ColdFusion MX inspects the cache to see if the chart has already been rendered. If so, ColdFusion MX loads the chart from the cache.

The following table describes the settings for the ColdFusion MX 7 charting and graphing engine:

Option	Description
Cache Type	Sets the cache type. Charts can be cached in memory or to disk. Caching in memory is faster, but more memory intensive.
Maximum number of images in cache	Specifies the maximum number of charts to store in the cache. When the limit is reached, the oldest chart in the cache is deleted to make room for a new one. The maximum number of charts you can store in the cache is 250.
Max number of charting threads	Specifies the maximum number of chart requests that can be processed concurrently. The minimum number is 1 and the maximum is 5. Higher numbers are more memory-intensive.
Disk cache location	When caching to disk, specifies the directory in which to store the generated charts.

Writing a chart to a variable

In some cases, your application might have charts that are static or charts that, because of the nature of the data input, take a long time to render. In this scenario, you can create a chart and write it to a variable.

Once written to a variable, other ColdFusion pages can access the variable to display the chart, or you can write the variable to disk to save the chart to a file. This lets you create or update charts only as needed, rather than every time someone requests a page that contains a chart.

You use the name attribute of the `cfchart` tag to write a chart to a variable. If you specify the name attribute, the chart is not rendered in the browser but is written to the variable.

You can save the chart as a Flash SWF file, or as a JPEG or PNG image file. If you save the image as a SWF file, you can pass the variable back to a Flash client using ColdFusion Flash Remoting. For more information, see Chapter 33, "Using the Flash Remoting Service," on page 807.

Note: If you write the chart to a JPEG or PNG file, mouseover tips and URLs embedded in the chart for data drill-down do not work when you redisplay the image from the file. However, if you save the image as a SWF file, both tips and drill-down URLs work. For more information on data drill-down, see "Linking charts to URLs" on page 758.

To write a chart to a variable and a file:

1. Create a new ColdFusion page with the following content:

```
<cfchart name="myChart" format="jpg">
  <cfchartseries type="pie">
    <cfchartdata item="New Vehicle Sales" value=500000>
    <cfchartdata item="Used Vehicle Sales" value=250000>
    <cfchartdata item="Leasing" value=300000>
    <cfchartdata item="Service" value=400000>
  </cfchartseries>
</cfchart>

<cffile
  action="WRITE"
  file="c:\inetpub\wwwroot\charts\vehicle.jpg"
  output="#myChart#">
<img src="/charts/vehicle.jpg" height=240 width=320>
```

2. Save the page as chartToFile.cfm in myapps under the web root directory.
3. View the chartToFile.cfm page in your browser.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description

`<cfchart`
`  name="myChart"`
`  format="jpg">`
	Define a chart written to the myChart variable using the JPEG format.

`<cffile`
`  action="WRITE"`
`  file=`
 	
`  "c:\inetpub\wwwroot\charts\vehicle.jpg"`
`  output="#myChart#">`
	Use the `cffile` tag to write the chart to a file.

`<img`
`  src="/chartsvehicle.jpg"`
`  height=240`
`  width=320>`
	Use the HTML `img` tag to display the chart.

Linking charts to URLs

ColdFusion provides a data drill-down capability with charts. This means that you can click the data and the legend areas of a chart to request a URL. For example, if you have a pie chart and want a user to be able to select a pie wedge for more information, you can build that functionality into your chart.

You use the `url` attribute of the `cfchart` tag to specify the URL to open when a user clicks anywhere on the chart. For example, the following code defines a chart that opens the page moreinfo.cfm when a user clicks on the chart:

```
<cfchart
  xAxisTitle="Department"
  yAxisTitle="Salary Average"
  url="moreinfo.cfm"
  >

    <cfchartseries
      seriesLabel="Department Salaries"
    ...
    />

</cfchart>
```

You can use the following variables in the `url` attribute to pass additional information to the target page:

- `$VALUE$` The value of the selected item, or an empty string
- `$ITEMLABEL$` The label of the selected item, or an empty string
- `$SERIESLABEL$` The label of the selected series, or an empty string

For example, to let users click on the graph to open the page moreinfo.cfm, and pass all three values to the page, you use the following `url`:

```
url="moreinfo.cfm?Series=$SERIESLABEL$&Item=$ITEMLABEL$&Value=$VALUE$"
```

The variables are not enclosed in number signs like ordinary ColdFusion variables. They are enclosed in dollar signs. If you click on a chart that uses this `url` attribute value, it could generate a URL in the following form:

```
http://localhost:8500/tests/charts/moreinfo.cfm?
  Series=Department%20Salaries&Item=Training&Value=86000
```

You can also use JavaScript in the URL to execute client-side scripts. For an example, see "Linking to JavaScript from a pie chart" on page 761.

Dynamically linking from a pie chart

In the following example, when you click a pie wedge, ColdFusion displays a table that contains the detailed salary information for the department represented by the wedge. The example is divided into two parts: creating the detail page and making the pie chart dynamic.

Part 1: creating the detail page

This page displays salary information for the department you selected when you click on a wedge of the pie chart. The department name is passed to this page using the `$ITEMLABEL$` variable.

To create the detail page:

1. Create a new application page with the following content:

```
<cfquery name="GetSalaryDetails" datasource="cfdocexamples">
  SELECT Departmt.Dept_Name,
    Employee.FirstName,
    Employee.LastName,
    Employee.StartDate,
    Employee.Salary,
    Employee.Contract
  FROM Departmt, Employee
  WHERE Departmt.Dept_Name = '#URL.Item#'
  AND Departmt.Dept_ID = Employee.Dept_ID
  ORDER BY Employee.LastName, Employee.Firstname
</cfquery>

<html>
<head>
  <title>Employee Salary Details</title>
</head>

<body>

<h1><cfoutput>#GetSalaryDetails.Dept_Name[1]# Department
  Salary Details</cfoutput></h1>
<table border cellspacing=0 cellpadding=5>
<tr>
  <th>Employee Name</th>
  <th>StartDate</th>
  <th>Salary</th>
  <th>Contract?</th>
</tr>
<cfoutput query="GetSalaryDetails" >
<tr>
  <td>#FirstName# #LastName#</td>
  <td>#dateFormat(StartDate, "mm/dd/yyyy")#</td>
  <td>#numberFormat(Salary, "$999,999")#</td>
  <td>#Contract#</td>
</tr>
</cfoutput>
</table>
</body>
</html>
```

2. Save the page as Salary_details.cfm in the myapps directory under the web root directory.

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfquery name="GetSalaryDetails" datasource="cfdocexamples">`
`SELECT`
`Departmt.Dept_Name,`
`Employee.FirstName,`
`Employee.LastName,`
`Employee.StartDate,`
`Employee.Salary,`
`Employee.Contract`
`FROM Departmt, Employee`
`WHERE`
`Departmt.Dept_Name =`
`'#URL.Item#'`
`AND Departmt.Dept_ID =`
`Employee.Dept_ID`
`ORDER BY Employee.LastName,`
`Employee.Firstname`
`</cfquery>`
	Get the salary data for the department whose name was passed in the URL parameter string. Sort the data by the employee's last and first names.

`<table border cellspacing=0 cellpadding=5>`
`<tr>`
`<th>Employee Name</td>`
`<th>StartDate</td>`
`<th>Salary</td>`
`<th>Contract?</td>`
`</tr>`
`<cfoutput query="GetSalaryDetails" >`
`<tr>`
`<td>#FirstName# #LastName#</td>`
`<td>#dateFormat(StartDate,`
`"mm/dd/yyyy")#</td>`
`<td>#numberFormat(Salary, "$999,999")#</td>`
`<td>#Contract#</td>`
`</tr>`
`</cfoutput>`
`</table>`
	Display the data retrieved by the query as a table. Format the start date into standard month/date/year format, and format the salary with a leading dollar sign, comma separator, and no decimal places.

Part 2: making the chart dynamic

1. Open chartdata.cfm in your editor.
2. Edit the `cfchart` tag for the pie chart so it appears as follows:

```
<cfchart
     font="Times"
     fontBold="yes"
     backgroundColor="##CCFFFF"
     show3D="yes"
     url="Salary_Details.cfm?Item=$ITEMLABEL$"
     >

   <cfchartseries
     type="pie"
     query="DeptSalaries"
     valueColumn="AvgByDept"
```

```
    itemColumn="Dept_Name"
    colorlist="##990066,##660099,##006699,##069666"
  />
</cfchart>
```

3. Save the file as chartdetail.cfm.
4. View the chartdata.cfm page in your browser.
5. Click the slices of the pie chart to request the Salary_details.cfm page and pass in the department name of the wedge you clicked. The salary information for that department appears.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
`url= "Salary_Details.cfm?Item=$ITEMLABEL$"`	When the user clicks a wedge of the pie chart, call the Salary_details.cfm page in the current directory, and pass it the parameter named Item that contains the department name of the selected wedge.

Linking to JavaScript from a pie chart

In the following example, when you click a pie wedge, ColdFusion uses JavaScript to display a pop-up window about the wedge.

To create a dynamic chart using JavaScript:

1. Create a new application page with the following content:

```
<script>
function Chart_OnClick(theSeries, theItem, theValue){
alert("Series: " + theSeries + ", Item: " + theItem + ", Value: " +
  theValue);
  }
</script>

<cfchart
    xAxisTitle="Department"
    yAxisTitle="Salary Average"
    tipstyle=none

  url="javascript:Chart_OnClick('$SERIESLABEL$','$ITEMLABEL$','$VALUE$');"
  >
  <cfchartseries type="bar" seriesLabel="Average Salaries by Department">
    <cfchartData item="Finance" value="75000">
    <cfchartData item="Sales" value="120000">
    <cfchartData item="IT" value="83000">
    <cfchartData item="Facilities" value="45000">
  </cfchartseries>
</cfchart>
```

2. Save the page as chartdata_withJS.cfm in the myapps directory under the web root directory.
3. View the chartdata_withJS.cfm page in your browser:
4. Click the slices of the pie chart to display the pop-up window.

CHAPTER 32
Creating Reports for Printing

This chapter explains how to use ColdFusion MX tags, functions, and tools to create pages and reports that are suitable for printing.

Contents

About printable output 763

Creating PDF and FlashPaper output with the cfdocument tag 764

Creating reports with the ColdFusion MX 7 reporting. 770

Font management with printable reports 804

Creating reports with Crystal Reports (Windows only) 805

About printable output

Although all web browsers let you print HTML pages, HTML-format pages are not optimized for printed output. For example, lack of control over line breaks, page breaks, headers, footers, and page numbers are just a few of the problems that you encounter when designing reports and other pages meant to be printed.

In the context of ColdFusion MX, the term *printable output* refers to pages that include the following features:

- Page numbers
- Headers and footers
- Page breaks
- Clickable hyperlinks when viewed online

ColdFusion MX provides the following tags for generating printable output:

- `cfdocument` Creates printable output and returns it to the browser or saves it in a file. For more information, see "Creating PDF and FlashPaper output with the cfdocument tag" on page 764.

- `cfreport` Invokes the specified report definition to create printable output and return it to the browser or save it in a file. ColdFusion MX supports report definitions from the following tools:
  - **ColdFusion Report Builder** The ColdFusion Report Builder is a banded report writer that is integrated with ColdFusion MX. For more information, see "Creating reports with the ColdFusion MX 7 reporting" on page 770.
  - **Crystal Reports** Crystal Reports is a report writer whose report definitions you can use with the `cfreport` tag. For more information, see "Creating reports with Crystal Reports (Windows only)" on page 805.

ColdFusion MX printable reports are available in the following formats:

FlashPaper ColdFusion MX creates a SWF file. Clients must have an up-to-date version of the Macromedia Flash Player installed.

Adobe Acrobat ColdFusion MX creates a PDF file. Clients must have the Acrobat Reader installed.

Microsoft Excel (ColdFusion MX 7 **reporting only)** ColdFusion MX 7 creates an Excel spreadsheet.

Note: The Excel report output format type provides limited support for the formatting options available in ColdFusion MX 7 reporting. Images and charts are not supported and numeric data containing formatting (commas, percents, currency, etc.) appears as plain text in Excel. The Excel output format supports simple reports only and it is recommended that careful design and layout consideration be given to reports designed for Excel output.

Crystal Reports (Windows only) ColdFusion MX passes control to Crystal Reports, which creates HTML. This option is available with the `cfreport` tag only.

Creating PDF and FlashPaper output with the cfdocument tag

The `cfdocument` tag converts everything between its start and end tags into PDF or FlashPaper output format and returns it to the browser or saves it to a file. This lets you easily convert HTML to printable output, as the following example shows:

```
<cfdocument format="FlashPaper">
<p>This is a document rendered by the
   cfdocument tag.</p>
</cfdocument>
```

The `cfdocument` tag supports all HTML and CFML tags, with the following exceptions:

- `cfchart`
- Macromedia Flash content
- Interactive tags, such as `form`, `cfform`, and `cfapplet`
- JavaScript that dynamically modifies elements or element positions

Additionally, the HTML wrapped by the `cfdocument` tag must be well-formed, with end tags for every start tag and proper nesting of block-level elements.

Note: ColdFusion MX does not return HTML and CFML outside of the `<cfdocument> </cfdocument>` pair.

Creating basic reports from HTML and CFML

You can convert HTML-based reports into PDF or FlashPaper output by wrapping the HTML in the `cfdocument` start and end tags, and specifying `cfdocument` attributes, as appropriate, to customize the following items:

- Page size
- Page orientation
- Margins
- Encryption (PDF only)
- User password and owner password (PDF only)
- Permissions (PDF only)

For complete information on these options, see the `cfdocument` tag discussion in *CFML Reference*.

Note: Embedding fonts in the report can help ensure consistent display across multiple browsers and platforms. For more information on the considerations related to embedding fonts, see "Font management with printable reports" on page 804.

The following example displays a list of employees, using a `cfoutput` tag to loop through the query:

```
<cfdocument format="flashpaper">
<h1>Employee List</h1>
<!--- Inline query used for example purposes only --->
<cfquery name="EmpList" datasource="cfdocexamples">
   SELECT FirstName, LastName, Salary, Contract
   FROM Employee
</cfquery>
<cfoutput query="EmpList">
#EmpList.FirstName#, #EmpList.LastName#, #LSCurrencyFormat(EmpList.Salary)#,
  #EmpList.Contract#<br>
</cfoutput>
</cfdocument>
```

Creating sections, headers, and footers

You can use the `cfdocumentitem` and `cfdocumentsection` tags to fine-tune your printable output, as follows:

- `cfdocumentitem` Creates page breaks, headers, or footers.
- `cfdocumentsection` Divides output into sections, optionally specifying custom margins. Within a section, use the `cfdocumentitem` tag to specify unique headers and footers for each section.

cfdocumentitem

You use one or more `cfdocumentitem` tags to specify headers and footers or to create a page break. You can use `cfdocumentitem` tags with or without the `cfdocumentsection` tag, as follows:

- With `cfdocumentsection` The `cfdocumentitem` attribute applies only to the section, and overrides previously specified headers and footers.
- Without `cfdocumentsection` The `cfdocumentitem` attribute applies to the entire document, as follows:
  - If the tag is at the top of the document, it applies to the entire document.
  - If the tag is in the middle of the document, it applies to the rest of the document.
  - If the tag is at the end of the document, it has no affect.

You can use the `cfdocumentitem` tag to create a running header for an entire document, as the following example shows:

```
<cfdocument format="PDF">
<!--- Running header --->
<cfdocumentitem type="header">
  <font size="-3"><i>Directory Report</i></font>
</cfdocumentitem>
<h3>cfdirectory Example</h3>
<!--- Use cfdirectory to display directories by name and size --->
<cfdirectory
  directory="#GetDirectoryFromPath(GetTemplatePath())#"
  name="myDirectory" recurse="yes"
  sort="directory ASC, name ASC, size DESC">
<!---- Output the contents of the cfdirectory as a cftable ----->
<cftable query="myDirectory"
  htmltable colheaders>
  <cfcol header="DIRECTORY:" text="#directory#">
  <cfcol header="NAME:" text="#Name#">
  <cfcol header="SIZE:" text="#Size#">
</cftable>
</cfdocument>
```

cfdocumentsection

When using `cfdocumentsection`, all text in the document must be enclosed within `cfdocumentsection` tags. ColdFusion MX ignores HTML and CFML outside of `cfdocumentsection` tags. The margin attributes override margins specified in previous sections or in the parent `cfdocument` tag. If you specify margin attributes, the units are controlled by the unit attribute of the parent `cfdocument` tag; the default for the unit attribute is inches.

Within a section, use the `cfdocumentitem` tag to specify unique headers and footers for each section and a page break before each section, as the following example shows:

```
<cfquery datasource="cfdocexamples" name="empSalary">
SELECT Emp_ID, firstname, lastname, e.dept_id, salary, d.dept_name
FROM employee e, departmt d
WHERE e.dept_id = d.dept_id
ORDER BY d.dept_name
```

```
</cfquery>

<cfdocument format="PDF">
  <cfoutput query="empSalary" group="dept_id">
  <cfdocumentsection>
  <cfdocumentitem type="header">
    <font size="-3"><i>Salary Report</i></font>
  </cfdocumentitem>
  <cfdocumentitem type="footer">
    <font size="-3">Page #cfdocument.currentpagenumber#</font>
  </cfdocumentitem>
  <h2>#dept_name#</h2>
   <table width="95%" border="2" cellspacing="2" cellpadding="2" >
    <tr>
     <th>Employee</th>
     <th>Salary</th>
    </tr>
    <cfset deptTotal = 0 >
    <!--- inner cfoutput --->
    <cfoutput>
      <tr>
          <td><font size="-1">
        #empSalary.lastname#, #empSalary.firstname#</font>
       </td>
       <td align="right"><font size="-1">
         #DollarFormat(empSalary.salary)#</font>
       </td>
          </tr>
     <cfset deptTotal = deptTotal + empSalary.salary>
    </cfoutput>
    <tr>
         <td align="right"><font size="-1">Total</font></td>
      <td align="right"><font size="-1">#DollarFormat(deptTotal)#</font></td>
       </tr>
    <cfset deptTotal = 0>
     </table>
    </cfdocumentsection>
  </cfoutput>
</cfdocument>
```

Using the cfdocument scope

When you use the `cfdocument` tag, ColdFusion MX creates a new scope named `cfdocument`. This scope contains the following variables:

currentpagenumber Displays the current page number.

totalpagecount Displays the total page count.

Note: The `cfdocument` scope variables are reserved for page number rendering. Do not use them in ColdFusion expressions.

You typically use these variables in a header or footer to display the current page number and total number or pages, as the following example shows:

```
<cfdocumentitem type= "footer> #cfdocument.currentpagenumber# of
  #cfdocument.totalpagecount#</cfdocumentitem>
```

Using cfhttp to display web pages

You can use the `cfhttp` tag in combination with the `cfdocument` tag to display entire web pages in PDF or FlashPaper output format, as the following example shows:

```
<!--- You can pass a URL in the URL string --->
<cfparam name="url.target_url" default="http://www.boston.com">
<cfoutput>
<cfhttp url="#url.target_url#" resolveurl="yes">

<cfdocument format="FlashPaper">
<cfdocumentitem type="header">
  <cfoutput>#url.target_url#</cfoutput>
</cfdocumentitem>
<cfdocumentitem type="footer">
  <cfoutput>#cfdocument.currentpagenumber# /
    #cfdocument.totalpagecount#</cfoutput>
</cfdocumentitem>
<!--- Display the page --->
#cfhttp.filecontent#
</cfdocument>
</cfoutput>
```

Using advanced PDF options

The `cfdocument` tag supports the Acrobat security options, as the following table shows:

Security option	Description
Encryption	Use the `encryption` attribute to specify whether PDF output is encrypted. Specify one of the following: • 128-bit • 40-bit • none
User password	Use the `userpassword` attribute to specify a password that users must enter to view the document.
Owner password	Use the `ownerpassword` attribute to specify a password that users must enter to view and optionally modify the document.

Additionally, the `cfdocument` tag supports the following Acrobat security permissions through the `permissions` attribute. Specify one or more of the following values; separate multiple permissions with a comma:

Permission	Description
Printing	Specify the `AllowPrinting` attribute to enable viewers to print the document.
Modification	Specify the `AllowModifyContents` attribute to let viewers modify the document, assuming they have the required software.
Copy	Specify the `AllowCopy` attribute to let viewers select and copy text from the document.
Annotation	Specify `AllowModifyAnnotations` to let viewers add comments to the document. If users add annotations, they must save the PDF after making changes.
Screen readers	Specify `AllowScreenReaders` to enable access to the document through a screen reader.
Fill in	Specify `AllowFillIn` to enable users to use form fields.
Assembly	Specify `AllowAssembly` to enable users to create bookmarks and thumbnails, as well as insert, delete, and rotate pages.
Degraded printing	Specify `AllowDegradedPrinting` to enable low-resolution printing. Low resolution printing prints each page as a bitmap, so printing may be slower.

Note: The defaults for these options vary, based on encryption level. These options apply to PDF only. For more information, see the `cfdocument` discussion in *CFML Reference*.

The following example creates a PDF document that allows copying only:

```
<cfdocument format="PDF" encryption="40-bit"
   ownerpassword="us3rpa$$w0rd" userpassword="us3rpa$$w0rd"
   permissions="AllowCopy" >
<h1>Employee List</h1>
<cfquery name="EmpList" datasource="cfdocexamples">
   SELECT FirstName, LastName, Salary
   FROM Employee
</cfquery>
<cfoutput query="EmpList">
#EmpList.FirstName#, #EmpList.LastName#,
  #LSCurrencyFormat(EmpList.Salary)#<br>
</cfoutput>

</cfdocument>
```

Saving printable reports in files

You can use the `cfdocument filename` attribute to save the generated PDF or SWF output to a file, as the following example shows:

```
<!--- The compasstravel database is part of the Getting Started
      tutorial application, found under the cfdocs directory. --->
<cfquery datasource="compasstravel" name="compasstrips">
SELECT tripName, tripDescription, tripLocation, price
FROM trips
ORDER BY price
</cfquery>
<cfdocument format="pdf"
   filename="#GetDirectoryFromPath(GetTemplatePath())#/compasstrips.pdf"
   overwrite="yes">
  <cfdocumentsection>
    <h1 align="center">Compass Travel</h1>
    <h2 align="center">Destination Guide</h2>
    <p align="center"><img
  src="cfdocs/getting_started/photos/somewhere.jpg"></p>
  </cfdocumentsection>
  <cfdocumentsection>
    <cfdocumentitem type="header">
      <font size="-3"> <i>Compass Travel Trip Descriptions</i></font>
  </cfdocumentitem>
    <cfdocumentitem type="footer">
      <font size="-3"> <cfoutput>Page
  #cfdocument.currentpagenumber#</cfoutput> </font>
    </cfdocumentitem>
    <cfoutput query="compasstrips">
      <hr>
      <h2>#tripName#</h2>
      <p><b>#tripLocation#</b></p>
      <p>Price: #DollarFormat(price)#</p>
      <p>#tripDescription#</p>
    </cfoutput>
  </cfdocumentsection>
</cfdocument>
```

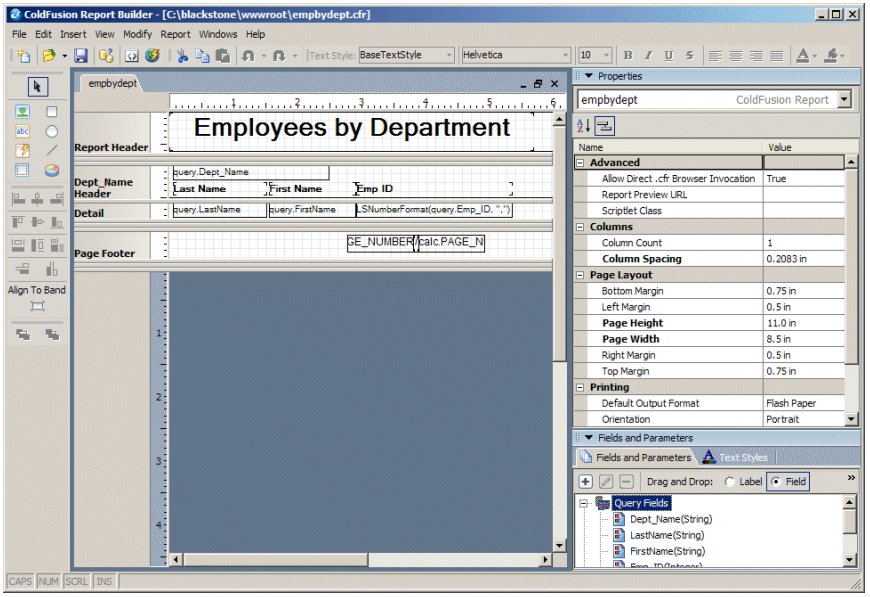
Creating reports with the ColdFusion MX 7 reporting

Macromedia ColdFusion MX 7 reporting adds integrated business reporting to ColdFusion, providing access to important business data. ColdFusion MX 7 reporting consists of server-side runtime processing and a graphical user interface (GUI), called the **Report Builder**.

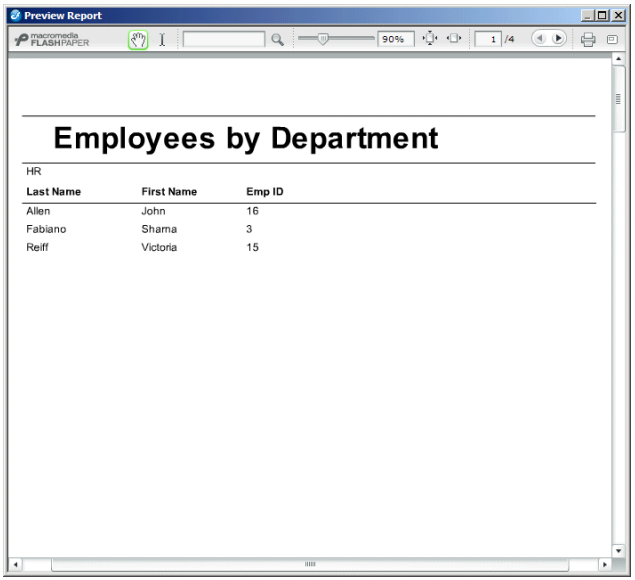
For information on installing the Report Builder, see "Getting started" on page 775.

This section describes how to use the Report Builder to define report definitions. The Report Builder is a Windows-only tool that lets you build **banded reports**. A banded report consists of multiple horizontal sections (bands), one band for each part of a printed report. For example, data and text in the report header band prints at the beginning of the report, data and text in the page header band prints at the beginning of each page, and data and text in the page footer band prints at the end of each page. In the middle of the report is the detail band, which, at runtime, contains one row for each row in the report's result set or database query.

The following figure shows the Report Builder workspace with a simple Employees by Department report:



The following figure shows a preview of that report in FlashPaper format:



This section contains the following discussions:

- Reporting features
- Reporting architecture
- Getting started
- Basic steps for creating reports
- Report definition guidelines
- Common reporting tasks and techniques
- Using the Report Creation Wizard
- Using the Query Builder
- Using CFML in reports
- Using charts
- Using subreports

Reporting features

Although you can use the `cfdocument` tag to create similar looking reports, ColdFusion MX 7 reporting and the Report Builder provide many features that ease the report creation process, as the following table describes:

Feature	Description
Easy-to-use interface	Employ a banded user interface with integrated online Help, For more information, see "User interface usage, tips, and techniques" on page 776.
Query Builder	Display, select, and order database tables and columns used in a report. The Query Builder creates SQL (also called internal SQL), which you can optionally customize using Advanced Query Builder. For more information, see "Using the Query Builder" on page 795.
Multiple display options	Display ColdFusion Report (CFR) files directly in a browser (no coding is required to create or display a ColdFusion report), or invoke a report with the `cfreport` tag, optionally passing query data to use instead of the internal SQL. For more information, see "Displaying reports" on page 791.
Report preview	Use Preview (F12) to display your report, optionally displaying a CFM page that invokes the report through the `cfreport` tag. For more information, see "Previewing reports" on page 787.

Feature	Description
Familiar language	Use CFML in the Report Builder, as follows: **Advanced query mode** Create a customized query object for the report using the `cfquery` tag, the `QueryNew` function, or any tag that returns a query. For more information, see "Advanced query mode" on page 797. **Report functions** Create user-defined functions that you can use to populate report fields. For more information, see "Report functions" on page 798. **Formatting** Use CFML functions to format report fields. For more information, see "Expressions and the Expression Builder" on page 799.
Global control of textual data	Use text styles for report-wide control over display fields. For more information, see "Using text styles" on page 786.
Wizards	The Report Builder provides the following wizards: **Report Creation Wizard** Define SQL, report fields, grouping, and other display options. For more information, see "Using the Report Creation Wizard" on page 794. **Chart Wizard** Define chart data, chart type, chart series, and other charting options. For more information, see "Using charts" on page 800. **Subreport Wizard** Insert a subreport and select other subreport options, including subreport parameters. For more information, see "Using subreports" on page 800. **Server Setup Wizard** Create settings for an RDS server, including file location and URL. For more information, see "Setup Wizard" on page 776.
Grouping	Define groups, each of which has its own group header and group footer bands. In particular, you can use calculated fields in group footer bands to easily display group totals for numeric fields. For more information, see "Grouping and group breaks" on page 780.
Charting	Insert a chart in any report band, displaying data from a variety of sources. For more information, see "Using charts" on page 800.
Subreports	Insert another report in any report band, optionally passing it a subreport parameter to control the data displayed in the subreport. For more information, see "Using subreports" on page 800.

Tip: The Report Builder contains an extensive online Help system, including quick-start tutorial topics and context-sensitive dialog box Help. Make it a habit to consult the online Help (F1) whenever you have a question.

Reporting architecture

The Report Builder is a stand-alone application that creates report definitions, interacting with a ColdFusion MX server, as necessary. The major architectural components are as follows:

- Report Builder and CFR files
- RDS
- Runtime

Note: The Report Builder runs in the Windows platform only. However, the CFR files created by the Report Builder run on all platforms that ColdFusion MX runs on and that have ColdFusion MX Reporting enabled.

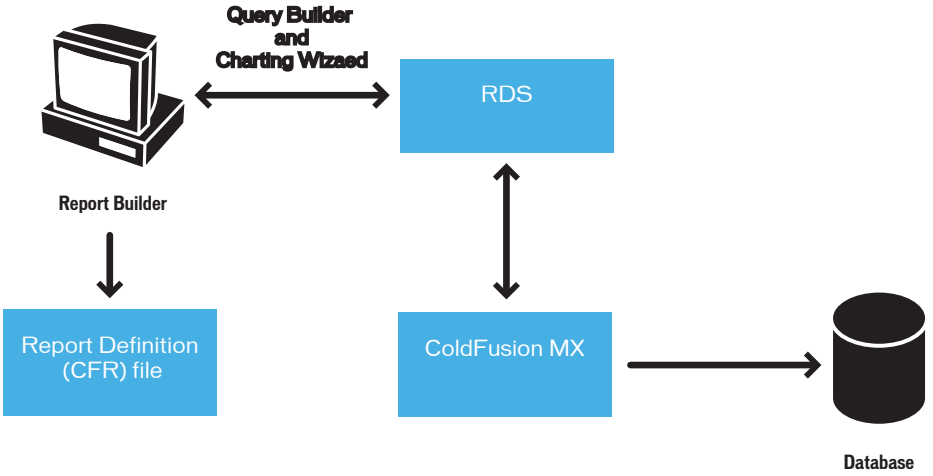
Report Builder and CFR files

The Report Builder stores report definition information in a ColdFusion Report (CFR) file. This file contains field definitions, formatting, database SQL statements, CFML, and other information. You display a CFR file using the cfreport tag and, if enabled for the report, you can display the report by invoking the CFR file in a browser.

RDS

Remote Development Services (RDS) is a proprietary protocol that uses HTTP to enable the Query Builder and Chart Wizard to access database data through a ColdFusion data source. To enable this functionality in the Report Builder, you define settings for an *RDS server*. RDS server is another name for an associated ColdFusion server that has enabled RDS.

The following figure shows the interaction of the Report Builder, CFR files, and RDS:

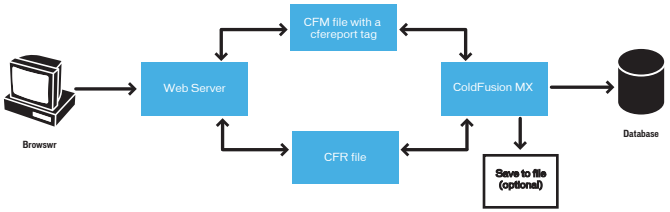


For more information, see "Configuring RDS" on page 792.

Runtime

At runtime, you invoke the CFR file using a ColdFusion MX server that has ColdFusion MX 7 Reporting enabled. For more information on enabling ColdFusion MX reporting, see "Getting started" on page 775.

You can display the CFR file directly or invoke it through the `cfreport` tag. Additionally, you can optionally save the report to a file instead of returning output to the browser. The following figure shows the ColdFusion MX reporting runtime architecture:



If the report requires input parameters or a passed query, you must use the `cfreport` tag. If you pass a query attribute in the `cfreport` tag, it overrides any internal SQL statement in the report definition.

Getting started

Before you get started using ColdFusion MX reporting, you must install the Report Builder.

After you install the Report Builder, start it by opening the Report Builder icon in the location you specified at installation time (for example, the desktop or the Start menu).

The first time you open the Report Builder, the Server Setup Wizard runs. After you complete the Server Setup Wizard, select File > New from the menu bar to define a new report using the Report Creation Wizard. For more information, see "Basic steps for creating reports" on page 778.

Installing the Report Builder

You can access the Report Builder installer from the following places:

The ColdFusion MX **installation** Go to the *webroot*/CFIDE/installers directory.

The ColdFusion MX **CD** Use the CD Browser to select the ColdFusion MX reporting option.

The Macromedia website Go to www.macromedia.com/software/coldfusion/reporting/.

When you install the Report Builder, it also registers Windows DLLs that are used by RDS. If these DLLs fail to register properly, the Report Builder generates errors at startup and when using RDS.

Setup Wizard

The first time you start the Report Builder, it runs the Setup Wizard. The Setup Wizard prompts you to define default settings for an associated ColdFusion MX server. These settings include the following:

- Default unit of measurement: Inches, centimeters, or pixels
- ColdFusion server (RDS must be enabled on this server). This is the RDS server that the Query Builder and Chart Wizard use to access database data. The Setup Wizard requires the following information:
  - Host name or IP address.
  - Web server port (typically 80 if using a web server connector, 8500 if using the built-in web server in the server configuration, 8300 if using the built-in web server with the cfusion server in the multiserver configuration, or a J2EE-server-specific web server port number).
  - RDS password for the associated ColdFusion server.
- Directory path to the webroot used by the associated ColdFusion server (for example, C:\Inetpub\wwwroot or C:\CFusionMX7\wwwroot).
- URL for the webroot used by the associated ColdFusion server (for example, http://localhost or http://localhost:8500).

After running the Setup Wizard, the Report Gallery dialog box appears. When you click the Using a Report Wizard radio button, the Report Builder runs the Report Creation Wizard, which prompts you for information and generates a complete report definition automatically.

For more information on the Report Creation Wizard, see "Using the Report Creation Wizard" on page 794.

User interface usage, tips, and techniques

The Report Builder user interface includes the following areas:

Toolbox Contains nonvariable elements placed in a report, including text, shapes, images, subreports, and graphs. To use toolbox elements, click on the element, and then click and drag in the report band to define the element's size. After you place an element on a report band, you can modify its appearance and behavior by using the Properties sheet.

Alignment palette Use Control-click or Shift-click to select multiple elements in a report band, and then click the appropriate alignment icon. You can also use Control-A to select all elements in a report band.

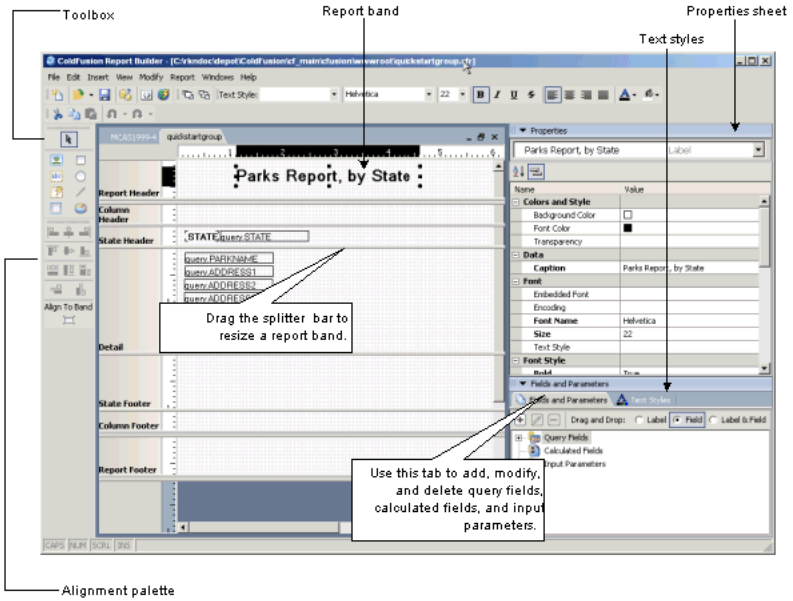
Report bands Place toolbox elements, query fields, and calculated fields on report bands. The default report bands are report header, page header, column header, detail, column footer, page footer, report footer, and watermark. Page header, page footer, and watermark are closed by default; to open them drag one of the adjacent splitter bars. To define additional bands for groups, select Report > Group Management.

Properties panel Contains display and report characteristics for the selected field. To change a property value, type or select a new value, and press Enter. For complete information on properties, see the online Help.

Fields and parameters panel Contains items for query fields, input parameters, and calculated fields. Use the add, edit, and delete icons to manage these fields. After you define a field, drag the field name to add the field, its associated label, or both, to a report band.

Text styles tool window Contains the text styles that you define for a report. Use the add, edit, and delete icons to manage text styles. After you define text styles, you apply them to text elements on the report instead of specifying font, font size, and so on, for each individual text element. If your report layout, platform, or font availability requirements change, you can modify the text style to apply the changes throughout the report. Additionally, you can specify that a text style is the report's default font and the Report Builder applies it to all text elements you add to a report.

The following figure shows the main areas of the Report Builder user interface:



The View menu lets you control whether toolboxes and tool windows appear. Additionally, you can click on a tool window's title to undock it and drag it to another area of the screen. To re-dock a tool window, drag it to the side or corner until a rectangle appears, and then release the mouse button.

For more information, see "Common reporting tasks and techniques" on page 779 and the online Help.

Basic steps for creating reports

The Report Builder is a very flexible tool that provides a variety of ways you can define a report. However, all reports use the following basic steps:

1. Begin a new report by selecting File > New from the menu bar, clicking the New toolbar icon, or pressing Control+N.

   The ColdFusion Report Builder Gallery dialog box appears.

2. Select the As a Blank Report option or the Using the Report Wizard option.

   Whether or not you use the Report Creation Wizard, continue with the following steps.

3. Define the data to be retrieved, either using the Query Builder or by defining the `cfquery` tag whose query is passed to the report through the `cfreport` tag. It's best to plan your data access before defining the report, as described in "Report definition guidelines" on page 779.

   Tip: You can use the Query Builder through the Report Creation Wizard or from the main Report Builder interface (select Report › Report Query from the menu bar).

4. Define query fields. In addition to the Query Builder, you can also define query fields using the Import or Add Fields buttons on the Report Creation Wizard Define Query Fields panel or the Query Fields node on the Fields and Parameters tab.

   Tip: Use the Import button to import query field definitions from an existing report.

5. Place query fields on the report. The Report Creation Wizard places query fields on the report bands automatically. To manually place query fields on a report, drag them from the Query Fields node on the Fields and Parameters tab to a report band. For more information, see "Common reporting tasks and techniques" on page 779.

6. (Optional) Define input parameters. To define input parameters, use the Input Parameters area on the Fields and Parameters tab. You pass input parameters to a report through the `cfreportparam` tag. For more information on input parameters, see "Using input parameters to pass variables and other data at runtime" on page 792.

7. (Optional) Define groups. Use the Report Creation Wizard Define Grouping panel or the Group Management dialog box (Report > Group Management) to define the query field whose change causes a group break. For more information on grouping, see "Grouping and group breaks" on page 780.

8. (Optional) Define the related CFM page. Whether or not the report has an internal query, you may want to invoke it through the `cfreport` tag in a CFM page. If you define a CFM page, be sure to specify the URL of the CFM page in the Report Preview URL field of the Report Properties dialog box (used in report preview).

   Tip: The Code Snippet Preview dialog box generates a prototype CFM page (or CFC and calling CFM page) and lets you save this code to a file (or files).

9. Preview, modify, repeat. To preview a report, select File > Preview from the menu bar, click the Preview toolbar icon, or press F12.

   For more information on report preview, see "Previewing reports" on page 787.

Report definition guidelines

To ensure a successful report, you should plan the following before defining it in the ColdFusion Report Builder:

- Report design issues:

  Audience Why are you creating this report? Who is the audience?

  Data What data needs to be in the report? Where does it come from? Whether you use the Query Builder or pass a query to the report, you should plan the data in advance.

  Grouping Do you need groups? If so, ensure that the result set is returned in the correct order, and you define a group based on the sort column.

  Calculated fields Are there fields that need to be totalled or calculated? For column totals, use calculated fields. For calculated totals on individual rows, use SQL. For more information, see "Common reporting tasks and techniques" on page 779.

  input parameters Do you need to pass variable data to the report. If so, define an input parameter and pass values to the report at runtime using the `cfreportparam` tag. For more information, see "Common reporting tasks and techniques" on page 779.

- Data retrieval strategy:

  Query Builder and basic SQL Use this option when your report has standard selection criteria (such as a WHERE clause with sorting and a fixed set of selection criteria) and when you need to develop a report quickly. This method also allows you to specify `cfquery` options, such as caching.

  Query Builder and advanced query mode Use this option when you want to use a ColdFusion query encapsulated in the report definition. This option is also useful if the query comes from the `cfdirectory`, `cfldap`, or `cfpop tags;` query of queries, or is dynamically constructed with the `QueryNew` function.

  The cfreport tag and a passed query Use this option when you need more control over the result set used in the report; for example, your application might have a form that your end users can use to construct dynamic selection criteria.

- Related visual information:

  Charts For more information, see "Using charts" on page 800.

  Subreports For more information, see "Using subreports" on page 800.

Common reporting tasks and techniques

This section describes the following reporting tasks and techniques:

- Grouping and group breaks
- Defining, modifying, and using fields and input parameters
- Using toolbox elements on report bands
- Aligning elements
- Using text styles
- Previewing reports

- Displaying page numbers
- Using layered controls
- Using links
- Using the Properties sheet
- Displaying reports
- Using input parameters to pass variables and other data at runtime
- Configuring RDS

For more information, including troubleshooting tips, see Report Builder online Help.

Grouping and group breaks

You can add clarity to a report's organization by grouping the information. You can define separate headings for each new group and can also display group-specific summary information, such as subtotals at the end of each group's area of the report. For example, you might create a report that displays departments, employees, and their salaries. Grouping the data by department lets users quickly understand department salary characteristics. When the department ID changes, the ColdFusion Report Builder triggers a group break, which completes the old group by displaying the group footer (which can contain calculated fields, such as total and average department salary) and starts the new group by displaying the group header.

The ColdFusion Report Builder does not group data itself. You must ensure that the SQL used for retrieve the result set is already grouped in the appropriate order; typically you implement grouping by specifying an ORDER BY clause in the SQL SELECT statement used for the report. For example, you might use the following SQL SELECT statement:

```
SELECT EmployeeID, LastName, FirstName, Title, City, Region, Country
FROM Employees
ORDER BY Country, City
```

For this example, you can define two groups: one that corresponds to Country, and a second group that corresponds to City. When you define more than one group, the Group Management dialog box appears Up Arrow and Down Arrow keys, which you can use to control group hierarchy. For example, country should be above city, because countries contain cities.

To define a group:

1. Select Report > Group Management from the menu bar.
2. Click Add.
3. Specify a group name.
4. Specify the value that controls grouping (also called a group expression). At runtime, ColdFusion triggers a group break when the result of this value changes. These values are often query field names. However, this value can also be a calculated field or other type of expression. Sample group expressions include the following:

   Query field Creates a group break when the associated column in the result set contains a different value. The field that you specify must be one of the sort criteria for the result set; for example, query.country.

Calculated field Creates a group break when a calculated field returns a different value. For example, if the expression calc.FirstLetter returns the first letter of a query column, you can group a report in alphabetical order.

Boolean expression Creates a group break when a Boolean expression returns a different value. For example, if your result set is sorted by the passpercentage column, you might use the Boolean expression query.passpercentage LT 50.

5. Specify group break options:

   Start New Column Forces a new column on a group break.

   Start New Page Forces a new page on a group break.

   Reset Page Number Resets the page number to 1 on a group break.

6. Specify band size and printing information:

   Min. height for group The minimum height that must remain on a page for ColdFusion to print the group band on that page.

   Reprint Header on Each Page Displays the group header on each page.

7. Click OK.

   The Report Builder adds the group to the report and creates header and footer bands for the group.

8. Click OK.

   The Group Management dialog box closes.

9. Add headings, text, query fields, calculated fields, and other information to the group's header and footer.

To create group subtotals:

1. Create a calculated field to contain the group subtotal. Create the calculated field using the following criteria:
   - Specify a numeric data type.
   - Select Sum in the Calculation field.
   - Specify the field to sum on in the Perform Calculation On field. For example, on an employees by department report, you might sum on query.emp_salary.
   - Specify that the field should be reset when the group changes.
2. Place the calculated field on the report.

For more information on calculated fields, see "Defining, modifying, and using fields and input parameters" on page 782 or the Report Builder online Help.

Defining, modifying, and using fields and input parameters

The Report Builder supports variable data through query fields, input parameters, and calculated fields, as follows:

Query field Maps to columns in the database result set associated with the report. You define one query field for each column in the associated database query.

Calculated field Analyzes or sums multiple detail rows in a report. ColdFusion dynamically generates calculated field values at report-generation time, optionally recalculating the value with each new report, page, column, or group.

Input parameter Specifies data fields that you pass to the report at runtime through the `cfreportparam` tag or from a main report to a subreport. You can place input parameters directly on a report band or you can use them as input to a calculated field.

To define a query field:

1. Click the Fields and Parameters tab. By default, this tab is in the lower-right area of the Report Builder window.
2. Click Query Fields.
3. Click the plus sign (+) at the top of the tab.

   The Add Query Field dialog box appears.
4. Type a value for the name field. This must match a column name in the corresponding cfquery statement and cannot contain a period.
5. Type a default label.
6. Specify the data type of the corresponding database column, as follows:

Object	Time	Long
Boolean	Double	Short
Byte	Float	Big Decimal
Date	Integer	Image
Time Stamp	BLOB	String

7. Click OK.

Note: The Query Builder defines query fields automatically for all database columns in the result set (does not apply to the Advanced Query Builder). Additionally, if you run the Query Builder as part of the Report Creation Wizard, the wizard places query fields on the report.

To define a calculated field:

1. Click the Fields and Parameters tab. By default, this tab is in the lower-right area of the Report Builder window.
2. Click Calculated Fields.
3. Click the plus sign (+) at the upper edge of the tab.

   The Add Calculated Field dialog box appears.

4. Specify a name, default label text, and data type. Data type options are the same as for query fields.
5. Specify calculation options:

   Calculation Specifies the type of calculation that ColdFusion performs. Valid values are: Average, Count, Highest, Lowest, Nothing, Standard Deviation, Sum, System, and Variance. If you specify Nothing, you typically use the Perform Calculation On field to specify a dynamic expression. With the exception of Nothing (for which you use the Perform Calculation On field) and System (for which you write a customized scriptlet class), you use these calculations for group, page, and report totals.

   Perform Calculation On Specifies a field or expression. Click the ... button to display the Expression Builder.

   Initial Value Specifies an initial value for the calculated field.
6. Specify reset options, as follows:

   Reset Field When Specifies when to reset the calculated field value. Valid values are: None, Report, Page, Column Group.

   Reset Group If Reset Field When is set to Group, use this field to specify the group whose group break triggers the reset.
7. Click OK.

For additional information on calculated fields, see "Add Calculated Field dialog box" in Report Builder online Help and "Expressions and the Expression Builder" on page 799.

To define an input parameter:

1. Click the Fields and Parameters tab. By default, this tab is in the lower-right area of the Report Builder window.
2. Click Input Parameters.
3. Click the plus sign (+) at the top of the tab.
4. The Add Input Parameter dialog box appears.

   Enter a value for the name field. This must match an input parameter, such as the name attribute of a `cfreportparam` tag included in the `cfreport` tag that invokes the report definition.
5. Enter the default label text.
6. Specify a data type and default value. Data type options are the same as for query fields.
7. Click OK.

For more information on using input parameters, see "Using input parameters to pass variables and other data at runtime" on page 792 and "Using subreports" on page 800.

To place a query field, calculated field, or input parameter on a report band:

1. In the Fields and Parameters tab, use the radio buttons to specify whether to place the label, the field, or both.
2. Drag the query field, calculated field, or input parameter from the Fields and Parameters tab to the appropriate report band.
3. Drag the query field, calculated field, or input parameter to the desired band.
4. (Optional) Use the Properties sheet to customize the field display.

For example, you might have a query field named query.emp_salary and a calculated field that sums query.emp_salary, resetting it with each group. Place query.emp_salary in the detail band, and the associated calculated field in the group footer band.

Using toolbox elements on report bands

You can use the toolbox to add graphic and textual elements, such as images, circles, squares, lines, dynamic fields, charts, and subreports, to report bands.

This section discusses text, images, circles, squares, lines, and dynamic fields. For information on charts, see "Using charts" on page 800. For information on subreports, see "Using subreports" on page 800.

The basic technique for adding toolbox elements is to click in the toolbox element and then drag to define an area in the appropriate report band. For some toolbox elements, such as image and text box, a dialog box immediately appears, prompting for more information. For all toolbox elements, you can customize the appearance of the element using the Properties sheet.

Tip: You can add toolbox elements through the Insert menu.

To use text boxes:

1. Click the Label icon (abc) in the toolbox.
2. Define the area for the label by dragging on the desired band.

   The Edit Label Text dialog box appears.
3. Enter the label text. To add a line break, press Control+Enter.
4. Click OK, or press Enter.

   Note: ColdFusion trims leading and trailing blanks from labels. To include leading and trailing blanks, define a dynamic field and include the blanks in the expression; for example, " My Title ".

To use image files:

1. Click the Image icon in the toolbox.
2. Define the area for the image by dragging on the desired band.

   The Image File Name dialog box appears.
3. Navigate to the file containing the image, select the file, and click OK.

To use a database BLOB column as an image source:

1. Click the Image icon in the toolbox.
2. Define the area for the image by dragging on the desired band.

   The Image File Name dialog box appears.

   Tip: You can also drag the BLOB field from the Fields and Parameters tab to a report band.
3. Click Cancel.

   The Expression Builder appears.
4. Click the Image Type drop-down list box and change File/URL to BLOB.
5. Select the query field or input parameter that contains the BLOB column.

   Note: The BLOB column must contain a binary image in GIF, JPEG, or PNG format.
6. Click OK.

Note: These instructions assume that the contents of the BLOB column can be rendered as an image.

To use rectangles, ellipses, and lines:

1. Click the rectangle, ellipses, or line icon in the toolbox.
2. Define the area or line by dragging on the desired band.
3. (Optional) Resize the selected element by dragging the handles that surround it.

   Tip: Pressing the Control key while resizing a rectangle, ellipsis, or line, constrains the element to a square, circle, or angles that are multiples of 45 degrees.

To use dynamic fields:

1. Click the Field icon in the toolbox.
2. Define the area for the dynamic field by dragging on the desired band.

   The Add Field dialog box appears (if you haven't defined any query fields, the Expression Builder appears).
3. Select the field to add. If you select a query field, calculated field, or input parameter, this is the same as dragging from the Fields and Parameters tab.
4. (Optional) Select Manually Entered Expression.

   The Expression Builder appears. This option is useful for calculations that use variables in the same row. For example, to compute total price for an order detail line item, you might use the following expression:

   ```
   LSNumberFormat((query.unitprice * query.quantity), ",_.__")
   ```
5. Click OK.

Aligning elements

Organized element layout is essential to a visually pleasing report. You achieve this organization by aligning, spacing, and centering visual elements on each band relative to each other, to the band itself, and to elements on other bands.

The Report Builder Align Palette includes the following options:

- Align left, center, and right
- Align top, horizontal, and bottom
- Same heights, widths, and both
- Space equally horizontally
- Space equally vertically

You align, size, and space multiple report elements, as follows:

Relative to the band they are in You control relative alignment through the Align to Band icon, which is the bottom icon in the Align Palette. When it is enabled, the Align to Band icon has a rectangle surrounding it, and the Report Builder aligns and spaces one or more elements relative to the height and width of the band.

Relative to each other When Align to Band is disabled, Report Builder aligns and spaces two or more elements relative to each other.

To use the Align Palette:

1. Select two or more elements by pressing Control-click, Shift-click, or using lasso select.
2. Click the alignment icon, or select Modify > Alignment > *alignment option* from the menu bar.

Tip: The Align Palette options are also available from Modify › Alignment on the menu bar.

For complete information on fine-tuning element display, see Report Builder online Help.

Using text styles

A text style is similar to a font style in Microsoft Word. Instead of explicitly associating a text element with font specifications, you associate the element with a text style. This provides you with report-wide control of the font characteristics of your report.

Additionally, you can specify that a text style is the default. The ColdFusion Report Builder uses the default text style for all text fields for which you have applied no other font specifications or text styles. The default text style, if defined, is displayed in bold in the Text Styles panel.

Note: When choosing fonts for your report, you must ensure that the fonts are available on the server that runs ColdFusion and (if you don't embed fonts) on the client computer. For more information on fonts, see "Font management with printable reports" on page 804.

To define a text style:

1. Click the Text Styles tab. By default, this tab is in the lower-right area of the Report Builder window.
2. Click the plus sign (+) at the top of the tab.

   The Add Text Style dialog box appears.
3. Type a value for the name field.
4. Add other text style characteristics.
5. Click OK.

To specify that a text style is the default:

1. Edit an existing text style or create one.

   The Add/Edit Text Style dialog box appears.
2. Select the Default check box.
3. Add or modify other text style characteristics, as appropriate.
4. Click OK.

For complete information on using text styles, see Report Builder online Help.

Previewing reports

Report building is an iterative process and most developers periodically display the in-progress report to review their most recent changes. If your report uses an internal query and you have established default webroot settings, preview functionality is enabled automatically. If your report uses a passed query, you must define an associated CFM page and associate that page with the report. The Report Builder invokes this page when you request Report Preview.

To use Report Builder preview with a report that uses an internal query:

1. (Optional) Define default server connection information using the Preferences dialog box, if you did not define these settings previously:
   - Default RDS server configuration (used for Query Builder and Chart Wizard only; not required for report preview).
   - Fully qualified path for the local web root directory; for example, C:\CFusionMX7\wwwroot or C:\Inetpub\wwwroot.
   - URL for the local web root, for example, http://localhost:8500 or http://localhost.
2. (Optional) Specify the output format in the Report Properties dialog box (the default is FlashPaper).
3. (Optional) If your report is designed to be invoked by a CFM page, specify the URL of the CFM page in the Report Properties dialog box.
4. Save your report.

5. Select File > Preview from the menu bar.
   or
   Click the Preview button on the toolbar.
   or
   Press F12.

   The Preview Report window displays your report.

   Note: If the Report Builder displays the Edit Preview Report URL dialog box instead of displaying the Preview window, select Edit › Preferences from the menu bar and ensure that the web root file and URL settings are correct on the Server Connection pane.

6. Close the preview window by pressing F12.

If your report is designed to accept a query object from a `cfreport` tag, you must associate a URL with the report. If necessary, the Report Builder prompts for this URL when you preview the report. Otherwise, you can open the Report Properties dialog box, and specify the URL of the CFM page in the Report Preview URL field.

Tip: You can use the `cfreport` tag to invoke a report, regardless of whether the report has an internal query or is passed a query.

To use Report Builder preview with an associated CFM file:

1. Select Report > Report Properties from the menu bar.
2. Specify the URL of the associated CFM page in the Report Preview URL field. This CFM page must contain a `cfreport` tag whose `template` attribute specifies the current CFR file and, if necessary, passes a query in the `query` attribute.
3. Save your report.
4. Select File > Preview from the menu bar.
   or
   Click the Preview button on the toolbar.
   or
   Press F12.

   The Preview Report window displays your report in PDF, FlashPaper, or Excel format.

Displaying page numbers

The Report Builder includes a built-in calculated field named `PAGE_NUMBER`, which displays the current page number when you place it on a report band.

To add a built-in calculated field to a report:

1. Click the Field tool in the toolbox.
2. Drag in the center of the header or footer band to define the size of the page number field.

   The Add Field dialog box appears, listing all fields defined for the report, including built-in calculated fields and input parameters.
3. Select `calc.PAGE_NUMBER`, and click OK.

Tip: You can use the Field tool to add any type of field (query field, calculated field, input parameter) to a report.

For information on the other built-in calculated fields, see the Report Builder online Help.

Using layered controls

Layered controls are elements that you place at the same location of a report band, and then use `PrintWhen` expressions to conditionally display one or the other at runtime. You can use layered elements to customize the circumstances under which the elements display and enhance a report's ability to communicate important information.

To place an element directly over another element:

1. Place the elements on the band.
2. Specify a `PrintWhen` expression, display properties, and placement properties for each element using the Properties sheet, as follows:
3. Specify a `PrintWhen` expression for each element. For example, you might specify the following expression to display one element when shippeddate is later than requireddate (that is, late) and another element when shippeddate is earlier than requireddate:

   First element query.shippeddate LTE query.requireddate

   Second element query.shippeddate GT query.requireddate
4. Specify different display characteristics for each element. For example, if an order is late, display it in red text.
5. Set the Top, Left, Height, and Width properties to the same values for each element.

Tip: When you specify identical placement properties, you access the individual elements through the Layered Controls menu.

To use the Layered Controls menu:

1. Right-click on the top element.
2. Select Layered Controls > *elementname* from the pop-up menu. The Report Builder identifies each layered element by displaying its `PrintWhen` expression.
3. Select the element.

   The Properties sheet displays properties for the selected element.

Using links

You can include hyperlinks from query fields, calculated fields, input parameters, charts, and images to a variety of destinations:

- An anchor or page within the same report
- An anchor or page within another report
- An HTML page, optionally specifying an anchor and URL parameters

One use for this feature is to create drill-down reports, in which you click an item to display detailed information. For example, clicking an employee line item passes the employee ID as a parameter to a page that displays complete information for the employee.

For complete usage information on creating anchors and hyperlinks, see the Report Builder online Help.

Using the Properties sheet

Every element on a report, including the report itself, is defined by a set of properties. These properties affect the look, feel, and behavior of each element.

For many properties, the Report Builder lets you define their values through user interface elements, such as dialog boxes, toolbar icons, and menu items. For example, you set a text label's font size using a toolbar icon. You can set values for all properties, however, through the Properties sheet, which display all properties for the currently selected element.

Tip: Sometimes a report contains multiple, closely spaced elements and it is difficult to select an individual element using the mouse. In this case, selecting the element from the Properties sheet drop-down list is an easy way to select an element.

The Properties sheet has two views:

Sort alphabetically All properties for the currently selected element display in alphabetical order.

Sort into groups The Properties sheet displays related properties in the following predefined groups:

- Colors and Style
- Data
- Font
- Font Style
- Formatting
- Hyperlinks
- Layout
- Processing

The Report Builder displays only groups that relate to the currently selected element.

To set or modify a property for an element in the workspace:

1. Select the element.
2. The Report Builder displays its properties in the Properties sheet.
3. Modify the property. Depending on the property, you enter a value, select a value from a list, or open the Expression Builder to use an expression.
4. Press Enter.

To choose a different element:

- Select the element from the drop-down list. When you select a new element, the Report Builder selects the element and displays its properties.

Tip: Although the Properties sheet is a powerful way to set properties, you typically set properties through dialog boxes and toolbar icons. For example, you use the Report Properties dialog box to set report-wide settings. For complete information on setting properties, see "Property reference" in Report Builder online Help.

Displaying reports

Your application can invoke a report by displaying the CFR file in a browser or by displaying a CFM page whose `cfreport` tag invokes the report.

You can optionally use the `cfreport` tag to save the report to a file.

The `cfreport` tag supports advanced PDF encryption options. For more information, see *CFML Reference* and "Using advanced PDF options" on page 768.

For information on report preview, see "Previewing reports" on page 787.

To display a report using the cfreport tag:

1. Create a report, with or without an internal query.
2. Create a CFM page and add a `cfreport` tag that invokes the report. If the report does not use an internal query, you must also populate a query and pass it using the `query` attribute. If the report uses an internal query and you use the `query` attribute, the passed query overrides the internal query.

```
<cfquery name="northwindemployees" datasource="localnorthwind">
  SELECT EmployeeID, LastName, FirstName, Title, City, Region, Country
      FROM Employees
      ORDER BY Country, City
</cfquery>

<CFREPORT format="PDF" template="EmpReport.cfr"
   query="#northwindemployees#" />
```

3. Open a browser and display the CFM page.

   ColdFusion MX generates the report.

   Tip: ColdFusion MX does not render text that occurs before or after the `cfreport` tag.

To display a CFR file in a browser:

1. Create a report that uses an internal query and does not use input parameters.
2. Open a browser and display the CFR file.

   ColdFusion MX generates the report.

To save a report to a file:

1. Create a report, with or without an internal query.
2. Create a CFM page and add a `cfreport` tag that invokes the report. Optionally pass a `query` attribute, as described in the previous procedure. Include a `filename` attribute that specifies the fully qualified name of the file to be created, as the following example shows:

```
<CFREPORT format="PDF" template="emppicture.cfr"
  filename="#GetDirectoryFromPath(GetTemplatePath())#/emppicture.pdf"
  overwrite="yes"/>
```

   Tip: Use the .pdf extension for PDF output format, the .swf extension for FlashPaper output format, and the .xls extension for Excel format.

3. Open a browser and display the CFM page.

   ColdFusion MX generates the report, saves the file, and displays an empty page.

To disable browser display of the CFR file:

1. Open the Report Properties dialog box by selecting Report > Report Properties from the menu bar.
2. Clear the Allow Direct .cfr Browser Invocation check box.
3. Click OK.

Using input parameters to pass variables and other data at runtime

Input parameters are data fields that you pass to the report at runtime. You can place input parameters directly on a report band or you can use them as input to a calculated field.

Define input parameters in the same manner as query fields. You can specify a default value that ColdFusion uses when there is no corresponding parameter. For more information on defining input parameters, see "Defining, modifying, and using fields and input parameters" on page 782.

You use input parameters in the following ways:

- **Through the `cfreportparam` tag** Input parameters must correspond, by name, to `cfreportparam` tags embedded in the `cfreport` invocation. For example, if you define an input parameter named ReportTime, you pass a `cfreportparam` tag with a name attribute set to ReportTime, as the following example shows:

```
<cfreport format="PDF" template="FourthReport.cfr" query="#coursedept#">
   <cfreportparam name="ReportTime" value="#DateFormat(Now())#,
  #TimeFormat(Now())#">
</cfreport>
```

- **Subreport parameters** When a subreport requires information from a main report, you define subreport parameters in the main report and corresponding input parameters in the subreport. For more information, see "Using subreports" on page 800.

For information on dynamically populating input parameters at runtime, see "Advanced query mode" on page 797.

Configuring RDS

You must configure one RDS server for each ColdFusion MX server for which you define reports. After you configure an RDS server, you can use the Query Builder to access data sources that you defined in the associated ColdFusion MX server, and select database columns for use as query fields in a report.

To add an RDS server:

1. Open the Preferences dialog box by selecting Edit > Preferences from the menu bar.
2. Click Server Connection.
3. Click the plus sign (+) next to the drop-down list in the upper-left corner of the dialog box.

   The Configure RDS Server dialog box appears.

4. Specify the following information:

   Description A name for the server connection. This name appears in the drop-down list on the left side of the Query Builder.

   Host name The host on which ColdFusion runs. Type localhost or an IP address.

   Port Web server port number. Accept the default port (80) or enter the port number of the ColdFusion server's built-in web server (8500 is the default).

   Context Root (J2EE configuration only) Specifies the context root (if any) for the ColdFusion web application.

   Use Secure Sockets Layer Select this option if needed for SSL security.

   User Name Not applicable to ColdFusion RDS.

   Password RDS password. You set this password in the ColdFusion MX Administrator.

   Tip: Do not confuse the RDS password with the ColdFusion MX Administrator password, which you also manage through the ColdFusion MX Administrator.

   Prompt for password Specifies whether to prompt for an RDS password each time you use the Query Builder. If you select this option, leave the User Name and Password fields blank.

5. Click OK.

To designate a default RDS server:

1. Open the Preferences dialog box by selecting Edit > Preferences from the menu bar.
2. Click Server Connection.
3. Select an RDS server from the Preferred RDS Server drop-down list.
4. Click OK.

   The Report Builder automatically connects to the specified server when you display the Query Builder or Chart Wizard.

Using the Report Creation Wizard

You can use the Report Creation Wizard to quickly start creating most types of reports. The wizard steps you through the report creation process through the following panels:

Panel	Description
Query Builder	Creates the SQL statement used for database access. You can type a SQL statement manually in the bottom pane or let the Query Builder create the SQL statement by using the data source browser to choose the data source, tables, and columns for the report. You can also use the Advanced Query Builder to create a query using CFML. For more information on using the Query Builder, see "Using the Query Builder" on page 795.
Define Query Fields	Lets you define query fields through the following ways: **Query Builder** Automatically creates query fields for all columns retrieved by basic SQL. The Report Builder does not define fields automatically for advanced queries. **Importing fields from an existing report** By importing query fields from an existing report. **Manually** By creating them manually. You also use the Define Query Fields panel to specify whether query fields appear on a report band. For more information on query fields, see "Defining, modifying, and using fields and input parameters" on page 782.
Define Report Grouping	Lets you specify which query fields, if any, are used for group breaks. The Report Builder does not automatically group data. You must ensure that the SQL used for retrieve the result set is already grouped in the appropriate order. For more information on grouping, see "Grouping and group breaks" on page 780.
Report Layout	Lets you choose one of several report layout options. The layout options, which differ depending on the number of fields in the report and on whether or not the report uses grouping. You also use the Report Layout panel to specify whether the report uses portrait or landscape orientation.
Report Style	Lets you further customize the appearance of the report through the following options: **Style** Includes data and labels only, or include predefined headings. **Numerical totals** Automatically creates column totals for all numerical fields. **Column specifications** Controls the number of columns, and the spacing between columns.

Panel	Description
Report Theme	Lets you specify a color scheme and whether the report header prints on a page by itself.
Report Name	Lets you specify the following: **Report header** The report header automatically includes a text field for company name. **Page header** The page header automatically includes the date. **Page footer** The page footer automatically includes the page number.

For more information on the Report Creation Wizard and Report Creation Wizard panels, see Report Builder online Help.

The Report Creation Wizard takes the information that you enter into these panels and creates a report. In some cases, this report will have everything you need. However, in most cases, you will continue to make modifications. In particular, the Report Creation Wizard does not define the following:

Calculated fields For more information, see "Defining, modifying, and using fields and input parameters" on page 782.

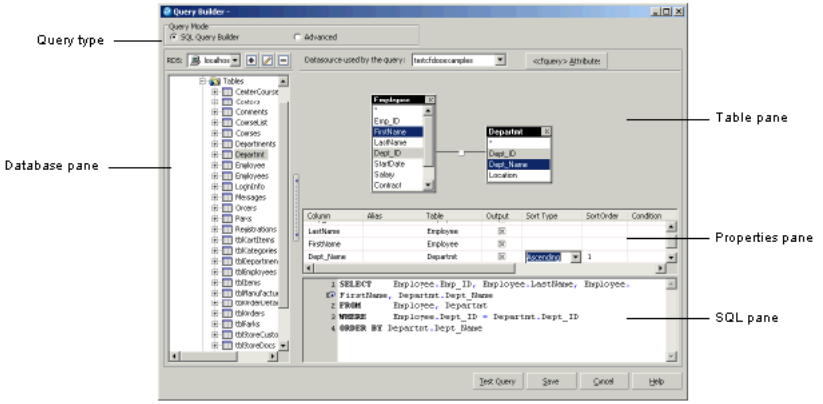
Input parameters For more information, see "Defining, modifying, and using fields and input parameters" on page 782.

Charts For more information, see "Using charts" on page 800.

Subreports For more information, see "Using subreports" on page 800.

Using the Query Builder

You use the Query Builder to define the SQL statement that retrieves the result set used by a report. The following image shows the parts of the Query Builder interface:



You use these parts, called *panes*, as follows:

Database pane Displays a list of data sources for an RDS server. Expand the data sources to display a list of tables in the related database. To use a table, double-click it or drag it to the Table pane. If you have selected a data source, you can also select a table by right-clicking on the Table pane.

Table pane Displays columns in the tables you selected. Double-click a column to add it to the Properties pane and to the SELECT statement that is displayed in the SQL pane.

Note: The Table pane can only browse data. You cannot modify data or add new records here. Also, views are available only in databases supporting the creation of views or tables stored as queries.

Properties pane Displays selected columns. Use this pane to specify additional criteria, such as ALIAS names, sorting, sort order, and selection criteria for the WHERE clause.

SQL pane Displays the SQL statement as it is being constructed. The SQL pane does not support reverse editing, so any changes that you make in this pane are not made in the Properties pane or the Table pane.

Query type Controls whether you create a SQL statement through the Query Builder or by creating a query using CFML (also known as advanced query mode). To create a query using CFML, click Advanced.

The Query Builder uses RDS to access database schema information through the data sources defined to a ColdFusion MX server. For more information, see "Configuring RDS" on page 792.

The Query Builder automatically defines query fields for the columns returned by your SQL statement. Because the Query Builder executes the SQL statement and analyzes the columns that it returns, automatic query field definition applies to both stored procedures and SELECT statements.

Note: The Query Builder does not create query fields automatically for queries generated using advanced query mode.

Not all DBMSs support all SQL syntax. Ensure that your DBMS supports the SQL statements generated by the Query Builder. For example, the Microsoft Access and Microsoft Access with Unicode drivers do not support FULL OUTER JOIN clauses.

To select an RDS server:

- Select the RDS server from the drop-down list in the upper left corner of the Query Builder.

Caution: The Query Builder clears the current SELECT statement when you select a new RDS server.

You can define RDS settings for multiple ColdFusion servers. For more information, see "Configuring RDS" on page 792.

To select a data source in the current RDS server:

- Select the data source name in the Data Source Used by the Query drop-down list at the upper edge of the Query Builder.

Caution: The Query Builder clears the current SQL statement when you select a new data source,

For more information on creating SQL statements with the Query Builder, see "Using the Query Builder" in Report Builder online Help.

To include cfquery attributes in the SQL statement:

1. Use the Query Builder to create a SQL statement.
2. Click the `<cfquery>` Attributes button.

   The CF Query Options dialog box appears.
3. Specify `cfquery` attributes through dialog box fields.
4. Click OK.

Using CFML in reports

CFML is the scripting language for the Report Builder. By leveraging CFML, you can easily create reports that select and format data to meet your needs. You can use CFML in the following areas of the Report Builder:

- Advanced query mode
- Report functions
- Expressions

Advanced query mode

In some cases, you may want to use create a complex query, reuse an existing query, or encapsulate additional CFML processing as part of query creation for the report. In this case, you can use advanced query mode to create CFML that returns a query. When you click the Advanced button at the top of the Query Builder, the Report Builder displays a text entry area in which you can enter CFML that generates a query. ColdFusion executes this tag at report execution time and passes the query result set to the report.

Note: When you use advanced query mode, the Query Builder does not create query fields automatically. You must create the associated query fields manually.

The CFML used in advanced query mode must include a query object whose name matches that in the Variable containing query object field. You can use any CFML tag that returns a query object or the `QueryNew` function. The CFML can use multiple query objects, but can only return one.

Note: If you set an empty variable (for example, <cfset name="">), the Report Builder throws a Report data binding error.

This example CFML uses the `cfhttp` tag to retrieve a query:

```
<cfhttp
  url="http://quote.yahoo.com/download/quotes.csv?Symbols=csco,jnpr&format=sc
  l11&ext=.csv"
method="GET"
name="qStockItems"
columns="Symbol,Change,LastTradedPrice"
textqualifier="""
```

```
delimiter=","
firstrowasheaders="no">
```

Another possible use of advanced query mode is to test for passed parameters in the URL or FORM scopes and use these parameters to retrieve data, as the following example shows:

```
<!--- First look for URL parm.
      URL overrides cfreportparam. --->
<cfif isDefined("url.deptidin")>
  <cfset param.deptidin = url.deptidin>
</cfif>

<!-- Then look for FORM parm. Overrides URL parm. --->
<cfif isDefined("form.deptidin")>
  <cfset param.deptidin = form.deptidin>
</cfif>

<cfquery name="CFReportDataQuery" datasource="cfdocexamples">
SELECT    LastName, FirstName, Dept_ID
FROM      Employee
WHERE     (Dept_ID = #param.deptidin#)
</cfquery>
```

Report functions

Report functions are user-defined CFML functions that you code using the Report Function Editor and invoke in report fields. You can use them to format data (such as concatenating and formatting all the field that make up an address), to retrieve data, and for many other purposes.

To create a report function:

1. Select Report > Report Functions from the menu bar.

   The Report Function Editor displays.

2. Click the plus sign to add a new report function.

   The Add Report Function dialog box displays.

3. Specify a name and click OK.

4. The Report Function Editor places a `cfreturn` tag in the text entry area.

5. Code the function. This is a ColdFusion user-defined function so all UDF rules and features are available for use. The following example show a report function that concatenates address fields:

   ```
   <cfargument name="Name" required="yes"/>
   <cfargument name="Address1" required="yes"/>
   <cfargument name="Address2" required="yes"/>
   <cfargument name="City" required="yes"/>
   <cfargument name="State" required="yes"/>
   <cfargument name="Zip" required="yes"/>

   <cfset variables.CRLF = Chr(13) & Chr(10)>
   <cfset variables.ResultVar="">

   <cfif Trim(arguments.Name) NEQ "">
   ```

```
    <cfset variables.ResultVar='#arguments.Name#'>
</cfif>
<cfif Trim(arguments.Address1) NEQ "">
    <cfif variables.ResultVar NEQ "">
        <cfset variables.ResultVar='#variables.ResultVar & variables.CRLF#'>
    </cfif>
    <cfset variables.ResultVar='#variables.ResultVar & arguments.Address1#'>
</cfif>
<cfif Trim(arguments.Address2) NEQ "">
    <cfif variables.ResultVar NEQ "">
        <cfset variables.ResultVar='#variables.ResultVar & variables.CRLF#'>
    </cfif>
    <cfset variables.ResultVar='#variables.ResultVar & arguments.Address2#'>
</cfif>
<cfif variables.ResultVar NEQ "">
    <cfset variables.ResultVar='#variables.ResultVar & variables.CRLF#'>
</cfif>

<cfset variables.ResultVar='#variables.ResultVar & arguments.City & ", " &
  arguments.State & " " & arguments.Zip#'>

<cfreturn variables.ResultVar>
```

6. Click OK when you are finished.

To use a report function:

1. Place a dynamic field on the appropriate report band.

   The Add Field dialog box displays.

2. Specify Manually Entered Expression and click OK.

   The Expression Builder displays.

3. Specify "report.*functionname*".

4. Click OK.

Expressions and the Expression Builder

Many elements of the Report Builder (including query fields, calculated fields, input parameters, images, and report object attributes) are single operand ColdFusion expressions. Because these elements are expressions, you can manipulate them with CFML functions.

The Expression Builder is a graphical interface that lets you quickly apply CFML functions to Report Builder elements. Uses for the Expression Builder include the following:

- Many of the report object attributes (such as PrintWhen) accept expressions, which you can associate with query parameters, input parameters, or ColdFusion page variables. You can tie report attributes and columns to display based on runtime data or user preference.
- Concatenating fields
- Formatting fields
- Calculated fields
- Accessing and displaying ColdFusion page variables and scopes

For information on using the Expression Builder, see Report Builder online Help.

For more information on expressions, see Chapter 4, "Using Expressions and Number Signs".

Using charts

Charts can help clarify large or complex data sets. The Report Builder lets you place a chart in any report band and supports many types of charts.

To add a chart to a report, you use the Chart Wizard, which steps you through the chart building process. The Chart Wizard, which is fully integrated with the Query Wizard to facilitate database-driven charts, helps you define the chart type, the data used for the report and other formatting options.

Tip: As you use the Chart Wizard to choose and define the various aspects of a given chart, the Report Builder uses RDS to generate chart images in real time. However, the data shown in these chart images is not real.

The Chart Wizard includes the following panels:

- **Chart Types** Select the chart type (for example, bar) and sub-type (for example, 3D-stacked).
- **Chart Series** Select the data for the series. When you add a series, the Report Builder lets you hard-code series data or open the Query Builder to populate the series using a database query.
- **Chart Formatting** Specifies title & series, general appearance, 3D appearance, lines & markers, and font.

Tip: The data you specify through the Chart Wizard corresponds to the attributes specified in the `cfchart`, `cfchartseries`, and `cfchartdata` tags. For more information on these tags, refer to *CFML Reference* and *ColdFusion MX Developer's Guide*.

For complete information on ColdFusion MX charting capabilities, see Chapter 31, "Creating Charts and Graphs". For more information on charting using the Report Builder, see Report Builder online Help.

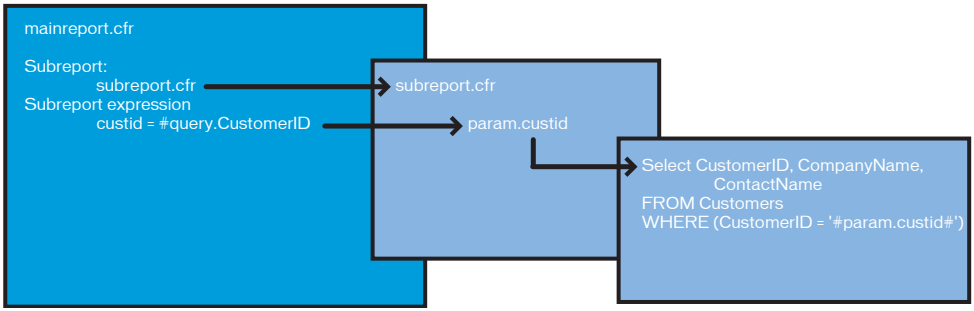
Using subreports

Subreports let you nest a report within your report. The data you display in a subreport is typically related to the data in the main report and you enable this by passing one or more subreport parameters to the subreport. However, the data displayed in a subreport can also be unrelated to the data in the main report.

Reasons to use subreports including the following:

- You wish to avoid complex SQL, such as a RIGHT OUTER JOIN.
- Your report requires data from multiple databases.

The following figure illustrates the use of subreport parameters and the relationship between a report and a subreport:



Note: Although the Report Builder supports multiple levels of nesting, it displays one level of nesting only.

For additional information on subreports, see Report Builder online Help.

Defining a subreport

You can define a subreport and include it in a report, or you can define it as part of inserting the subreport in the main report.

However, regardless of how you define it, a subreport has the following characteristics:

- Data displayed in the detail band only. A subreport uses no header or footer bands.
- If the subreport is related to the main report, it must include an internal query that uses a SELECT statement with a WHERE clause specifying the name of the input parameter used in the main report's Subreport Expression property.

Adding an existing subreport to a report

If you have already defined a subreport, you add it to the main report and define subreport parameters, as necessary.

To add an existing subreport to a report:

1. Define or open your main report.
2. Click the Subreport icon in the toolbox.
3. Drag an area for the subreport in the desired report band.

   The Subreport Source panel appears.
4. Select From an Existing Report, specify the subreport, and click Next.

   The Report to Subreport Parameter Binding panel appears.
5. Select the fields in the main report that correspond to fields in the subreport and click Next.

   The Finish Subreport Setup panel appears.

6. Click Finish.

   The Report Builder adds the subreport to the main report, saving the report to subreport mappings as subreport parameters.

7. To modify subreport parameter settings, select the subreport and click on Subreport Parameters in the Properties sheet.

Adding a new subreport to a report

If you are certain of the data required for a subreport, you can define a new subreport while adding it to the main report.

To add a new subreport to a report:

1. Define or open your main report.
2. Click the Subreport icon in the toolbox.
3. Drag an area for the subreport in the desired report band.

   The Subreport Source panel appears.

4. Select As a New Report and click Next.

   The Define Query Fields panel appears.

5. Click Query Builder.

   The Query Builder appears.

6. Select the tables and columns for the subreport.
7. Specify a WHERE clause for the report by using the Condition and Criteria columns for the key columns.

   Specify WHERE for Condition and either `='#CFVariable#'` (string column) or `=#CFVariable#` (numeric column) for Criteria, and then overtype `CFVariable` with the name of the input parameter for the subreport (you define the input parameter name later in the procedure.)

8. Click Save.
9. Click Next.

   The Define Report Grouping panel appears.

10. Specify grouping fields, if appropriate for your subreport.
11. Click Next.

    The Report Layout panel appears.

12. Specify Free Form or Grid. Then click Next.

    The Report Style panel appears.

13. Specify Only Detail Band and click Next.

    The Report Theme panel appears.

14. Specify a color scheme and click Next.

    The Report Name panel appears.

15. Specify headings, as appropriate, and click Next.

    The Subreport Binding panel appears.

16. For each parameter required by the subreport, specify the following:
    - Parameter name.
    - Associated value from the main report (click the drop-down list box).
    - Data type.

17. Click Next.

    The Subreport Filename panel appears.

18. Specify a fully qualified filename for the subreport.
19. Click Next.

    The Finish Subreport Setup panel appears.

20. Click Finish.
21. Modify the subreport, as necessary.

    The Report Builder adds the subreport to the main report.

Modifying subreport settings

The Report Builder lets you change subreport name and modify subreport parameters in a main report.

To modify subreport settings:

1. Click the subreport element in the main report.

   Subreport settings display in the Properties sheet.

2. To change the subreport, modify Subreport Expression.
3. To modify subreport parameters:
   a. Click the Subreport Parameters property.
   b. Click the ... button.

      The Subreport Parameters dialog box appears.

   c. Add, modify, or delete subreport parameters.
   d. Click OK.

Font management with printable reports

Ideally, reports should achieve a consistent look across all client platforms and all browsers. ColdFusion MX handles this automatically for graphics and images, using the size specifications in the report definition. However, potential differences in font availability across browsers, browser versions, languages, and platforms can affect the font display for your report. There are a variety of factors that you must understand to ensure consistent report display.

Embedded fonts

You can ensure consistent report display by embedding fonts. However, reports with embedded fonts have a larger file size.

Output format

The FlashPaper and PDF output formats handle embedded fonts differently.

FlashPaper FlashPaper always embeds fonts, which ensures that reports always display appropriately.

PDF PDF reports can optionally embed fonts, however, if your report doesn't use embedded fonts, you must ensure that the fonts are available on the client computers.

Font availability on the server computer and the client computer

ColdFusion MX has different requirements for rendering the fonts in a report, depending on where the fonts are located.

Server computer For all formats, the fonts used in a report must reside on the computer that runs ColdFusion MX. ColdFusion MX requires these fonts to render the report accurately. ColdFusion MX automatically locates Acrobat built-in fonts and fonts stored in typical font locations (such as the Windows\fonts directory). However, if your server has additional fonts installed in non-standard locations, you must register them with the ColdFusion MX Administrator so that the `cfdocument` and `cfreport` tags can locate and render PDF and FlashPaper reports.

Client computer If your PDF report does not embed fonts, the fonts reside on the client computer to ensure consistent report display.

Mapping logical fonts to physical fonts

If you are using Java logical fonts, such as serif, sans serif, or monospaced, ColdFusion MX maps these fonts to physical fonts using specifications in the *cf_root*/lib/cffont.properties file (on the multiserver or J2EE configuration, this is the *cf_webapp_root*/WEB-INF/cfusion/lib directory). You can modify these mappings, if necessary. Also, if you are using an operating system whose locale is not English, you can create a locale-specific mapping file by appending *.java-locale-code* to the filename. If ColdFusion MX detects that it is running on a non-English locale, it first checks for a cffont.properties.*java-locale-code* file. For example, on a computer that uses the Chinese locale, name the file cffont.properties.cn. For more information on Java locale codes, see the Sun website.

Tip: The ColdFusion MX install includes a cffont.properties.ja file for the Japanese locale.

This discussion applies to both `cfdocument` and `cfreport`. For more information, see Report Builder online Help.

Creating reports with Crystal Reports (Windows only)

When running on Windows, the `cfreport` tag also supports the execution of reports created using Crystal Reports version 9 or 10.

Note: When you install Crystal Reports, you must select the Enable export to HTML and Enable export to Disk options. These options are not enabled by default, so you must use the Custom Install option.

To invoke a Crystal Report report definition using the cfreport tag:

1. Create a report definition in Crystal Reports.
2. Create a CFM page and add a `cfreport` tag that invokes the Crystal Reports report definition. The following example shows the cfreport tag invoking a Crystal Reports report definition and passing a filter condition:

```
<cfreport report = '/reports/monthlysales.rpt'>
     {Departments.Department} = 'International'
</cfreport>
```

3. Open a browser and display the CFM page.

Tip: ColdFusion MX uses COM to call Craxdrt9.dll for Crystal Reports version 9, and Craxdrt.dll for Crystal Reports version 10. If you have problems with the `cfreport` tag, ensure that these DLLs are registered and, if not, use regsvr32 to register them (the default location for these DLLs is C:\Program Files\Crystal Decisions\Report Designer Component\).

For complete information on defining reports in Crystal Reports, see the Crystal Reports documentation.

CHAPTER 33
Using the Flash Remoting Service

Using the Macromedia Flash Remoting service of Macromedia ColdFusion MX 7, ColdFusion MX developers can work together with Macromedia Flash MX 2004 designers to build dynamic Flash user interfaces for ColdFusion MX applications.

This chapter describes how ColdFusion MX interacts with Flash Remoting for Flash MX 2004 ActionScript 2.0. For a complete description of Flash Remoting capabilities, including how ColdFusion MX interacts with Flash Remoting, see *Using Flash Remoting MX 2004* and *Flash Remoting ActionScript Dictionary* in Flash Help. You can also access the Flash Remoting documentation on the Flash Remoting Developer Center at www.macromedia.com/devnet/mx/flashremoting.

Contents

About using the Flash Remoting service with ColdFusion 807

Configuring the Flash Remoting Gateway 809

Using the Flash Remoting service with ColdFusion pages 812

Using Flash with CFCs 817

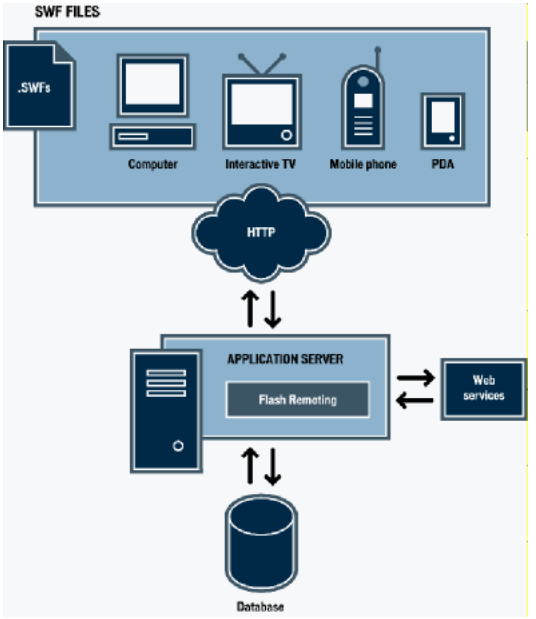
Using the Flash Remoting service with ColdFusion Java objects. 819

Handling errors with ColdFusion and Flash. 820

About using the Flash Remoting service with ColdFusion

Using the Flash Remoting service of ColdFusion MX, ColdFusion developers can work together with Flash MX 2004 designers to build Flash user interfaces (UIs) for ColdFusion applications. Building Flash UIs requires the separation of UI code from business logic code. You build user interface controls in Flash MX, and you build the business logic in ColdFusion MX.

The following is a simplified representation of the relationship between Flash and ColdFusion:



Planning your Flash application

When you are planning ColdFusion MX application development with Flash UIs, remember the importance of separating display code from business logic. Separating display code from business logic enables your ColdFusion applications to interact with multiple client types, such as Flash applications, web browsers, and web services.

When you build ColdFusion MX applications for multiple clients, your ColdFusion pages and components return common data types, including strings, integers, query objects, structures, and arrays. Clients that receive the results can process the passed data according to the client type, such as ActionScript with Flash, or CFML with ColdFusion MX.

To use the Macromedia Flash Remoting service with Macromedia ColdFusion MX, you build ColdFusion pages and components or deploy Java objects. In ColdFusion pages, you use the Flash variable scope to interact with Flash applications. ColdFusion components (CFCs) natively support Flash interaction. The public methods of Java objects are also available to the Flash Remoting service.

The Flash Remoting ActionScript API has been updated to comply with ActionScript 2.0. The ActionScript 2.0 version of the API consists of the following significant features:

Flash Remoting MX 2004 ActionScript 2.0 API Features
Enforcement of strict data typing, which requires you to declare the data types of variables and prohibits you from assigning different types of data to them.
Enforcement of case sensitivity, which means that `myvar` and `myVar` are two different variables, though they were considered the same variable with different spellings in ActionScript 1.0.
A new Service class, which lets you create a gateway connection and at the same time obtain a reference to a service and its methods. It includes the `connection` property, which returns the connection and also lets you set credentials for authorization on the remote server. ***Note:*** The NetServices class is still supported but has been deprecated in favor of the new Service and Connection classes
A new Connection class that helps you create and use Flash Remoting connections. ***Note:*** The Connection class supersedes the former NetConnection class.
A new PendingCall object returned on each call to a service method that is invoked using the Service object. The PendingCall object contains the responder property, which you use to specify the methods to handle the results of the service call.
A new RelayResponder class, which specifies the methods to which the result and fault outcomes of a service call are relayed.
A RecordSet object that contains new properties (`columnNames`, `items`, and `length`), new methods (`clear()`, `contains()`, `editField()`, `getEditingData()`, `getIterator()`, `getLocalLength()`, `getRemoteLength()`, `isEmpty()`, and `sortItems()`), and the new `modelChanged` event.

For more information on the ActionScript 2.0 Flash Remoting API, see Flash Remoting ActionScript Dictionary Help.

The remaining sections in this chapter explain how ColdFusion MX interacts with Flash Remoting applications.

Configuring the Flash Remoting Gateway

The following parameters in the ColdFusion web.xml file point the Flash Remoting gateway to the gateway-config.xml file.

```
<init-param>
  <param-name>gateway.configuration.file</param-name>
  <param-value>/WEB-INF/gateway-config.xml</param-value>
    </init-param>
<init-param>
  <param-name>whitelist.configuration.file</param-name>
  <param-value>/WEB-INF/gateway-config.xml</param-value>
</init-param>
<init-param>
  <param-name>whitelist.parent.node</param-name>
  <param-value>gateway-config</param-value>
</init-param>
```

Both the web.xml file and the gateway-config.xml file are located in the WEB-INF directory of your ColdFusion server. As a general rule, there is no need to change these web.xml settings.

ColdFusion MX 7 configures Flash gateways differently from previous ColdFusion MX releases. Parameters that worked prior to this release are no longer supported, and you specify all configuration parameters in the gateway-config.xml file. Also, the Flash gateway now supports a whitelist, which specifies which remote sources can be accessed through the gateway. The two web.xml entries that identify the whitelist should specify your gateway-config.xml file and gateway-config as the parent node.

You can modify the gateway-config.xml file to configure service adapters, add service names to the whitelist, change the logging level, and specify how the gateway handles case sensitivity.

You can configure gateway features in the gateway-config.xml file as follows:

Feature	Description

service adapters	By default, the PageableResultSetAdapter, the ColdFusionAdapter, the CFCAdapter (for ColdFusion components), and the CFSSASAdapter (for server-side ActionScript) adapters are enabled in ColdFusion.
	You can also enable the JavaBeanAdapter, JavaAdapter, EJBAdapter, ServletAdapter, and CFWSAdapter (for web services) by removing their enclosing comment symbols (<!-- and -->). The following service adapter tags are defined as the default tag values:
	`<service-adapters>`
	`  <adapter>flashgateway.adapter.resultset.PageableResultSetAdapter`
	`  </adapter>`
	`  <adapter>coldfusion.flash.adapter.ColdFusionAdapter</adapter>`
	`  <adapter>coldfusion.flash.adapter.CFCAdapter</adapter>`
	`  <adapter>coldfusion.flash.adapter.CFSSASAdapter</adapter>`
	`  <!-- <adapter type="stateful-class">flashgateway.adapter.java.`
	`  JavaBeanAdapter</adapter> -->`
	`  <!-- <adapter type="stateless-class">flashgateway.adapter.java.`
	`  JavaAdapter</adapter> -->`
	`  <!-- <adapter type="ejb">flashgateway.adapter.java.EJBAdapter`
	`  </adapter> -->`
	`  <!-- <adapter type="servlet">flashgateway.adapter.java.ServletAdapter`
	`  </adapter> -->`
	`  <!-- <adapter>coldfusion.flash.adapter.CFWSAdapter</adapter> -->`
	`</service-adapters>`

Feature	Description

security	You can edit security settings in child tags of the `<security>` tag.
	In the `<login-command>` tag, you can set the `flashgateway.security.LoginCommand` implementation for performing local authentication on a specific application server. By default, the ‹login-command› tag is set to the following values:
	`<login-command>`
	`  <class>flashgateway.security.JRunLoginCommand</class>`
	`  <server-match>JRun</server-match>`
	`</login-command>`
	
	In the `<show-stacktraces>` tag, you can enable stack traces. Stack traces are useful for debugging and product support, but they should not be sent to the client in production mode because they can expose internal information about the system. The following `<show-stacktraces>` tag is the default tag:
	`<show-stacktraces>false</show-stacktraces>`
	
	The `<whitelist>` tag specifies which remote sources can be accessed through the gateway. The * character can be used as a wildcard to imply ALL matches. The following `<whitelist>` tag shows the default value:
	`<whitelist>`
	`  <source>*</source>`
	`</whitelist>`
	When you deploy your application, ensure that you change this setting so that it specifies only the services that the gateway needs to access to run your application. Remember that for ColdFusion based services, directories are treated as "packages" and thus you specify a period delimited path from the web root to the directory or file containing the services you will allow access to. An asterisk wildcard allows access to all services in a particular directory. You can have multiple ‹source› tags.
	The following whitelist allows access to the *webroot*/cfdocs/exampleapps/ directory, which includes the flash1 through flash5 Flash Remoting example directories. It also allows access to a *webroot*/BigApp/remoting directory and its children.
	`<whitelist>`
	`  <source>cfdocs.exampleapps.*</source>`
	`  <source>BigApp.remoting.*</source>`
	`</whitelist>`

logger level	You can set the level of logging between `None`, `Error`, `Info`, `Warning`, and `Debug`. The following tag is the default logger level tag: `<logger level="Error">coldfusion.flash.ColdFusionLogger</logger>`
redirect URL	In the `<redirect-url>` tag, you can specify a URL to receive HTTP requests that are not sent with AMF data. By default, the `<redirect-url>` tag is set to `{context.root}`, which is the context root of the web application: `<redirect-url>{context.root}</redirect-url>`
case sensitivity	The `<lowercase-keys>` tag specifies how the gateway handles case sensitivity. ActionScript 1.0 and ColdFusion use case insensitive data structures to store associative arrays, objects and structs. The Java representation of these data types requires a case-insensitive Map, which the gateway achieves by forcing all keys to lowercase. ActionScript 2.0 is case sensitive and requires a ‹lowercase-keys› tag value of `false`. The following `<lowercase-keys>` tag is the default tag: `<lowercase-keys>true</lowercase-keys>`

Using the Flash Remoting service with ColdFusion pages

When you build a ColdFusion page that interacts with a Flash application, the directory name that contains the ColdFusion pages translates to the service name that you call in ActionScript. The individual ColdFusion page names within that directory translate to service functions that you call in ActionScript.

Note: Flash Remoting cannot interact with virtual directories accessed through a ColdFusion mapping.

In your ColdFusion pages, you use the Flash variable scope to access parameters passed to and from a Flash application. To access parameters passed from a Flash application, you use the parameter name appended to the `Flash` scope or the `Flash.Params` array. To return values to the Flash application, use the `Flash.Result` variable. To set an increment value for records in a query object to be returned to the Flash application, use the `Flash.Pagesize` variable.

The following table shows the variables contained in the Flash scope:

Variable	Description	For more information
Flash.Params	Array that contains the parameters passed from the Flash application. If you do not pass any parameters, `Flash.params` still exists, but it is empty.	See "Accessing parameters passed from Flash" on page 813.
Flash.Result	The variable returned from the ColdFusion page to the Flash application that called the function. `Note:` Because ActionScript performs automatic type conversion, do not return a Boolean literal to Flash from ColdFusion. Return `1` to indicate `true`, and return `0` to indicate `false`.	See "Returning results to Flash" on page 815.
Flash.Pagesize	The number of records returned in each increment of a record set to a Flash application.	See "Returning records in increments to Flash" on page 816.

The following table compares the ColdFusion MX data types and their ActionScript equivalents:

ActionScript data type	ColdFusion MX data type
Number (primitive data type)	Number
Boolean (primitive data type)	Boolean (0 or 1)
String (primitive data type)	String
ActionScript Object	Structure
ActionScript Object (as the only argument passed to a service function)	Arguments of the service function. ColdFusion pages (CFM files): `flash` variable scope, ColdFusion components (CFC files): named arguments
Null	Null (`Asc()` returns 0, which translates to not defined)

ActionScript data type	ColdFusion MX data type
Undefined	Null (`Asc()` returns 0, which translates to not defined)
Ordered array **Note:** ActionScript array indexes start at zero (for example: my_ASarray[0]).	Array **Note:** ColdFusion array indexes start at one (for example: my_CFarray[1]).
Named (or associative) array	Struct
Date object	Date
XML object	XML document
RecordSet	Query object (when returned to a Flash application only; you cannot pass a RecordSet from a Flash application to a ColdFusion MX application)

Also, remember the following considerations regarding data types:

- If a string data type on the server represents a valid number in ActionScript, Flash can automatically cast it to a number if needed.
- To return multiple, independent values to the Flash application, place them in a complex variable that converts to a Flash Object, Array, or Associative Array, that can hold all of the required data. Return the single variable and access its elements in the Flash application.

For a complete explanation of using Flash Remoting data in ActionScript, see Using Flash Remoting MX 2004 Help.

Accessing parameters passed from Flash

To access variables passed from Flash applications, you append the parameter name to the Flash scope or use the `Flash.Params` array. Depending on how the values were passed from Flash, you refer to array values using ordered array syntax or structure name syntax. Only ActionScript objects can pass named parameters.

For example, if you pass the parameters as an ordered array from Flash, `array[1]` references the first value. If you pass the parameters as named parameters, you use standard structure-name syntax like `params.name`.

You can use most of the CFML array and structure functions on ActionScript collections. However, the `StructCopy` CFML function does not work with ActionScript collections. The following table lists ActionScript collections and describes how to access them in ColdFusion MX:

Collection	ActionScript example	Notes

Strict array	`var myArray:Array = new Array();`
	`myArray[0] = "zero";`
	`myArray[1] = "one";`
	`myService.myMethod(myArray);`
		The Flash Remoting service converts the Array argument to a ColdFusion MX array. All CFML array operations work as expected.
		`<cfset p1=Flash.Params[1][1]>`
		`<cfset p2=Flash.Params[1][2]>`

Named or associative array	`var myStruct:Array = new Array();`
	`myStruct["zero"] = "banana";`
	`myStruct["one"] = "orange";`
	`myService.myMethod(myStruct);`
		Named array keys are not case-sensitive in ActionScript.
		`<cfset p1=Flash.Params[1].zero>`
		`<cfset p2=Flash.Params[1].one>`

Mixed array	`var myMxdArray:Array = new Array();`
	`myMxdArray["one"] = 1;`
	`myMxdArray[2] = true;`
		Treat this collection like a structure in ColdFusion MX. However, keys that start with numbers are invalid CFML variable names. Depending on how you attempt to retrieve this data, ColdFusion MX might throw an exception. For example, the following CFC method throws an exception:
		`<cfargument name="ca" type="struct">`
		`<cfreturn ca.2>`
		
		The following CFC method does not throw an exception:
		`<cfargument name="ca" type="struct">`
		`<cfreturn ca["2"]>`

Using an ActionScript object initializer for named arguments	`myService.myMethod`
	`({ x:1, Y:2, z:3 });`
		This notation provides a convenient way of passing named arguments to ColdFusion pages. You can access these arguments in ColdFusion pages as members of the Flash scope:
		`<cfset p1 = Flash.x>`
		`<cfset p2 = Flash.y>`
		`<cfset p3 = Flash.z>`
		Or, you can access them as normal named arguments of a CFC method.

The `Flash.Params` array retains the order of the parameters as they were passed to the method. You use standard structure name syntax to reference the parameters; for example:

```
<cfquery name="flashQuery" datasource="cfdocexamples">
  SELECT ItemName, ItemDescription, ItemCost
  FROM tblItems
  WHERE ItemName EQ '#Flash.paramName#'
</cfquery>
```

In this example, the query results are filtered by the value of `Flash.paramName`, which references the first parameter in the passed array. If the parameters were passed as an ordered array from the Flash application, you use standard structure name syntax; for example:

```
<cfset Flash.Result = "Variable 1:#Flash.Params[1]#, Variable 2:
  #Flash.Params[2]#">
```

Tip: ActionScript array indexes start at zero. ColdFusion array indexes start at one.

Returning results to Flash

In ColdFusion pages, only the value of the `Flash.Result` variable is returned to the Flash application. For more information about supported data types between ColdFusion MX and Flash, see the data type table in "Using the Flash Remoting service with ColdFusion pages" on page 812. The following procedure creates the service function `helloWorld`, which returns a structure that contains simple messages to the Flash application.

To create a ColdFusion page that passes a structure to a Flash application:

1. Create a folder in your web root, and name it helloExamples.
2. Create a ColdFusion page, and save it as helloWorld.cfm in the helloExamples directory.
3. Modify helloWorld.cfm so that the CFML code appears as follows:

   ```
   <cfset tempStruct = StructNew()>
   <cfset tempStruct.timeVar = DateFormat(Now ())>
   <cfset tempStruct.helloMessage = "Hello World">
   ```

   In the example, two string variables are added to a structure; one with a formatted date and one with a simple message. The structure is passed back to the Flash application using the `Flash.Result` variable.

   ```
   <cfset Flash.Result = tempStruct>
   ```

4. Save the file.

Remember, the directory name is the service address. The helloWorld.cfm file is a method of the `helloExamples` Flash Remoting service. The following ActionScript example calls the helloWorld ColdFusion page and displays the values that it returns:

```
import mx.remoting.*;
import mx.services.Log;
import mx.rpc.*;

// Connect to helloExamples service and create the howdyService service object
  var howdyService:Service = new Service(
       "http://localhost/flashservices/gateway",
       null,
       "helloExamples",
       null,
       null );
// Call the service helloWorld() method
var pc:PendingCall = howdyService.helloWorld();
// Tell the service what methods handle result and fault conditions
pc.responder = new RelayResponder( this, "helloWorld_Result",
  "helloWorld_Fault" );
```

```
function helloWorld_Result(re:ResultEvent)
{
  // Display successful result
  messageDisplay.text = re.result.HELLOMESSAGE;
  timeDisplay.text = re.result.TIMEVAR;
}

function helloWorld_Fault(fe:FaultEvent)
{
  // Display fault returned from service
  messageDisplay.text = fe.fault;
}
```

Note: Due to ActionScript's automatic type conversion, do not return a Boolean literal to Flash from ColdFusion. Return `1` to indicate `true`, and return `0` to indicate `false`.

Returning records in increments to Flash

ColdFusion MX lets you return record set results to Flash in increments. For example, if a query returns 20 records, you can set the `Flash.Pagesize` variable to return five records at a time to Flash. Incremental record sets let you minimize the time that a Flash application waits for the application server data to load.

To create a ColdFusion page that returns a incremental record set to Flash:

1. Create a ColdFusion page, and save it as getData.cfm in the helloExamples directory.
2. Modify getData.cfm so that the code appears as follows:

```
<cfparam name="pagesize" default="10">
<cfif IsDefined("Flash.Params")>
  <cfset pagesize = Flash.Params[1]>
</cfif>
<cfquery name="myQuery" datasource="cfdocexamples">
  SELECT *
  FROM tblParks
</cfquery>
<cfset Flash.Pagesize = pagesize>
<cfset Flash.Result = myQuery>
```

   In this example, if a single parameter is passed from the Flash application, the `pagesize` variable is set to the value of the `Flash.Params[1]` variable; otherwise, the value of the variable is the default, 10. Next, a `cfquery` statement queries the database. After that, the `pagesize` variable is assigned to the `Flash.Pagesize` variable. Finally, the query results are assigned to the `Flash.Result` variable, which is returned to the Flash application.
3. Save the file.

When you assign a value to the `Flash.Pagesize` variable, you are specifying that if the record set has more than a certain number of records, the record set becomes pageable and returns the number of records specified in the `Flash.Pagesize` variable. For example, the following code calls the `getData()` function of the CFMService and uses the first parameter to request a page size of 5:

```
import mx.remoting.*;
import mx.services.Log;
import mx.rpc.*;
// Connect to helloExamples service and create the CFMService service object
  var CFMService:Service = new Service(
       "http://localhost/flashservices/gateway",
       null,
       "helloExamples",
       null,
       null );
// Call the service getData() method
var pc:PendingCall = CFMService.getData(5);
// Tell the service what methods handle result and fault conditions
pc.responder = new RelayResponder( this, "getData_Result", "getData_Fault" );

function getData_Result(re:ResultEvent)
{
  // Display successful result
  DataGlue.bindFormatStrings(employeeData, re.result, "#PARKNAME#, #CITY#,
  #STATE#");
}
function getData_Fault(fe:FaultEvent)
{
  // Display fault returned from service
  trace("Error description from server: " + fe.fault.description);
}
```

In this example, `employeeData` is a Flash list box. The result handler, `getData_Result`, displays the columns PARKNAME, CITY, and STATE in the `employeeData` list box. After the initial delivery of records, the RecordSet ActionScript class assumes the task of fetching records. In this case, the list box requests more records as the user scrolls the list box.

You can configure the client-side RecordSet object to fetch records in various ways using the `ResordSet.setDeliveryMode` ActionScript function.

Using Flash with CFCs

CFCs require little modification to work with a Flash application. The `cffunction` tag names the method and contains the CFML logic, the `cfargument` tag names the arguments, and the `cfreturn` tag returns the result to the Flash application. The name of the CFC file (*.cfc) translates to the service name in ActionScript.

Note: For CFC methods to communicate with Flash applications, you must set the `cffunction` tag's `access` attribute to `remote`.

The following example replicates the helloWorld function that was previously implemented as a ColdFusion page. For more information, see "Using the Flash Remoting service with ColdFusion pages" on page 812.

To create a CFC that interacts with a Flash application:

1. Create a CFC and save it as flashComponent.cfc in the helloExamples directory.
2. Modify the code in flashComponent.cfc so that it appears as follows:

```
<cfcomponent name="flashComponent">
  <cffunction name="helloWorld" access="remote" returnType="Struct">
    <cfset tempStruct = StructNew()>
    <cfset tempStruct.timeVar = DateFormat(Now ())>
    <cfset tempStruct.helloMessage = "Hello World">
    <cfreturn tempStruct>
  </cffunction>
</cfcomponent>
```

   In this example, the helloWorld function is created. The cfreturn tag returns the result to the Flash application.

3. Save the file.

The helloWorld service function is now available through the flashComponent service to ActionScript. The following ActionScript example calls this function:

```
import mx.remoting.*;
import mx.services.Log;
import mx.rpc.*;

// Connect to the Flash component service and create service object
  var CFCService:Service = new Service(
       "http://localhost/flashservices/gateway",
       null,
       "helloExamples.flashComponent",
       null,
       null );
// Call the service helloWorld() method
var pc:PendingCall = CFCService.helloWorld();
// Tell the service what methods handle result and fault conditions
pc.responder = new RelayResponder( this, "helloWorld_Result",
  "helloWorld_Fault" );

function helloWorld_Result(re:ResultEvent)
{
  // Display successful result
  messageDisplay.text = re.result.HELLOMESSAGE;
  timeDisplay.text = re.result.TIMEVAR;
}

function helloWorld_Fault(fe:FaultEvent)
{
  // Display fault returned from service
  messageDisplay.text = fe.fault;
}
```

In this example, the `CFCService` object references the `flashComponent` component in the helloExamples directory. Calling the `helloWorld` function in this example executes the function that is defined in `flashComponent`.

For ColdFusion components, the component file name, including the directory structure from the web root, serves as the service name. Remember to delimit the path directories with periods rather than backslashes.

Using the Flash Remoting service with ColdFusion Java objects

You can run various kinds of Java objects with ColdFusion MX, including JavaBeans, Java classes, and Enterprise JavaBeans. You can use the ColdFusion MX Administrator to add additional directories to the classpath.

To add a directory to ColdFusion classpath:

1. Open the ColdFusion MX Administrator.
2. In the Server Settings menu, click the Java and JVM link.
3. Add your directory to the Class Path form field.
4. Click Submit Changes.
5. Restart ColdFusion.

When you place your Java files in the classpath, the public methods of the class instance are available to your Flash movie.

For example, assume the Java class `utils.UIComponents` exists in a directory in your ColdFusion classpath. The Java file contains the following code:

```
package utils;
public class UIComponents
{
  public UIComponents()
  {
  }
  public String sayHello()
  {
    return "Hello";
  }
}
```

Note: You cannot call constructors with Flash Remoting. You must use the default constructor.

In ActionScript, the following `javaService` call invokes the `sayHello` public method of the `utils.UIComponents` class:

```
import mx.remoting.*;
import mx.services.Log;
import mx.rpc.*;

// Connect to service and create service object
  var javaService:Service = new Service(
       "http://localhost/flashservices/gateway",
       null,
```

```
        utils.UIComponents",
        null,
        null );
// Call the service sayHello() method
var pc:PendingCall = javaService.sayHello();
// Tell the service what methods handle result and fault conditions
pc.responder = new RelayResponder( this, "sayHello_Result",
  "sayHello_Fault" );

function sayHello_Result(re:ResultEvent)
{
  // Display successful result
  trace("Result is: " + re.result);
}

function sayHello_Fault(fe:FaultEvent)
{
  // Display fault returned from service
  trace("Error is: " + fe.fault.description);
}
```

Note: For more information about using Java objects with ColdFusion, see Chapter 37, "Using Java objects," on page 928.

Handling errors with ColdFusion and Flash

To help with debugging, use the `cftry` and `cfcatch` tags in your ColdFusion page or component to return error messages to the Flash Player. For example, the ColdFusion page, causeError.cfm, contains the code:

```
<cftry>
  <cfset dev = Val(0)>
  <cfset Flash.Result = (1 / dev)>
  <cfcatch type = "any">
    <cfthrow message = "An error occurred in this service: #cfcatch.message#">
  </cfcatch>
</cftry>
```

The second `cfset` tag in this example fails because it tries to divide by zero (0). The `message` attribute of the `cfthrow` tag describes the error; ColdFusion returns this attribute to the Flash application.

To handle the error in your Flash application, create a fault handler similar to `causeError_Fault` in the following example:

```
import mx.remoting.*;
import mx.services.Log;
import mx.rpc.*;

// Connect to service and create service object
  var CFMService:Service = new Service(
       "http://localhost/flashservices/gateway",
       null,
       "helloExamples",
       null,
```

```
       null );
// Call the service causeError() method
var pc:PendingCall = CFMService.causeError();
// Tell the service what methods handle result and fault conditions
pc.responder = new RelayResponder( this, "causeError_Result",
  "causeError_Fault" );

function causeError_Result(re:ResultEvent)
{
  // Display successful result
  messageDisplay.text = re.result;
}

function causeError_Fault(fe:FaultEvent)
{
  // Display fault returned from service
  trace("Error message from causeError is: " + fe.fault.description);
}
```

This example displays the trace message from the `causeError_Fault` function in the Flash Output panel. The portion of the message that is contained in `fe.fault.description` is the portion of the message that is contained in `#cfcatch.message#` in the causeError.cfm page.

Note: When you create a ColdFusion page that communicates with Flash, ensure that the ColdFusion page works before using it with Flash.

CHAPTER 34
Using Server-Side ActionScript

Macromedia ColdFusion MX server configuration includes the Macromedia Flash Remoting service, a module that lets Macromedia Flash MX developers create server-side ActionScript. These ActionScript files can directly access ColdFusion MX query and HTTP features through two new ActionScript functions: `CF.query` and `CF.http`.

Contents

About server-side ActionScript 823
Connecting to the Flash Remoting service 826
Using server-side ActionScript functions 827
Global and request scope objects 829
About the CF.query function and data sources 829
Using the CF.query function 831
Building a simple application 833
About the CF.http function 836
Using the CF.http function 837

About server-side ActionScript

ColdFusion MX includes a module called the Flash Remoting service that acts as a broker for interactions between Flash MX and ColdFusion MX. Flash Remoting supports a range of object types, and lets you reference an ActionScript file that lives on a ColdFusion server. You can partition data-intensive operations on the server, while limiting the amount of network transactions necessary to get data from the server to the client.

Flash developers can create server-side ActionScript files to access ColdFusion MX resources; they do not have to learn CFML (ColdFusion Markup Language). This ability lets you logically separate the Flash presentation elements of your applications from the business logic. You have the option of creating ActionScript files that reside on the server to partition this processing away from your client applications.

You have a very simple interface for building queries using server-side ActionScript, and an equally simple interface for invoking these queries from your client-side ActionScript.

Client-side ActionScript requirements

On the client side, you only need a small piece of code that establishes a connection to the Flash Remoting service and references the server-side ActionScript you want to use.

For example (notice the embedded comments):

```
// This #include is needed to connect to the Flash Remoting service
#include "NetServices.as"

// This line determines where Flash MX should look for the Flash Remoting
  service.
// Ordinarily, you enter the URL to your ColdFusion server.
// Port 8500 is the Flash Remoting service default.
NetServices.setDefaultGatewayUrl("http://mycfserver:8500");

// With the Flash Remoting service URL defined, you can create a connection.
gatewayConnnection = NetServices.createGatewayConnection();

// Reference the server-side ActionScript.
// In this case, the stockquotes script file lives in the web root of the
// ColdFusion server identified previously. If it lived in a subdirectory
// of the web root called "mydir," you would reference it
// as "mydir.stockquotes".
stockService = gatewayConnnection.getService("stockquotes", this);

// This line invokes the getQuotes() method defined in the stockquotes
// server-side ActionScript.
stockService.getQuotes("macr");

// Once the record set is returned, you handle the results.
// This part is up to you.
function getQuotes_Result ( result )
{
  // Do something with results
}
```

Note: Client-side ActionScript does not support the two new server-side ActionScript functions, `CF.query` and `CF.http`.

Server-side requirements

Creating ActionScript that executes on the server helps leverage your knowledge of ActionScript. It also provides direct access to ColdFusion MX query and HTTP features. The `CF.query` and `CF.http` ActionScript functions let you perform ColdFusion MX HTTP and query operations.

Note: On the server side, ActionScript files use the extension .asr.

For example, the following server-side ActionScript code builds on the client-side code shown previously:

```
// Filename: stockquotes.asr
// Here is the getQuotes method invoked in the client-side ActionScript.
// It accepts a single stock quote symbol argument.
function getQuotes(symbol)
{
  // Query some provider for the specified stock quote and return the
  // results. In this case, the getQuotesFromProvider method is
  // defined elsewhere in this ActionScript code.
  data = getQuotesFromProvider(symbol);
  // Return the data to the client.
  // Note: this example does not include any of the error checking
  // logic you would normally use prior to returning the data.
  return data;
}
```

The `getQuotes` function conducts the stock quote request and returns the results of the request to the client as a RecordSet object.

Software requirements

To use server-side ActionScript files, you must have the following software installed:

- Macromedia Flash MX
- Macromedia ColdFusion MX
- Flash Remoting Components

For more information about these products, go to www.macromedia.com.

Location of server-side ActionScript files

You can place ActionScript files (`*.asr`) on the server anywhere below the web server's root directory. To specify subdirectories of the web root or a virtual directory, use package dot notation (use dots instead of slashes in a fully qualified directory name). For example, in the following assignment code, the stockquotes.asr file is located in the mydir/stock/ directory:

```
stockService = gatewayConnnection.getService("mydir.stock.stockquotes", this);
```

You can also point to virtual mappings, such as `cfsuite.asr.stock.stockquotes` where `cfsuite` is a virtual mapping and `asr.stock` is subdirectories of that mapping.

Benefits

Server-side ActionScript lets your ActionScript engineers use their knowledge of ActionScript to write code for the back end of their Flash applications, which can mean more meaningful levels of interactivity for your users. Your Flash applications can share a library of server-side ActionScript functions, which means you can define functions that are specifically tailored to your own business.

You could, for example, create a server-side ActionScript file that defines a whole library of SQL query methods. With these query methods defined on the server side, your Flash designers only have to invoke the specific query function they want to return data to their Flash MX movies. They do not have to write any SQL, and they do not have to create a new query every time they need to retrieve data from a ColdFusion MX data source. It is a way of creating reusable queries that your entire Flash design team can use.

Coding the ColdFusion MX query and HTTP operations in ActionScript is very straightforward. The `CF.query` and `CF.http` functions provide a well-defined interface for building SQL queries and HTTP operations.

For example, the following is a typical server-side ActionScript function definition that returns query data:

```
// This function shows a basic CF.query operation using only
// arguments for data source name and for SQL.
function basicQuery()
{
  mydata = CF.query({datasource:"customers",
    sql:"SELECT * FROM myTable"});
  return mydata;
}
```

What to do next

If you are already familiar with ActionScript, you only need to know a few things to get started:

- How to establish a connection with the Flash Remoting service using client-side ActionScript. See "Connecting to the Flash Remoting service" on page 826
- How to reference server-side ActionScript functions and methods. See "Using server-side ActionScript functions" on page 827.
- How to code the server-side `CF.query` and `CF.http` functions. See "Using the CF.query function" on page 831 and "Using the CF.http function" on page 837. Also see the reference pages for these functions in the *CFML Reference*.

For additional information on using Flash Remoting MX, see Chapter 33, "Using the Flash Remoting Service," on page 807, and *Using Flash Remoting MX.*

Connecting to the Flash Remoting service

Before you can use functions defined in your server-side ActionScript files, you must connect the Macromedia Flash MX movie to the server-side Flash Remoting service.

To create a Flash Remoting service connection:

1. Include the necessary ActionScript classes in the first frame of the Flash movie that will be using server-side ActionScript functions.

   a Use the following command to include the `NetServices` class:

   ```
   #include "NetServices.as"
   ```

   b (Optional) Use the following command to include the `NetDebug` class:

   ```
   #include "NetDebug.as"
   ```

   For more information about the `NetDebug` and `RecordSet` classes, see *Using Flash Remoting MX*.

2. Since the Flash Remoting service serves as a broker for calls to server-side ActionScript functions, you must identify the Flash Remoting service URL as an argument in the `NetServices.setDefaultGatewayUrl` function. For example:

   ```
   NetServices.setDefaultGatewayURL("http://localhost:8500/flashservices")
   ```

   You must specify a server hostname. The default port number for the Flash Remoting service is 8500.

3. Create the gateway connection using the `NetServices.createGatewayConnection` function; for example:

   ```
   gatewayConnection = NetServices.createGatewayConnection();
   ```

Using server-side ActionScript functions

After you connect to the Flash Remoting service, you call functions that are defined in your server-side ActionScript files, and return results.

To call a function:

1. Create an instance of the server-side ActionScript file using the `getService` function. This function instantiates the server-side ActionScript file as an object to be used on the client side. For example:

   ```
   albumService = gatewayConnection.getService("recordsettest", this)
   ```

   Where `recordsettest` represents the name of the server-side ActionScript file, without the file extension `.asr`.

2. Call a function defined in your server-side ActionScript object. Use dot notation to specify the object name followed by the function name; for example:

   ```
   albumService.getAlbum("The Color And The Shape", "1999");
   ```

   Where `albumService` is the instance of the server-side ActionScript file and `getAlbum` is a function that passes two arguments, `"The Color and The Shape"` and `"1999"`.

   Note: Arguments must occur in the order defined in the function declaration.

3. Handle the function results in ActionScript. See "Using the function results in ActionScript" on page 828.

Using the function results in ActionScript

To use the results returned by server-side ActionScript, you must create a corresponding *results function*. The results function uses a special naming convention that ties it to the function that calls the server-side ActionScript. For example, if you defined a client-side ActionScript function called `basicCustomerQuery`, you also must create a results function called `basicCustomerQuery_Result`.

The results returned by server-side ActionScript functions differ somewhat depending on whether you are using `CF.http` or `CF.query`:

- The `CF.query` function returns a record set, which you manipulate using methods available in the RecordSet ActionScript class object. See "Using results returned by the CF.query function" on page 828.
- The `CF.http` function returns simple text strings through properties that you reference in your server-side ActionScript. See "Using results returned by the CF.http function" on page 828.

Using results returned by the CF.query function

You use functions in the RecordSet ActionScript object to access the data returned in a `CF.query` record set; for example, how many records are in the record set and the names of the columns. You can also use the RecordSet functions to pull the query data out of the record set. To do so, you reference a specific row number in the record set and use the `getItemAt` RecordSet function, as in the following example:

```
// This function populates a Flash text box with data in the first row
// of the record set under the "email" column name.
function selectData_Result ( result )
{
  stringOutput.text = result.getItemAt(0)["email"];
  _root.employeesView.setDataProvider(result);
}
```

In the example, the column name is referenced in the `getItemAt` function between square brackets [ ]. (In ActionScript, indexes start at 0, so `getItemAt(0)` returns the first row.)

For more information, see "Using the CF.query function" on page 831.

Using results returned by the CF.http function

The `CF.http` server-side ActionScript function returns data as simple text. You write server-side functions that reference the properties available in the object returned by the `CF.http` function. These properties store the file content of the retrieved file, HTTP status codes, the MIME type of the returned file, and so on. On the client side, you create return functions to handle data returned by the `CF.http` function. You write these functions to handle simple text data.

For more information, see "Using the CF.http function" on page 837.

Global and request scope objects

Global and request scope objects are implicitly available in all server-side ActionScript. The following table describes these scope objects:

Scope name	Type	Description
config	Global	Initialization information for the server-side ActionScript adapter. Class: `javax.servlet.ServletConfig`
application	Global	The context for the current web application. The context defines methods that provide, for example, the MIME type of a file that can be used to write to a log file. There is one context per web application. Class: `javax.servlet.ServletContext`
request	Request	An object containing client request information. The object provides data, including parameter name and values, attributes, and an input stream. Class: `HttpServletRequest` (subtype of `javax.servlet.ServletRequest`)
response	Request	An object to assist in sending a response to the client. It provides HTTP-specific functionality in sending a response. Do not use the OutputStream or PrintWriter to send data back to the client. Class: `HttpServletResponse` (subtype of `javax.servlet.ServletResponse`)

For more information about these scope objects, see the documentation on the javax.servlet class at http://java.sun.com.

About the CF.query function and data sources

You use the `CF.query` function to populate Macromedia Flash MX movie elements with data retrieved from a Macromedia ColdFusion MX data source. To use the `CF.query` function you do the following:

To pull data into your Flash MX movie from a ColdFusion MX data source:

1. Create a server-side ActionScript file that performs queries against a ColdFusion MX data source.
2. Write ActionScript code in your Flash MX movie that references your ActionScript file (.asr) on the ColdFusion server.

You create server-side ActionScript to execute the query and return the data in a record set to the client—your Flash MX movie. You can use methods in the RecordSet ActionScript object on the client to manipulate data in the record set and present data in your Flash MX movie.

Note: Client-side ActionScript files use the .as extension. Server-side ActionScript files use the .asr (*A*ctionScript *r*emote) extension.

Publishing dynamic data

You use the server-side ActionScript feature in ColdFusion MX to publish dynamic data. To do this, you write server-side ActionScript files that perform queries against ColdFusion MX data sources. Before using ActionScript, you must understand how to do the following:

- Create database queries in the server-side ActionScript file using the `CF.query` ActionScript function. See "Using the CF.query function" on page 831.
- Reference the server-side ActionScript file in your Flash MX movie. See "Connecting to the Flash Remoting service" on page 826.

Using the `CF.query` function, you can do the following tasks:

- Create user login interfaces that validate users against a ColdFusion MX data source.
- Populate form elements and data grids with data from a ColdFusion MXdata source.
- Create banners that pull data (such as URLs or image file paths) out of a database.

The `CF.query` function can retrieve data from any supported ColdFusion MX data source (see "About ColdFusion MX data sources" on page 830).

About ColdFusion MX data sources

For ColdFusion MX developers, the term *data source* can refer to a number of different types of structured data accessible locally or across a network. You can query websites, Lightweight Directory Access Protocol (LDAP) servers, POP mail servers, and documents in a variety of formats. For server-side ActionScript, a data source ordinarily means the entry point to a ColdFusion MX database.

Your ColdFusion MX administrator can help you identify and configure data sources. To create ActionScript files that successfully perform queries on ColdFusion MX data sources, you must know how the data source is identified by ColdFusion MX, as well as any other parameters that affect your ability to connect to that database, such as whether a username and password are required to connect.

You use server-side ActionScript in ColdFusion MX to return record set data to a Flash MX client from a ColdFusion MX data source. You specify the ColdFusion MX data source name and the SQL statement you execute on the data source as arguments in the `CF.query` function in server-side ActionScript.

Typically, your server-side ActionScript handles the interaction with the ColdFusion MX data source, and returns a record set to the Flash MX client through the Flash Remoting service.

For more detailed information about ColdFusion MX data sources, see *Configuring and Administering ColdFusion MX*.

Using the CF.query function

You use the `CF.query` function in your server-side ActionScript to retrieve data from a ColdFusion MX data source. This function lets you perform queries against any ColdFusion MX data source.

Note: The `CF.query` function maps closely to the `cfquery` CFML tag, although it currently supports a subset of the `cfquery` attributes.

Use the `CF.query` function to do the following:

- Identify the data source you want to query.
- Pass SQL statements to the data source.
- Pass other optional parameters to the database.

For reference information about the `CF.query` function, see `CF.query` in *CFML Reference*.

About CF.query function syntax

You can write the `CF.query` ActionScript function using either named arguments or positional arguments. The named argument style is more readable, but it requires more code. Although the positional argument style supports a subset of `CF.query` arguments, it allows a more compact coding style that is more appropriate for simple expressions of the `CF.query` function.

Using CF.query named argument syntax

The `CF.query` function accepts the following named arguments:

```
// CF.query named argument syntax
CF.query
  ({
    datasource:"data source name",
    sql:"SQL stmts",
    username:"username",
    password:"password",
    maxrows:number,
    timeout:milliseconds
  })
```

Note: The named argument style requires curly braces {} to surround the function arguments.

Using CF.query positional argument syntax

Positional arguments support a subset of `CF.query` arguments, and you can create more efficient code. The following is the syntax for the positional argument style:

```
// CF.query positional argument syntax
CF.query(datasource, sql);
CF.query(datasource, sql, maxrows);
CF.query(datasource, sql, username, password);
CF.query(datasource, sql, username, password, maxrows);
```

Note: When using positional arguments, do not use curly braces {}.

About the CF.query record set

The `CF.query` function returns a RecordSet object, which is an instance of the RecordSet class of objects. The RecordSet class provides a wide range of functions for handling record set data.

You use methods in the RecordSet ActionScript class in your client-side ActionScript to change data returned in the `CF.query` record set.

Currently, the following methods are available in the RecordSet class:

Method	Description
addItem	Appends a record to the end of the specified RecordSet
addItemAt	Inserts a record at the specified index
addView	Requests notification of changes in a RecordSet object's state
filter	Creates a new RecordSet object that contains selected records from the original RecordSet object
getColumnNames	Returns the names of all the columns of the RecordSet
getItemAt	Retrieves a record from a RecordSet object
getItemID	Gets the unique ID corresponding to a record
getLength	Returns the total number of records in a RecordSet object
getNumberAvailable	Returns the number of records that have been downloaded from the server
isFullyPopulated	Determines whether a RecordSet object can be edited or manipulated
isLocal	Determines whether a RecordSet object is local or server-associated
removeAll	Removes all records from the RecordSet object
removeItemAt	Removes a specified record
replaceItemAt	Replaces the entire contents of a record
setDeliveryMode	Changes the delivery mode of a server-associated record set
setField	Replaces one field of a record with a new value
sort	Sorts all records by a specified compare function
sortItemsBy	Sorts all the records by a selected field

These functions are available for every RecordSet object returned by the `CF.query` function to the Flash MX client. You invoke these functions as follows:

```
objectName.functionName();
```

For example, in the result function that you create to handle record set data returned by the `CF.query` function, you can reference the database column names returned in the record set using the `getColumnNames` RecordSet function:

```
function selectData_Result ( result )
{
  //result holds the query data; employeesView is a Flash list box
  stringOutput.text = result.getColumnNames();
  _root.employeesView.setDataProvider(result);
}
```

Building a simple application

The following procedure describes how to build a simple server-side ActionScript application. The example application, a corporate personnel directory, uses the NetServices object to connect to the `personneldirectory` server-side ActionScript. The `personneldirectory` server-side ActionScript retrieves data from a ColdFusion MX data source and returns the results to the Flash application as a RecordSet object.

Note: The server-side ActionScript application that you create provides the back-end services in an application.

This example requires the following:

- A server-side ActionScript file named personneldirectory.asr that includes functions that interact with a ColdFusion MX data source.
- A client-side Flash MX movie in which the NetServices object is created.

To create the application:

1. Write server-side ActionScript that performs the database query and returns data to the client through the Flash Remoting service.
2. Create the Flash movie interface. See "Creating the Flash movie interface" on page 834.
3. Define a search function that sends user data to the Flash Remoting service. See "Submitting user data to the Flash Remoting service" on page 834.
4. Define a result function that captures the results returned from the Flash Remoting service. See "" on page 835.
5. Ensure that the Flash movie has established a connection to the Flash Remoting service. See "Checking for a Flash Remoting service connection" on page 836.

Writing the server-side ActionScript function

The example in this section creates a search function that performs a simple search operation against a ColdFusion MX data source. This function accepts two arguments, firstName and lastName, and returns any records found that match these arguments.

To create a server-side ActionScript function:

1. Create a server-side ActionScript file that contains the following code:

```
//search takes firstName lastName arguments
function search(firstName, lastName)
{
  searchdata = CF.query({datasource: "bigDSN",
    sql:"SELECT * from personnel WHERE fname = firstName AND lname =
  lastName"{);

  if (searchdata)
    return searchdata;
  else
    return null;
}
```

2. Save the file as personneldirectory.asr.

Creating the Flash movie interface

The Flash movie interface example in this section consists of one frame with a variety of text boxes and a submit button.

To create the Flash movie interface:

1. In the Flash MX authoring environment, create a new Flash source file, and save it as pDirectory.fla.
2. Create two input text boxes. Name one text box variable *lastName* and the other *firstName*.
3. Create a dynamic text box, and name its variable *status*.
4. Insert a list box component, and name it *dataView*.
5. Insert a push button component.
6. Save your work.

The following figure shows what the pDirectory Flash movie might look like:



Submitting user data to the Flash Remoting service

To send data to server-side ActionScript, you must create a function that passes the data from the Flash movie to server-side ActionScript. The `search` function, applied at the frame level, collects the user-entered data from the firstName and lastName text boxes and passes the data as function arguments to the directoryService object, which is created when the Flash movie connects to the Flash Remoting service. For more information, see "Checking for a Flash Remoting service connection" on page 836.

The following is a Flash MX ActionScript example:

```
#include "NetServices.as"
function search()
{
  // The search() method is defined in the server-side AS file
  directoryService.search(firstName.text, lastName.text);
  dataView.setDataProvider(null);
  status.text = "waiting...";
}
```

Reviewing the code

The following table describes the code and its function:

Code	Description
`directoryService.search (firstName.text, lastName.text);`	Passes the contents of the firstName and lastName text boxes to server-side ActionScript.
`dataView.setDataProvider (null);`	Clears the `dataView` list box component.
`status.text = "waiting...";`	Displays a message in the status text box while the record set is being retrieved from server-side ActionScript.

Capturing Flash Remoting service results

When you create a function that calls a server-side ActionScript function, you must also create a function to handle the data returned by server-side ActionScript. Define the function with the same name as the function making the initial call, but you append `_Result` to the name.

For example, if you create a function called `basicQuery` to return query data, you also need to define a results function to handle returned data; declare the results function as `basicQuery_Result`.

In the following example, the results function `search_Result` supplies the record set to the `dataView.setDataProvider` function:

```
function search_Result(resultset)
{
  dataView.setDataProvider(resultset);
  status.text = (0+resultset.getLength())+" names found.";
}
```

Reviewing the code

The following table describes the code and its function:

Code	Description
`function search_Result (resultset)`	The `_Result` suffix tells the Flash Remoting service to return the results of the search function to this function.
`dataView.setDataProvider (resultset);`	Assigns the results returned by the Flash Remoting service to the `dataView` list box.
`status.text = (0+resultset. getLength())+" names found.";`	Displays the number of records returned by the Flash Remoting service.

Checking for a Flash Remoting service connection

To ensure that the Flash movie is connected to the Flash Remoting service, you use an `if` statement; for example:

```
if (inited == null)
{
  inited = true;
  NetServices.setDefaultGatewayUrl("http://localhost:8500/flashservices/
    gateway");
  gateway_conn = NetServices.createGatewayConnection();
  directoryService = gateway_conn.getService(personneldirectory, this);
  status.text = "Type into the text boxes, then click 'Search'";
}
```

In this example, the `inited` variable is evaluated for a value. If `inited` is `null` (not connected), the movie connects to the Flash Remoting service using the NetServices object. For more information about connecting to the Flash Remoting service, see "Connecting to the Flash Remoting service" on page 826.

About the CF.http function

You use the `CF.http` ActionScript function to retrieve information from a remote HTTP server using HTTP `Get` and `Post` methods, as follows:

- Using the `Get` method, you send information to the remote server directly in the URL. This is common for a one-way transaction in which the `CF.http` function retrieves an object, such as the contents of a web page.
- The `Post` method can pass variables to a form or CGI program, and can also create HTTP cookies.

The most basic way to use the `CF.http` function is to use it with the `Get` method argument to retrieve a page from a specified URL. The `Get` method is the default for the `CF.http` function.

The following server-side example retrieves file content from the specified URL:

```
function basicGet(url)
{
  // Invoke with just the url argument. This is an HTTP GET.
  result = CF.http(url);
  return result.get("Filecontent");
}
```

The client-side example could look like the following:

```
#include "NetServices.as"
NetServices.setDefaultGatewayUrl("http://mycfserver:8500");
gatewayConnnection = NetServices.createGatewayConnection();
myHttp = gatewayConnnection.getService("httpFuncs", this);

// This is the server-side function invocation
url = "http://anyserver.com";
myHttp.basicGet(url);
```

```
// Create the results function
function basicGet_Result()
{
  url = "http://anyserver.com
  ssasFile.basicGet(url)
}
```

Using the CF.http function

The CF.http function returns an object that contains properties, also known as attributes. You reference these attributes to access the contents of the file returned, header information, HTTP status codes, and so on. The following table shows the available properties:

Property	Description
Text	A Boolean value indicating whether the specified URL location contains text data.
Charset	The charset used by the document specified in the URL. HTTP servers normally provide this information, or the charset is specified in the charset parameter of the Content-Type header field of the HTTP protocol. For example, the following HTTP header announces that the character encoding is EUC-JP: Content-Type: text/html; charset=EUC-JP
Header	Raw response header. For example, macromedia.com returns the following header : HTTP/1.1 200 OK Date: Mon, 04 Mar 2002 17:27:44 GMT Server: Apache/1.3.22 (Unix) mod_perl/1.26 Set-Cookie: MM_cookie=207.22.48.162.4731015262864476; path=/; expires=Wed, 03-Mar-04 17:27:44 GMT; domain=.macromedia.com Connection: close Content-Type: text/html
Filecontent	File contents, for text and MIME files.
Mimetype	MIME type. Examples of MIME types include text/html, image/png, image/gif, video/mpeg, text/css, and audio/basic.
responseHeader	Response header. If there is one instance of a header key, this value can be accessed as a simple type. If there is more than one instance, values are put in an array in the responseHeader structure.
Statuscode	HTTP error code and associated error string. Common HTTP status codes returned in the response header include the following: 400: Bad Request 401: Unauthorized 403: Forbidden 404: Not Found 405: Method Not Allowed

Referencing HTTP Post parameters in the CF.http function

To pass HTTP `Post` parameters in the `CF.http` function, you must construct an array of objects and assign this array to a variable named `params`. The following arguments can only be passed as an array of objects in the `params` argument of the `CF.http` function:

Parameter	Description
name	The variable name for data that is passed
type	Transaction type: • URL • FormField • Cookie • CGI • File
value	Value of URL, FormField, Cookie, File, or CGI variables that are passed

In the following example, the `CF.http` function passes HTTP `Post` parameters in an array of objects:

```
function postWithParamsAndUser()
{
  // Set up the array of Post parameters. These are just like cfhttpparam tags.
  params = new Array();
  params[1] = {name:"arg2", type:"URL", value:"value2"};

  url = "http://localhost:8500/";

  // Invoke with the method, url, params, username, and password
  result = CF.http("post", url, params, "karl", "salsa");
  return result.get("Filecontent");
}
```

Using the CF.http Post method

You use the `Post` method to send cookie, form field, CGI, URL, and file variables to a specified ColdFusion page or CGI program for processing. For POST operations, you must use the `params` argument for each variable that you post. The `Post` method passes data to a specified ColdFusion page or an executable that interprets the variables being sent, and returns data.

For example, when you build an HTML form using the `Post` method, you specify the name of the page to which form data is passed. You use the `Post` method in the `CF.http` function in a similar way. However, with the `CF.http` function, the page that receives the Post does not display anything. See the following example:

```
function postWithParams()
{
  // Set up the array of Post parameters. These are just like cfhttpparam tags.
  // This example passes formfield data to a specified URL.
  params = new Array();
  params[1] = {name:"Formfield1", type:"FormField", value:"George"};
  params[2] = [name:"Formfield2", type:"FormField", value:"Brown"};
```

```
  url = "http://localhost:8500/";

  // Invoke CF.http with the method, url, and params
  result = CF.http("post", url, params);
  return result.get("Filecontent");
}
```

Using the CF.http Get method

You use the `Get` method to retrieve files, including text and binary files, from a specified server. You reference properties of the object returned by the `CF.http` function to access things like file content, header information, MIME type, and so on.

The following example uses the `CF.http` function to show a common approach to retrieving data from the web:

```
// Returns content of URL defined in url variable
// This example uses positional argument style
function get()
{
  url = "http://www.macromedia.com/software/coldfusion/";

  //Invoke with just the url argument. Get is the default.
  result = CF.http(url);
  return result.get("Filecontent");
}
```

For more information about `CF.http` function properties, see `CF.http` in *CFML Reference*.

PART VI
Using Web Elements and External Objects

This part describes how you can use web elements such as XML, web services, Enterprise JavaBeans (EJBs), JSP pages, and Java servlets in ColdFusion applications. It also describes how to use external objects, including Java, Component Object Model (COM), and Common Object Request Broker Architecture (CORBA) objects in CFML applications.

The following chapters are included:

Chapter 35: Using XML and WDDX 843
Chapter 36: Using Web Services 885
Chapter 37: Integrating J2EE and Java Elements in CFML Applications 917
Chapter 38: Integrating COM and CORBA Objects in CFML Applications. 945

CHAPTER 35
Using XML and WDDX

This chapter describes how to use Macromedia ColdFusion MX to create, use, and manipulate XML documents. This chapter also presents Web Distributed Data Exchange (WDDX), an XML dialect for transmitting structured data, and describes how to use it to transfer data between applications and between CFML and JavaScript.

This chapter does not present XML concepts. Before you read this chapter you should become familiar with XML.

Contents

About XML and ColdFusion 843
The XML document object 844
ColdFusion XML tag and functions 850
Using an XML object 852
Creating and saving an XML document object 856
Modifying a ColdFusion XML object 858
Validating XML documents 868
Transforming documents with XSLT 869
Extracting data with XPath 869
Example: using XML in a ColdFusion application 870
Moving complex data across the web with WDDX 875
Using WDDX 879

About XML and ColdFusion

In the last few years, XML has rapidly become the universal language for representing documents and data on the web. These documents can extend beyond the traditional concept of a paper document or its equivalent. For example, XML is often used to represent database or directory information. XML is also commonly used to represent transaction information, such as product orders or receipts, and for information such as inventory records and employee data.

Because XML represents data in a tagged, textual format it is an excellent tool for representing information that must be shared between otherwise-independent applications such as order entry and inventory management. No application needs to know anything about the other. Each application only needs to be prepared to get data in a format that is structured according to the XML DTD or Schema. For example, in a distributed order processing application, the order placement component, order fulfilment component, inventory management component, and billing component can all share information with each other in XML format. They could use a common XML DTD, of different components could communicate with each other using different DTDs.

After an application parses the XML document, it can then manipulate the information in any way that is appropriate. For example, you can convert tabular XML data into a ColdFusion recordset, perform queries on the data and then export the data an XML document. For example, the code in "Example: using XML in a ColdFusion application" on page 870 takes a customer order in XML, converts the data to a recordset, and uses a query to determine the order cost. It then prepares a receipt as an XML document.

ColdFusion provides a comprehensive and easy-to-use set of tools for creating and using XML documents. ColdFusion lets you do the following with XML documents:

- Convert XML text into ColdFusion XML document objects.
- Create new ColdFusion XML document objects.
- Modify ColdFusion XML document objects.
- Validate XML against a DTD or Schema
- Transform XML using XSLT (Extensible Stylesheet Language Transformation).
- Extract data from XML documents using XPath expressions.
- Convert ColdFusion XML document objects to text and save them in files.

ColdFusion MX can also represent forms that you create using the `cfform` tag as XML. You can have ColdFusion generate the XML and process it using an XSLT skin to generate output for display, or ColdFusion MX can generate XML text and put it in a variable for further processing. For more information on XML Forms, see Chapter 30, "Creating Skinnable XML Forms," on page 709.

The XML document object

ColdFusion represents an XML document as an object, called an *XML document object*, that is much like a standard ColdFusion structure. In fact, most ColdFusion structure functions, such as `StructInsert`, work with XML document objects. For a full list of ColdFusion functions that work on XML document objects, see "Functions for XML object management" on page 858.

You can look at the overall structure of an XML document in two ways: a basic view and a DOM (Document Object Model)-based node view. The basic view presents all the information in the document, but does not separate the data into as fine-grained units as the node view. ColdFusion can access XML document contents using either view.

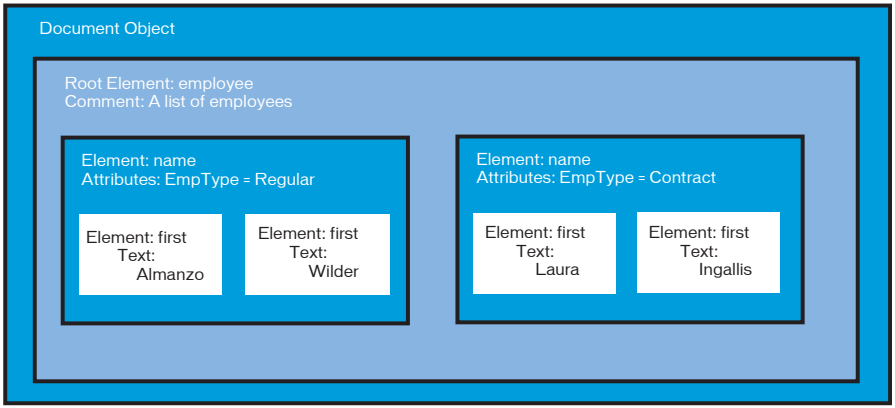
A simple XML document

The next sections describe the basic and node views of the following simple XML document. This document is used in many of the examples in this chapter.

```
<?xml version="1.0" encoding="UTF-8"?>
<employee>
<!-- A list of employees -->
  <name EmpType="Regular">
    <first>Almanzo</first>
    <last>Wilder</last>
  </name>
  <name EmpType="Contract">
    <first>Laura</first>
    <last>Ingalls</last>
  </name>
</employee>
```

Basic view

The basic view of an XML document object presents the object as a container that holds one root element structure. The root element can have any number of nested element structures. Each element structure represents an XML tag (start tag/end tag set) and all its contents; it can contain additional element structures. A basic view of the simple XML document looks like the following:



DOM node view

The DOM node view presents the XML document object using the same format as the document's XML *Document Object Model (DOM)*. In fact, an XML document object is a representation of a DOM object.

The DOM is a World Wide Web Consortium (W3C) recommendation (specification) for a platform- and language-neutral interface to dynamically access and update the content, structure, and style of documents. ColdFusion conforms to the DOM Level 2 Core specification, available at www.w3.org/TR/DOM-Level-2-Core.

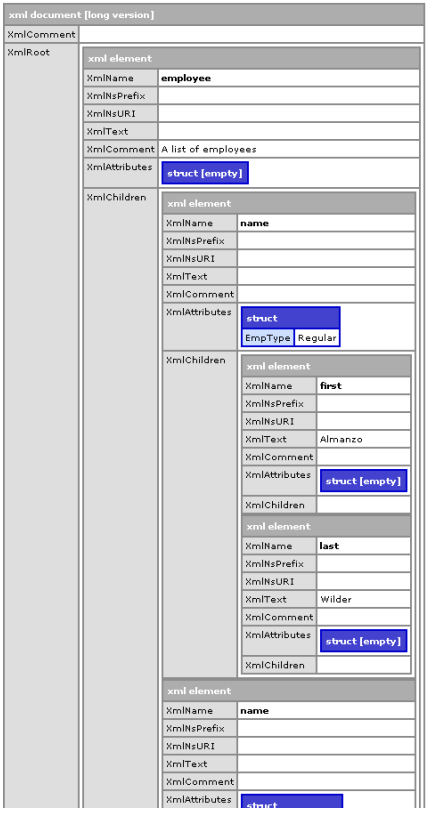
In the DOM node view, the document consists of a hierarchical tree of nodes. Each node has a DOM node type, a node name, and a node value. Node types include Element, Comment, Text, and so on. The DOM structures the document object and each of the elements it contains into multiple nodes of different types, providing a finer-grained view of the document structure than the basic view. For example, if an XML comment is in the middle of a block of text, the DOM node view represents its position in the text while the basic view does not.

ColdFusion also lets you use the DOM objects, methods, and properties defined in the W3C DOM Level 2 Core specification to manipulate the XML document object.

For more information on referencing DOM nodes, see "XML DOM node structure" on page 849. This document does not cover the node view and using DOM methods and properties in detail.

XML document structures

An XML document object is a structure that contains a set of nested XML element structures. The following figure shows a section of the `cfdump` tag output for the document object for the XML in "A simple XML document" on page 845. This figure shows the long version of the dump, which provides complete details about the document object. Initially, ColdFusion displays a short version, with basic information. Click the dump header to change between short, long, and collapsed versions of the dump.



The following code displays this output. It assumes that you save the code in a file under your web root, such as C:\Inetpub\wwwroot\testdocs\employeesimple.xml

```
<cffile action="read" file="C:\Inetpub\wwwroot\testdocs\employeesimple.xml"
  variable="xmldoc">
<cfset mydoc = XmlParse(xmldoc)>
<cfdump var="#mydoc#">
```

The document object structure

At the top level, the XML document object has the following three entries:

Entry name	Type	Description
XmlRoot	Element	The root element of the document.
XmlComment	String	A string made of the concatenation of all comments on the document, that is, comments in the document prologue and epilog. This string does not include comments inside document elements.
XmlDocType	XmlNode	The DocType attribute of the document. This entry only exists if the document specifies a DocType. This value is read-only; you cannot set it after the document object has been created This entry does not appear when `cfdump` displays an XML element structure.

The element structure

Each XML element has the following entries:

Entry name	Type	Description
XmlName	String	The name of the element; includes the namesapce prefix.
XmlNsPrefix	String	The prefix of the namespace.
XmlNsURI	String	The URI of the namespace.
XmlText or XmlCdata	String	A string made of the concatenation of all text and CData text in the element, but not inside any child elements. When you assign a value to the XmlCdata element, ColdFusion puts the text inside a CDATA information item. When you retrieve information from document object, these element names return identical values.
XmlComment	String	A string made of the concatenation of all comments inside the XML element, but not inside any child elements.
XmlAttributes	Structure	All of this element's attributes, as name-value pairs.
XmlChildren	Array	All this element's children elements.
XmlParent	XmlNode	The parent DOM node of this element. This entry does not appear when `cfdump` displays an XML element structure.
XmlNodes	Array	An array of all the XmlNode DOM nodes contained in this element. This entry does not appear when `cfdump` displays an XML element structure.

XML DOM node structure

The following table lists the contents of an XML DOM node structure:

Entry name	Type	Description
XmlName	String	The node name. For nodes such as Element or Attribute, the node name is the element or attribute name.
XmlType	String	The node XML DOM type, such as Element or Text.
XmlValue	String	The node value. This entry is used only for Attribute, CDATA, Comment, and Text type nodes.

Note: The `cfdump` tag does not display XmlNode structures. If you try to dump an XmlNode structure, the `cfdump` tag displays "Empty Structure."

The following table lists the contents of the XmlName and XmlValue fields for each node type that is valid in the XmlType entry. The node types correspond to the objects types in the XML DOM hierarchy.

Node type	XmlName	xmlValue
`CDATA`	`#cdata-section`	Content of the CDATA section
`COMMENT`	`#comment`	Content of the comment
`ELEMENT`	Tag name	Empty string
`ENTITYREF`	Name of entity referenced	Empty string
`PI (processing instruction)`	Target entire content excluding the target	Empty string
`TEXT`	`#text`	Content of the text node
`ENTITY`	Entity name	Empty string
`NOTATION`	Notation name	Empty string
`DOCUMENT`	`#document`	Empty string
`FRAGMENT`	`#document-fragment`	Empty string
`DOCTYPE`	Document type name	Empty string

Note: Although XML attributes are nodes on the DOM tree, ColdFusion does not expose them as XML DOM node data structures. To view an element's attributes, use the element structure's XMLAttributes structure.

The XML document object and all its elements are exposed as DOM node structures. For example, you can use the following variable names to reference nodes in the DOM tree that you created from the XML example in "A simple XML document" on page 845:

```
mydoc.XmlName
mydoc.XmlValue
mydoc.XmlRoot.XmlName
mydoc.employee.XmlType
mydoc.employee.XmlNodes[1].XmlType
```

ColdFusion XML tag and functions

The following table lists the ColdFusion tag (`cfxml`) and functions that create and manipulate XML documents:

Tag or function	Description
`<cfxml variable="objectName" [caseSensitive="Boolean"]>`	Creates a new ColdFusion XML document object consisting of the markup in the tag body. The tag can include XML and CFML tags. ColdFusion processes all CFML in the tag body before converting the resulting text to an XML document object. If you specify the `CaseSensitive="True"` attribute, the case of names of elements and attributes in the document is meaningful. The default value is `False`. For more information on using the `cfxml` tag, see "Creating a new XML document object using the cfxml tag" on page 856.
`XmlParse(XMLText [[, caseSensitive], validator])`	Converts an XML document in a file or a string variable into an XML document object, and optionally validates the document against a DTD or schema. If you specify the optional second argument as `True`, the case of names of elements and attributes in the document is meaningful. The default value is `False`. For more information on using the `XmlParse` function, see "Creating an XML document object from existing XML" on page 857.
`XmlNew([caseSensitive])`	Returns a new, empty XML document object. If you specify the optional argument as `True`, the case of names of elements and attributes in the document is meaningful. The default value is `False`. For more information on using the `XmlNew` function, see "Creating a new XML document object using the XmlNew function" on page 856.
`XmlElemNew(objectName(, namespaceURI], elementName)`	Returns a new XML document object element with the specified name, optionally belonging to the specified namespace. You can omit the *namespaceURI* parameter and use only a namespace prefix if the prefix is defined elsewhere in the object. For more information on using the `XmlElemNew` function, see "Adding an element" on page 863.
`XmlTransform(XMLVar, XSLTStringVar[, parameters])`	Applies an Extensible Stylesheet Language Transformation (XSLT) to an XML document. The document can be represented as a string variable or as an XML document object. The function returns the resulting XML document as a string. For more information on using the `XmlTransform` function, see "Transforming documents with XSLT" on page 869.
`XmlSearch(objectName, XPathExpression)`	Uses an XPath expression to search an XML document object and returns an array of XML elements that match the search criteria. For more information on using the `XmlSearch` function, see "Extracting data with XPath" on page 869.

Tag or function	Description
XmlValidate(*xmlDoc*[, *validator*])	Uses a Document Type Definition (DTD) or XML Schema to validate an XML text document (in a string or file) or an XML document object. The validator can be a DTD or Schema. If you omit the validator parameter, the document must specify a DTD or schema. For more information on using the XmlValidate function, see "Validating XML documents" on page 868
XmlChildPos(*element*, *elementName*, *position*)	Returns the position (index) in an XmlChildren array of the Nth child with the specified element name. For example, XmlChildPos(mydoc.employee, "name", 2) returns the position in mydoc.employee.XmlChildren of the mydoc.employee.name[2] element. This index can be used in the ArrayInsertAt and ArrayDeleteAt functions. For more information on using the XmlChildPos function, see "Determining the position of a child element with a common name" on page 862, "Adding an element" on page 863, and "Deleting elements" on page 865.
XmlGetNodeType(*xmlNode*)	Returns a string identifying the type of an XML document object node returned by the XmlSearch function or in an element's XmlNodes array.
IsWDDX(*String*)	Determines whether a string is a well-formed WDDX packet.
IsXML(*String*)	Determines whether a string is well-formed XML text.
IsXmlAttribute(*variable*)	Determines whether the function parameter is an XML Document Object Model (DOM) attribute node.
IsXmlDoc(*objectName*)	Returns True if the function argument is an XML document object.
IsXmlElem(*elementName*)	Returns True if the function argument is an XML document object element.
IsXmlNode(*variable*)	Determines whether the function parameter is an XML document object node.
IsXmlRoot(*elementName*)	Returns True if the function argument is the root element of an XML document object.
ToString(*objectName*)	Converts an XML document object to a string representation.
XmlFormat(*string*)	Escapes special XML characters in a string so that the string can be used as text in XML.

About case-sensitivity and XML document objects

The tags and functions that create XML document objects let you specify whether ColdFusion will treat the object in a case-sensitive manner. If you do not specify case-sensitivity, ColdFusion ignores the case of XML document object component identifiers, such as element and attribute names. If you do specify case-sensitivity, names with different cases refer to different components. For example, if you do not specify case-sensitivity, the names mydoc.employee.name[1] and mydoc.employee.NAME[1] always refer to the same element. If you specify case-sensitivity, these names refer to two separate elements. You cannot use dot notation references for element or attribute names in a case-sensitive XML document; for more information see "Referencing the contents of an XML object" on page 852.

Using an XML object

Because an XML document object is represented as a structure, you can access XML document contents using either, or a combination of both, of the following ways:

- Using the element names, such as mydoc.employee.name[1]
- Using the corresponding structure entry names (that is, `XmlChildren` array entries), such as mydoc.employee.XmlChildren[1]

Similarly, you can use either, or a combination of both, of the following notation methods:

- Structure (dot) notation, such as mydoc.employee
- Associative array (bracket) notation, such as mydoc["employee"]

Referencing the contents of an XML object

Use the following rules when you reference the contents of an XML document object *on the right side* of an assignment or as a function argument:

- By default, ColdFusion ignores element name case. As a result, it considers the element name MyElement and the element name myELement to be equivalent. To make element name matching case-sensitive, specify `CaseSensitive="True"` in the `cfxml` tag, or specify `True` as a second argument in the `XmlNew` or `XmlParse` function that creates the document object.
- If your XML object is case sensitive, do not use dot notation to reference an element or attribute name. Use the name in associative array (bracket) notation, or a reference that does not use the case-sensitive name. For example, do not use names such as the following:

  ```
  MyDoc.employee.name[1]

  MyDoc.employee.XmlAttributes.Version
  ```

  Instead, use names such as the following:

  ```
  MyDoc.xmlRoot.XmlChildren[1]
  MyDoc.xmlRoot["name"][1]
  MyDoc.["employee"]["name"][1]

  MyDoc.xmlRoot.XmlAttributes["Version"]
  MyDoc["employee"].XmlAttributes["Version"]
  ```

Caution: Because ColdFusion always treats variable names as case-insensitive, using dot notation for element and attribute names in a case sensitive XML document can generate unexpected results (such as all-uppercase variable names), exceptions, or both.

- If your XML object is case sensitive, you cannot use dot notation to reference an element or attribute name. Use the name in associative array (bracket) notation, or a reference that does not use the case-sensitive name (such as XmlChildren[1]) instead.
- Use an array index to specify one of multiple elements with the same name; for example, `#mydoc.employee.name[1] and #mydoc.employee.name[2]`.

  If you omit the array index on the last component of an element identifier, ColdFusion treats the reference as the array of all elements with the specified name. For example, `mydoc.employee.name` refers to an array of two name elements.
- Use an array index into the `XmlChildren` array to specify an element without using its name; for example, `mydoc.XmlRoot.XmlChildren[1]`.
- Use associative array (bracket) notation to specify an element name that contains a period or colon; for example, `myotherdoc.XmlRoot["Type1.Case1"]`.
- You can use DOM methods in place of structure entry names.

For example, the following variables all refer to the `XmlText` value "Almanzo" in the XML document created in "A simple XML document" on page 845:

```
mydoc.XmlRoot.XmlChildren[1].XmlChildren[1].XmlText
mydoc.employee.name[1].first.XmlText
mydoc.employee.name[1]["first"].XmlText
mydoc["employee"].name[1]["first"].XmlText
mydoc.XmlRoot.name[1].XmlChildren[1]["XmlText"]
```

The following variables all refer to the EmpType attribute of the first name element in the XML document created in "A simple XML document" on page 845:

```
mydoc.employee.name[1].XmlAttributes.EmpType
mydoc.employee.name[1].XmlAttributes["EmpType"]
mydoc.employee.XmlChildren[1].XmlAttributes.EmpType
mydoc.XmlRoot.name[1].XmlAttributes["EmpType"]
mydoc.XmlRoot.XmlChildren[1].XmlAttributes.EmpType
```

Neither of these lists contains a complete set of the possible combinations that can make up a reference to the value or attribute.

Assigning data to an XML object

When you use an XML object reference on the left side of an expression, most of the preceding rules apply to the reference *up to the last element* in the reference string.

For example, the rules in "Referencing the contents of an XML object" on page 852 apply to mydoc.employee.name[1].first in the following expression:

```
mydoc.employee.name[1].first.MyNewElement = XmlElemNew(mydoc, NewElement);
```

The rule for naming in case correct document objects, however, *applies to the full reference string*, as indicated by the following caution:

Caution: Because ColdFusion always treats variable names as case insensitive, using dot notation for element and attribute names in a case-sensitive XML document can generate unexpected results (such as all-uppercase variable names), exceptions, or both. In case-sensitive XML documents, use associative array notation or DOM notation names (such as XmlRoot or XmlChildren[2]).

Referencing the last element on the left side of an expression

The following rules apply to the meaning of the last component on the left side of an expression:

- The component name is an element structure key name (XML property name), such as `XmlComment`, ColdFusion sets the value of the specified element structure entry to the value of the right side of the expression. For example, the following line sets the XML comment in the mydoc.employee.name[1].first element to "This is a comment":

  ```
  mydoc.employee.name[1].first.XmlComment = "This is a comment";
  ```

- If the component name specifies an element name and does not end with a numeric index, for example `mydoc.employee.name`, ColdFusion assigns the value on the right of the expression to the first matching element.

  For example, if both `mydoc.employee.name[1]` and `mydoc.employee.name[2]` exist, the following expression replaces `mydoc.employee.name[1]` with a new element named address, not an element named name:

  ```
  mydoc.employee.name = XmlElemNew(mydoc, "address");
  ```

  After executing this line, if there had been both `mydoc.employee.name[1]` and `mydoc.employee.name[2]`, there is now only one `mydoc.employee.name` element with the contents of the original `mydoc.employee.name[2]`.

- If the component name does not match an existing element, the element names on the left and right sides of the expression must match. ColdFusion creates a new element with the name of the element on the left of the expression. If the element names do not match, it generates an error.

  For example if there is no `mydoc.employee.name.phoneNumber` element, the following expression creates a new `mydoc.employee.name.phoneNumber` element:

  ```
  mydoc.employee.name.phoneNumber = XmlElemNew(mydoc, "phoneNumber");
  ```

  The following expression causes an error:

  ```
  mydoc.employee.name.phoneNumber = XmlElemNew(mydoc, "address");
  ```

- If the component name does not match an existing element and the component's parent or parents also do not exist, ColdFusion creates any parent nodes as specified on the left side and use the previous rule for the last element. For example, if there is no `mydoc.employee.phoneNumber` element, the following expression creates a phoneNumber element containing an AreaCode element:

  ```
  mydoc.employee.name.phoneNumber.AreaCode = XmlElemNew(mydoc, "AreaCode");
  ```

Assigning and retrieving CDATA values

To identify that element text is CDATA by putting it inside CDATA start and end marker information items, assign the text to the XmlCdata element, not the XmlText element. You must do this because ColdFusion escapes the < and > symbols in the element text when you assign it to an XmlText entry. You can assign a value to an element's XmlText entry *or* its XmlCdata entry, but not to both, as each assignment overwrites the other.

When you retrieve data from the document object, references to XmlCdata and XmlText return the same string.

The following example shows how ColdFusion handles CDATA text:

```
<cfscript>
  myCDATA = "This is CDATA text";
  MyDoc = XmlNew();
  MyDoc.xmlRoot = XmlElemNew(MyDoc,"myRoot");
  MyDoc.myRoot.XmlChildren[1] = XmlElemNew(MyDoc,"myChildNodeCDATA");
  MyDoc.myRoot.XmlChildren[1].XmlCData = "#myCDATA#";
</cfscript>

<h3>Assigning a value to MyDoc.myRoot.XmlChildren[1].XmlCdata.</h3>
<cfoutput>
  The type of element MyDoc.myRoot.XmlChildren[1] is:
  #MyDoc.myRoot.XmlChildren[1].XmlType#<br>
  The value when output using XmlCdata is:
  #MyDoc.myRoot.XmlChildren[1].XmlCData#<br>
  The value when output using XmlText is:
  #MyDoc.myRoot.XmlChildren[1].XmlText#<br>
</cfoutput>
<br>
The XML text representation of Mydoc is:
<cfoutput><XMP>#tostring(MyDoc)#</XMP></cfoutput>

<h3>Assigning a value to MyDoc.myRoot.XmlChildren[1].XmlText.</h3>
<cfset MyDoc.myRoot.XmlChildren[1].XmlText = "This is XML plain text">
<cfoutput>
  The value when output using XmlCdata is:
  #MyDoc.myRoot.XmlChildren[1].XmlCData#<br>
  The value when output using XmlText is:
  #MyDoc.myRoot.XmlChildren[1].XmlText#<br>
</cfoutput>
<br>
The XML text representation of Mydoc is:
<cfoutput><XMP>#tostring(MyDoc)#</XMP></cfoutput>
```

Creating and saving an XML document object

The following sections show the ways you can create and save an XML document object. The specific technique that you use will depend on the application and your coding style.

Creating a new XML document object using the cfxml tag

The `cfxml` tag creates an XML document object that consists of the XML markup in the tag body. The tag body can include CFML code. ColdFusion processes the CFML code and includes the resulting output in the XML. The following example shows a simple `cfxml` tag:

```
<cfset testVar = True>
<cfxml variable="MyDoc">
  <MyDoc>
    <cfif testVar IS True>
      <cfoutput>The value of testVar is True.</cfoutput>
    <cfelse>
      <cfoutput>The value of testVar is False.</cfoutput>
    </cfif>
    <cfloop index = "LoopCount" from = "1" to = "4">
      <childNode>
        This is Child node <cfoutput>#LoopCount#.</cfoutput>
      </childNode>
    </cfloop>
  </MyDoc>
</cfxml>
<cfdump var=#MyDoc#>
```

This example creates a document object with a root element MyDoc, which includes text that displays the value of the ColdFusion variable testVar. MyDoc has four nested child elements, which are generated by an indexed `cfloop` tag. The `cfdump` tag displays the resulting XML document object.

Note: When you use the `cfxml` tag, do not include an <?xml ?> processing directive in the tag body. This directive is not required, and causes an error. To process XML text that includes the <?xml ?> directive, use the `XmlParse` function.

Creating a new XML document object using the XmlNew function

The `XmlNew` function creates a new XML document object, which you must then populate. For information on how to populate a new XML document, see "Adding, deleting, and modifying XML elements" on page 862.

Note: You cannot set the `XmlDocType` property for an XML document object that you create with the `XmlNew` function.

The following example creates and displays the same ColdFusion document object as in "Creating a new XML document object using the cfxml tag" on page 856.

```
<cfset testVar = True>
<cfscript>
  MyDoc = XmlNew();
  MyDoc.xmlRoot = XmlElemNew(MyDoc,"MyRoot");
  if (testVar IS TRUE)
    MyDoc.MyRoot.XmlText = "The value of testVar is True.";
```

```
  else
    MyDoc.MyRoot.XmlText = "The value of testVar is False.";
  for (i = 1; i LTE 4; i = i + 1)
    {
    MyDoc.MyRoot.XmlChildren[i] = XmlElemNew(MyDoc,"childNode");
    MyDoc.MyRoot.XmlChildren[i].XmlText = "This is Child node " & i &".";
    }
</cfscript>
<cfdump var=#MyDoc#>
```

Creating an XML document object from existing XML

The `XmlParse` function converts an XML document or document fragment represented as text into a ColdFusion document object. You can use a string variable containing the XML or the name or URL of a file that contains the text. For example, if your application uses `cfhttp action="get"` to get the XML document, use the following line to create the XML document object:

```
<cfset myXMLDocument = XmlParse(cfhttp.fileContent)>
```

The following example converts an XML text document is in a file to an XML document object:

```
C:\temp\myxmldoc.xml" variable="XMLFileText">
<cfset myXMLDocument=XmlParse("C:\temp\myxmldoc.xml" variable=")>
```

The `XmlParse` function takes a second, optional, attribute that specifies whether to maintain the case of the elements and attributes in the document object. The default is to have the document object be case-insensitive. For more information on case-sensitivity, see "Referencing the contents of an XML object" on page 852.

The `XmlParse` function also lets you specify a DTD or Schema to validate the XML text; if the XML is not valid, ColdFusion MX generates an error. You can specify the filename or URL of the validator, or the DTD or Schema can be in a CFML variable. You can also tell ColdFusion MX to use a DTD or Schema that is identified in the XML text. If you specify validation, you must also specify whether the document is be case-sensitive. The following example validates an XML document on file using a DTD that it specifies using a URL:

```
myDoc=XMLParse("C:\CFusionMX7\wwwroot\examples\custorder.xml", false,
     "http://localhost:8500/examples/custorder.dtd")>
```

Saving and exporting an XML document object

The `ToString` function converts an XML document object to a text string. You can then use the string variable in any ColdFusion tag or function.

To save the XML document in a file, use the `ToString` function to convert the document object to a string variable, then use the `cffile` tag to save the string as a file. For example, use the following code to save the XML document myXMLDocument in the file C:\temp\myxmldoc.xml:

```
<cfset XMLText=ToString(myXMLDocument)>
<cffile action="write" file="C:\temp\myxmldoc.xml" output="#XMLText#">
```

Modifying a ColdFusion XML object

As with all ColdFusion structured objects, you can often use a number of methods to change the contents of an XML document object. For example, you often have the choice of using an assignment statement or a function to update the contents of a structure or an array. The following section describes the array and structure functions that you can use to modify an XML document object. The section "XML document object management reference" on page 860 provides a quick reference to modifying XML document object contents. Later sections describe these methods for changing document content in detail.

Functions for XML object management

The following table lists the ColdFusion array and structure functions that you can use to manage XML document objects and their functions, and describes their common uses. In several cases you can use either an array function or a structure function for a purpose, such as for deleting all of an element's attributes or children.

Function	Use
`ArrayLen`	Determines the number of child elements in an element, that is, the number of elements in an element's `XmlChildren` array.
`ArrayIsEmpty`	Determines whether an element has any elements in its `XmlChildren` array.
`StructCount`	Determines the number of attributes in an element's `XmlAttributes` structure.
`StructIsEmpty`	Determines whether an element has any attributes in its `XmlAttributes` structure. Returns `True` if the specified structure, including the XML document object or an element, exists and is empty.
`StructKeyArray` `StructKeyList`	Gets an array or list with the names of all of the attributes in an element's `XmlAttributes` structure. Returns the names of the children of an XML element.
`ArrayInsertAt`	Adds a new element at a specific location in an element's `XmlChildren` array.
`ArrayAppend` `ArrayPrepend`	Adds a new element at the end or beginning of an element's `XmlChildren` array.
`ArraySwap`	Swaps the children in the `XmlChildren` array at the specified position.
`ArraySet`	Sets a range of entries in an `XmlChildren` array to equal the contents of a specified element structure. Each entry in the array range will be a copy of the structure. Can be used to set a single element by specifying the same index as the beginning and end of the range.
`ArrayDeleteAt`	Deletes a specific element from an element's `XmlChildren` array.
`ArrayClear`	Deletes all child elements from an element's `XmlChildren` array.

Function	Use
StructDelete	Deletes a selected attribute from an element's XMLAttributes structure. Deletes all children with a specific element name from an element's XmlChildren array. Deletes all attributes of an element. Deletes all children of an element. Deletes a selected property value.
StructClear	Deletes all attributes from an element's XMLAttributes structure.
Duplicate	Copies an XML document object, element, or node structure.
IsArray	Returns True for the XmlChildren array. Returns false if you specify an element name, such as mydoc.XmlRoot.name, even if there are multiple name elements in XmlRoot.
IsStruct	Returns False for XML document objects, elements, and nodes. Returns True for XmlAttributes structures.
StructGet	Returns the specified structure, including XML document objects, elements, nodes, and XmlAttributes structures.
StructAppend	Appends a document fragment XML document object to another XML document object.
StructInsert	Adds a new entry to an XmlAttributes structure.
StructUpdate	Sets or replaces the value of a document object property such as XmlName, or of a specified attribute in an XmlAttributes structure.

Note: Array and structure functions not in the preceding or table or the table in the next section, **do not** work with XML document objects, XML elements, or XML node structures.

Treating elements with the same name as an array

In many cases an XML element has multiple children with the same name. For example, the example document used in this chapter has multiple name elements in the employee elements. In many cases, you can treat the child elements with identical names as an array. For example, to reference the second name element in mydoc.employee, you can specify mydoc.employee.name[2]. However, you can only use a limited set of Array functions when you use this notation. The following table lists the array functions that are valid for such references:

Array function	Result
IsArray(*elemPath.elemName*)	Always returns False.
ArrayClear(*elemPath.elemName*)	Removes all the elements with name *elemName* from the *elemPath* element.
ArrayLen(*elemPath.elemName*)	Returns the number of elements named *elemName* in the *elemPath* element.
ArrayDeleteAt(*elemPath.elemName, n*)	Deletes the *n*th child named *elemlName* from the *elemPath* element.

Array function	Result
`ArrayIsEmpty(elemPath.elemName)`	Always returns `False`.
`ArrayToList(elemPath.elemName, n)`	Returns a comma separated list of all the `XmlText` properties of all the children of *elemPath* named *elemName*.

XML document object management reference

The following tables provide a quick reference to the ways you can modify the contents of an XML document object. The sections that follow describe in detail how to modify XML contents.

Note: If your XML object is case sensitive, you cannot use dot notation to reference an element or attribute name. Use the name in associative array (bracket) notation, or a reference that does not use the case-sensitive name (such as xmlChildren[1]) instead.

Adding information to an element

Use the following techniques to add new information to an element:

Type	Using a function	Using an assignment statement

Attribute	`StructInsert(xmlElemPath.XmlAttributes, "key", "value")`	`xmlElemPath.XmlAttributes.key = "value"`
		
		`xmlElemPath.XmlAttributes["key"] = "value"`

Child element	To append:
	`ArrayAppend(xmlElempath.XmlChildren, newElem)`
	
	To insert:
	`ArrayInsertAt(xmlElempath.XmlChildren, position, newElem)`
		To append:
		`xmlElemPath.XmlChildren[i] = newElem`
		
		`xmlElemPath.newChildName = newElem`
		(where *newChildName* must be the same as *newElem*.XmlName and cannot be an indexed name such as name[3])

Deleting information from an element

Use the following techniques to delete information from an element:

Type	Using a function	Using an assignment statement
Property	`StructDelete(xmlElemPath, propertyName)`	`xmlElemPath.propertyName=""`

Type	Using a function	Using an assignment statement

Attribute	All attributes:
	`StructDelete(xmlElemPath, XmlAttributes)`
	
	A specific attribute:
	`StructDelete(xmlElemPath.XmlAttributes, "attributeName")`
		Not available

Child element	All children of an element:
	`StructDelete(xmlElemPath, "XmlChildren")`
	or
	`ArrayClear(xmlElemPath.XmlChildren)`
	
	All children with a specific name:
	`StructDelete(xmlElementpath, "elemName")`
	`ArrayClear(xmlElemPath.elemName)`
	
	A specific child:
	`ArrayDeleteAt(xmlElemPath.XmlChildren, position)`
	`ArrayDeleteAt(xmlElemPath.elemName, position)`
		Not available

Changing contents of an element

Use the following techniques to change the contents of an element:

Type	Using a function	Using an assignment statement

Property	`StructUpdate(xmlElemPath, "propertyName", "value")`	`xmlElemPath.propertyName = "value"`
		
		`xmlElemPath["propertyName"] = "value"`

Attribute	`StructUpdate(xmlElemPath.XmlAttributes, "attributeName", "value")`	`xmlElemPath.XmlAttributes.attributeName="value"`
		
		`xmlElemPath.XmlAttributes["attributeName"] = "value"`

Child element (replace)	`ArraySet(xmlElemPath.XmlChildren, index, index, newElement)`
	
	(use the same value for both index entries to change one element)
		Replace first or only child named *elementName*:
		`parentElemPath.elementName = newElement`
		
		`parentElemPath["elementName"] = newElement`
		
		Replace a specific child named *elementName*:
		`parentElemPath.elementName[index] = newElement`
		or
		`parentElemPath["elementName"][index] = newElement`

Adding, deleting, and modifying XML elements

The following sections describe the basic techniques for adding, deleting, and modifying XML elements. The example code uses the XML document described in "A simple XML document" on page 845.

Counting and finding the position of child elements

Often, an XML element has several children with the same name. For example, in the XML document defined in the simple XML document, the employee root element has multiple name elements.

To manipulate such an object, you often need to know the number of children of the same name, and you might need to know the position in the `XmlChildren` array of a specific child name that is used for multiple children. The following sections describe how to get this information.

Counting child elements

The following user-defined function determines the number of child elements with a specific name in an element:

```
<cfscript>
function NodeCount (xmlElement, nodeName)
{
  nodesFound = 0;
  for (i = 1; i LTE ArrayLen(xmlElement.XmlChildren); i = i+1)
  {
    if (xmlElement.XmlChildren[i].XmlName IS nodeName)
      nodesFound = nodesFound + 1;
  }
  return nodesFound;
}
</cfscript>
```

The following lines use this function to display the number of nodes named "name" in the mydoc.employee element:

```
<cfoutput>
Nodes Found: #NodeCount(mydoc.employee, "name")#
</cfoutput>
```

Determining the position of a child element with a common name

The `XmlChildPos` function determines the location in the `XmlChildren` array of a specific element with a common name. You use this index when you need to tell ColdFusion where to insert or delete child elements. For example, if there are several name elements in mydoc.employee, use the following code to locate name[2] in the `XmlChildren` array:

```
<cfset nameIndex = XmlChildPos(mydoc.employee, "name", 2)>
```

Adding an element

You can add an element by creating a new element or by using an existing element.

Use the `XmlElemNew` function to create a new, empty element. This function has the following form:

```
XmlElemNew(docObject, elementName)
```

where *docObject* is the name of the XML document object in which you are creating the element, and *elementName* is the name you are giving the new element.

Use an assignment statement with an existing element on the right side to create a new element using an existing element. See "Copying an existing element" on page 864 for more information on adding elements using existing elements.

Adding an element using a function

You can use the `ArrayInsertAt` or `ArrayAppend` functions to add an element to an XML document object. For example, the following line adds a phoneNumber element after the last element for employee.name[2]:

```
<cfset ArrayAppend(mydoc.employee.name[2].XmlChildren, XmlElemNew(mydoc,
       "phoneNumber"))>
```

The following line adds a new department element as the first element in employee. The name elements become the second and third elements.

```
<cfset ArrayInsertAt(mydoc.employee.XmlChildren, 1, XmlElemNew(mydoc,
       "department"))>
```

You must use the format *parentElement*.`XmlChildren` to specify the array of elements to which you are adding the new element. For example, the following line causes an error:

```
<cfset ArrayInsertAt(mydoc.employee.name, 2, XmlElemNew(mydoc, "PhoneNumber")>
```

If you have multiple child elements with the same name, and you want to insert a new element in a specific position, use the `XmlChildPos` function to determine the location in the `XmlChildren` array where you want to insert the new element. For example, the following code determines the location of mydoc.employee.name[1] and inserts a new name element as the second name element:

```
<cfscript>
nameIndex = XmlChildPos(mydoc.employee, "name", 1);
ArrayInsertAt(mydoc.employee.XmlChildren, nameIndex + 1, XmlElemNew(mydoc,
           "name"));
</cfscript>
```

Using a namespace When you use a function to add an element, you can assign the element to a namespace by including the namespace prefix in the element name. If you have not yet associated the prefix with a namespace URI, you must also include a parameter with the namespace URI in the `XmlElemNew` function. This parameter must be the *second* parameter in the method, and the element name must be the third parameter. ColdFusion MX then associates the namespace prefix with the URI, and you can omit the URI parameter in further `xmlElemNew` functions.

The following example adds two to the supplies document root two elements in the Prod namespace. The first `XmlElemNew` function use sets the association between the Prod namespace prefix and the URI; the second use only needs to specify the prefix on the element name.

```
<cfscript>
  mydoc.supplies.XmlChildren[1] = XmlElemNew(mydoc,
  "http://www.foo.com/Products", "Prod:soap");
  mydoc.supplies.XmlChildren[2] = XmlElemNew(mydoc, “Prod:shampoo");
</cfscript>
```

Adding an element using direct assignment

You can use direct assignment to append a new element to an array of elements. You cannot use direct assignment to insert an element into an array of elements.

When you use direct assignment, you can specify on the left side an index into the `XmlChildren` array greater than the last child in the array. For example, if there are two elements in mydoc.employee, you can specify any number greater than two, such as mydoc.employee.XmlChildren[6]. The element is always added as the last (in this case, third) child.

For example, the following line appends a name element to the end of the child elements of mydoc.employee:

```
<cfset mydoc.employee.XmlChildren[9] = XmlElemNew(mydoc, "name")>
```

If the parent element does not have any children with the same name as the new child, you can specify the name of the new node or the left side of the assignment. For example, the following line appends a phoneNumber element to the children of the first name element in mydoc.employee:

```
<cfset mydoc.employee.name[1].phoneNumber = XmlElemNew(mydoc, "phoneNumber")>
```

You cannot use the node name on the left to add an element with the same name as an existing element in the parent. For example, if mydoc.employee has two name nodes, the following line causes an error:

```
<cfset mydoc.employee.name[3] = XmlElemNew(mydoc, "name")>
```

However, the following line does work:

```
<cfset mydoc.employee.XmlChilren[3] = XmlElemNew(mydoc, "name")>
```

Copying an existing element

You can add a copy of an existing element elsewhere in the document. For example, if there is a mydoc.employee.name[1].phoneNumber element, but no mydoc.employee. name[2].phoneNumber, the following line creates a new mydoc.employee. name[2]. phoneNumber element with the same value as the original element. This assignment copies the original element. Unlike with standard ColdFusion structures, you get a true copy, not a reference to the original structure. You can change the copy without changing the original.

```
<cfset mydoc.employee.name[2].phoneNumber =
  mydoc.employee.name[1].phoneNumber>
```

When you copy an element, the new element must have the same name as the existing element. If you specify the new element by name on the left side of an assignment, the element name must be the same as the name on the right side. For example, the following expression causes an error:

```
<cfset mydoc.employee.name[2].telephne = mydoc.employee.name[1].phoneNumber>
```

Deleting elements

There are many ways to delete individual or multiple elements.

Deleting individual elements

Use the `ArrayDeleteAt` function to delete a specific element from an XML document object. For example, the following line deletes the second child element in the mydoc.employee element:

```
<cfset ArrayDeleteAt(mydoc.employee.XmlChildren, 2)>
```

If an element has only one child element with a specific name, you can also use the `StructDelete` function to delete the child element. For example, the following line deletes the phoneNumber element named in the second employee.name element:

```
<cfset StructDelete(mydoc.employee.name[2], "phoneNumber")>
```

When there are multiple child elements of the same name, you must specify the element position, either among the elements of the same name, or among all child elements. Fore example, you can use the following line to delete the second name element in mydoc.employee:

```
<cfset ArrayDeleteAt(mydoc.employee.name, 2)>
```

You can also determine the position in the `XmlChildren` array of the element you want to delete and use that position. To do so, use the `XmlChildPos` function. For example, the following lines determine the location of mydoc.employee.name[2] and delete the element:

```
<cfset idx = XmlChildPos(mydoc.employee, "name", 2)>
<cfset ArrayDeleteAt(mydoc.employee.XmlChildren, idx)>
```

Deleting multiple elements

If an element has multiple children with the same name, use the `StructDelete` function or `ArrayClear` function with an element name to delete all of an element's child elements with that name. For example, both of the following lines delete all name elements from the employee structure:

```
<cfset StructDelete(mydoc.employee, "name")>
<cfset ArrayClear(mydoc.employee.name)>
```

Use the `StructDelete` or `ArrayClear` function with `XmlChildren` to delete all of an element's child elements. For example, each of the following lines deletes all child elements of the mydoc.employee.name[2] element:

```
<cfset StructDelete(mydoc.employee.name[2], "XmlChildren")>
<cfset ArrayClear(mydoc.employee.name[2].XmlChildren)>
```

Adding, changing, and deleting element attributes

You modify an element's attributes the same way you change the contents of any structure. For example, each of the following lines adds a Status attribute the second mydoc.employee.name element:

```
<cfset mydoc.employee.name[2].XmlAttributes.Status="Inactive">
<cfset StructInsert(mydoc.employee.name[2].XmlAttributes, "Status",
  "Inactive")>
```

To change an attribute, use a standard assignment statement; for example:

```
<cfset mydoc.employee.name[2].XmlAttributs.Status="Active">
```

To delete an attribute, use StructDelete; for example:

```
<cfset StructDelete(mydoc.employee.name[1].XmlAttributes, "Status")>
```

Changing element properties

To change an element's properties, including its text and comment, use a standard assignment expression. For example, use the following line to add "in the MyCompany Documentation Department" to the mydoc.employee XML comment:

```
<cfset mydoc.employee.XmlComment = mydoc.employee.XmlComment & "in the
MyCompany Documentation Department">
```

Changing an element name

The XML DOM does not support changing an element name directly. To change the name of an element, you must create a new element with the new name, insert it into the XML document object before or after the original element, copy all the original element's contents to the new element, and then delete the original element.

Clearing an element property value

To clear an element property value, either assign the empty string to the property or use the `StructDelete` function. For example, each of the following lines clears the comment string from mydoc.employee:

```
<cfset mydoc.employee.XmlComment = "">
<cfset StructDelete(mydoc.employee, "XmlComment")>
```

Replacing or moving an element

To replace an element with a new element, use a standard replacement expression. For example, to replace the mydoc.employee.department element with a new element named organization, use either of the following lines:

```
<cfset mydoc.employee.department = XmlElemNew(mydoc, "Organization")>
<cfset mydoc.employee.XmlChildren[1] = XmlElemNew(mydoc, "Organization")>
```

To replace an element with a copy of an existing element, use the existing element on the right side of an expression. For example, the following line replaces the phoneNumber element for mydoc.employee.name[2] with the phoneNumber element from mydoc.employee.name[1]:

```
<cfset mydoc.employee.name[2].phoneNumber=mydoc.employee.name[1].phoneNumber>
```

This creates a true copy of the name[1].phoneNumber element as name[2].phoneNumber.

To move an element, you must assign it to its new location, then delete it from its old location. For example, the following lines move the phoneNumber element from mydoc.employee.name[1] to mydoc.employee.name[2]:

```
<cfset mydoc.employee.name[2].phoneNumber=mydoc.employee.name[1].phoneNumber>
<cfset StructDelete(mydoc.employee.name[1], "phoneNumber")>
```

Note: You cannot copy or move an element from one document object to another document object.

Using XML and ColdFusion queries

You can convert XML documents into ColdFusion query objects and manipulate them using queries of queries. This technique does not require the use of XPath and provides a method of searching XML documents and extracting data that is natural to ColdFusion programmers.

Converting XML to a ColdFusion query

The following example reads an XML document, converts it to a query object, and then performs a query of queries on the object to extract selected data:

```
<!--- Read the file and convert it to an XML document object --->
<cffile action="read" file="C:\CFusionMX7\wwwroot\myexamples\employees.xml"
  variable="myxml">
<cfset mydoc = XmlParse(myxml)>

<!--- get an array of employees --->
<cfset emp = mydoc.employee.XmlChildren>
<cfset size = ArrayLen(emp)>

<cfoutput>
Number of employees = #size#
<br>
</cfoutput>
<br>
<!--- create a query object with the employee data --->
<cfset myquery = QueryNew("fname, lname") >
<cfset temp = QueryAddRow(myquery, #size#)>
<cfloop index="i" from = "1" to = #size#>
  <cfset temp = QuerySetCell(myquery, "fname",
    #mydoc.employee.name[i].first.XmlText#, #i#)>
  <cfset temp = QuerySetCell(myquery, "lname",
    #mydoc.employee.name[i].last.XmlText#, #i#)>
</cfloop>

<!--- Dump the query object --->
Contents of the myquery Query object: <br>
<cfdump var=#myquery#>
<br><br>

<!--- Select entries with the last name starting with A and dump the result --
  ->
<cfquery name="ImqTest" dbType="query">
  SELECT lname, fname
```

```
  FROM myquery
  WHERE lname LIKE 'A%'
</cfquery>
<cfdump var=#imqtest#>
```

Converting a query object to XML

The following example shows how to convert a query object to XML. It uses `cfquery` to get a list of employees from the cfdocexamples database and saves the information as an XML document.

```
<!--- Query the database and get the names in the employee table --->
<cfquery name="myQuery" datasource="cfdocexamples">
  SELECT FirstName, LastName
  FROM employee
</cfquery>

<!--- Create an XML document object containing the data --->
<cfxml variable="mydoc">
  <employee>
    <cfoutput query="myQuery">
      <name>
        <first>#FirstName#</first>
        <last>#LastName#</last>
      </name>
    </cfoutput>
  </employee>
</cfxml>

<!--- dump the resulting XML document object --->
<cfdump var=#mydoc#>
<!--- Write the XML to a file --->
<cffile action="write" file="C:\inetpub\wwwroot\xml\employee.xml"
    output=#toString(mydoc)#>
```

Validating XML documents

ColdFusion MX provides the following methods for validating a document against a DTD or an XML Schema:

- The `XmlParse` function can validate XML text that it is parsing against a DTD or Schema. It the function encounters a validation error, ColdFusion generates an error and stops parsing the text. If the validator generates warnings, but no errors, ColdFusion parses the document and returns the result.
- The `XmlValidate` function can validate an XML text document or XML document object. against a DTD or Schema. The function returns a data structure with detailed information from the validator, including arrays of warning, error, and fatal error messages, and a Boolean status variable indicating whether the document is valid. Your application can examine the status information and determine how to handle it further.

For examples of XML validation, see `XmlParse` and `XmlValidate` in *CFML Reference*. The `XmlParse` example validates a document using a DTD. The `XmlValidate` example validates the document using an XML Schema that represents the same document structure as the DTD.

Transforming documents with XSLT

The Extensible Stylesheet Language Transformation (XSLT) technology transforms an XML document into another format or representation. For example, one common use of XSLT is to convert XML documents into HTML for display in a browser. XSLT has many other uses, including converting XML data to another format, such as converting XML in a vocabulary used by an order entry application into a vocabulary used by an order fulfillment application.

XSLT transforms an XML document by applying an Extensible Stylesheet Language (XSL) stylesheet. (When stored in a file, XSL stylesheets typically have the suffix xsl.) ColdFusion provides the `XmlTransform` function to apply an XSL transformation to an XML document. The function takes an XML document in string format or as an XML document object, and an XSL stylesheet in string format, and returns the transformed document as a string.

The following code:

1. Reads the simpletransform.xsl stylesheet file into a string variable.
2. Uses the stylesheet to transform the mydoc XML document object.
3. Saves the resulting transformed document in a second file.

```
<cffile action="read"
  file="C:\CFusionMX7\wwwroot\testdocs\simpletransform.xsl"
  variable="xslDoc">
<cfset transformedXML = XmlTransform(mydoc, xslDoc)>
<cffile action="write"
  file="C:\CFusionMX7\wwwroot\testdocs\transformeddoc.xml"
  output=transformedXML>
```

XSL and XSLT are specified by the World Wide Web Consortium (W3C). For detailed information on XSL, XSLT, and XSL stylesheets, see the W3C website at www.w3.org/Style/XSL/. There are also several books available on using XSL and XSLT.

Extracting data with XPath

XPath is a language for addressing parts of an XML document. Like XSL, XPath is a W3C specification. One of the major uses of XPath is in XSL transformations. However, XPath has more general uses. In particular, it can extract data from XML documents, such as complex data set representations. Thus, XPath is another data querying tool.

XPath uses a pattern called an XPath expression to specify the information to extract from an XML document. For example, the simple XPath expression /employee/name selects the name elements in the employee root element.

The `XmlSearch` function uses XPath expressions to extract data from XML document objects. The function takes an XML document object and an XPath expression in string format, and returns an array of XML document objects containing the elements that meet the expression criteria.

The following example extracts all the elements named last, which contain the employee's last names, from the employeesimple.xml file, and displays the names:

```
<cffile action="read"
  file="C:\inetpub\wwwroot\examples\employeesimple.xml"
  variable="myxml">
<cfscript>
  myxmldoc = XmlParse(myxml);
  selectedElements = XmlSearch(myxmldoc, "/employee/name/last");
  for (i = 1; i LTE ArrayLen(selectedElements); i = i + 1)
    writeoutput(selectedElements[i].XmlText & "<br>");
</cfscript>
```

XPath is specified by the World-Wide Web Consortium. For detailed information on XPath, see the W3C website at www.w3.org/TR/xpath. Most books that cover XSLT also discuss XPath.

Example: using XML in a ColdFusion application

The example in this section shows how you can use XML to represent data, and how ColdFusion can use XML data in an application. Although the example is too simple to be used in an application without substantial changes, it presents some of the common uses of XML with ColdFusion.

The example receives an order in the form of an XML document, processes it, and generates an XML receipt document. In this case, the order document is in a file, but it could be received as the result of an HTTP request, or retrieved using `cfpop`, `cfftp`, or other methods. The ColdFusion page does the following with the order:

1. Generates a query object from an XML document.
2. Queries a database table to determine the order discount percentage to use.
3. Uses a query of queries to calculate the total price, then calculates the discounted price.
4. Generates the receipt as an XML document.

This example displays the results of the processing steps to show you what has been done.

The XML document

The order.xml document has the following structure:

- The root element is named order and has one attribute, id.
- There is one customer element with firstname, lastname, and accountnum attributes. The customer element does not have a body
- There is one items element that contains multiple item elements
- Each item element has an id attribute and contains a name, quantity, and unitprice element. The name, quantity, and unitprice elements contain their value as body text.

The following order.xml document works correctly with the information in the cfdocexamples database:

```
<order id="4323251">
  <customer firstname="Philip" lastname="Cramer" accountNum="21"/>
  <items>
    <item id="43">
      <name>
        Large Hammer
      </name>
      <quantity>
        1
      </quantity>
      <unitprice>
        15.95
      </unitprice>
    </item>
    <item id="54">
      <name>
        Ladder
      </name>
      <quantity>
        2
      </quantity>
      <unitprice>
        40.95
      </unitprice>
    </item>
    <item id="68">
      <name>
        Paint
      </name>
      <quantity>
        10
      </quantity>
      <unitprice>
        18.95
      </unitprice>
    </item>
  </items>
</order>
```

The ColdFusion page

The ColdFusion page looks like the following:

```
<!--- Convert file to XML document object --->
<cffile action="read" file="C:\CFusionMX7\wwwroot\examples\order.xml"
  variable="myxml">
<cfset mydoc = XmlParse(myxml)>

<!--- Extract account number --->
 <cfset accountNum=#mydoc.order.customer.XmlAttributes.accountNum#>
<!--- Display Order Information --->
<cfoutput>
```

```
  <b>Name=</b>#mydoc.order.customer.XmlAttributes.firstname#
       #mydoc.order.customer.XmlAttributes.lastname#
  <br>
  <b>Account=</b>#accountNum#
  <br>
  <cfset numItems = ArrayLen(mydoc.order.items.XmlChildren)>
  <b>Number of items ordered=</b> #numItems#
</cfoutput>
<br><br>

<!--- Process the order into a query object --->
<cfset orderquery = QueryNew("item_Id, name, qty, unitPrice") >
<cfset temp = QueryAddRow(orderquery, #numItems#)>
<cfloop index="i" from = "1" to = #numItems#>
  <cfset temp = QuerySetCell(orderquery, "item_Id",
    #mydoc.order.items.item[i].XmlAttributes.id#,#i#)>
  <cfset temp = QuerySetCell(orderquery, "name",
    #mydoc.order.items.item[i].name.XmlText#, #i#)>
  <cfset temp = QuerySetCell(orderquery, "qty",
    #mydoc.order.items.item[i].quantity.XmlText#, #i#)>
  <cfset temp = QuerySetCell(orderquery, "unitPrice",
    #mydoc.order.items.item[i].unitprice.XmlText#, #i#)>
</cfloop>

<!--- Display the order query --->
<cfdump var=#orderquery#>
<br><br>

<!--- Determine the discount --->
<cfquery name="discountQuery" datasource="cfdocexamples">
  SELECT *
  FROM employee
  WHERE Emp_Id = #accountNum#
</cfquery>
<cfset drate = 0>
<cfif #discountQuery.RecordCount# is 1>
  <cfset drate = 10>
</cfif>

<!--- Display the discount rate --->
<cfoutput>
  <b>Discount Rate =</b> #drate#%
</cfoutput>
<br><br>

<!--- Compute the total cost and discount price--->
<cfquery name="priceQuery" dbType="query">
  SELECT SUM(qty*unitPrice)
  AS totalPrice
  FROM orderquery
</cfquery>
<cfset discountPrice = priceQuery.totalPrice * (1 - drate/100)>

<!--- Display the full price and discounted price --->
<cfoutput>
```

```
  <b>Full Price=</b> #priceQuery.totalPrice#<br>
  <b>Discount Price=</b> #discountPrice#
</cfoutput>
<br><br>

<!---Generate an XML Receipt --->
<cfxml variable="receiptxml">
<receipt num = "34">
  <cfoutput>
    <price>#discountPrice#</price>
    <cfif drate GT 0 >
      <discountRate>#drate#</discountRate>
    </cfif>
  </cfoutput>
  <itemsFilled>
    <cfoutput query="orderQuery">
        <name>#name# </name>
        <qty>   #qty# </qty>
        <price> #qty*unitPrice# </price>
    </cfoutput>
  </itemsFilled>
</receipt>
</cfxml>

<!--- Display the resulting receipt --->
<cfdump var=#receiptxml#>
```

Reviewing the code

The following table describes the CFML code and its function. For the sake of brevity, it does not include code that displays the processing results.

Code	Description
`<cffile action="read" file="C:\CFusionMX7\wwwroot\examples\order.xml" variable="myxml">`	Reads the XML from a file and convert it to an XML document object.
`<cfset mydoc = XmlParse(myxml)>` `<cfset accountNum=#mydoc.order.customer.XmlAttributes.accountNum#>`	Sets the accountNum variable from the customer entry's accountnum attribute.

Code	Description
<cfset orderquery = QueryNew("item_Id, name, qty, unitPrice") > <cfset temp = QueryAddRow(orderquery, #numItems#)> <cfloop index="i" from = "1" to = #numItems#> <cfset temp = QuerySetCell(orderquery, "item_Id", #mydoc.order.items.item[i]. XmlAttributes.id#, #i#)> <cfset temp = QuerySetCell(orderquery, "name", #mydoc.order.items.item[i]. name.XmlText#, #i#)> <cfset temp = QuerySetCell(orderquery, "qty", #mydoc.order.items.item[i]. quantity.XmlText#, #i#)> <cfset temp = QuerySetCell(orderquery, "unitPrice", #mydoc.order.items.item[i]. unitprice.XmlText#, #i#)> </cfloop>	Converts the XML document object into a query object. Creates a query with columns for the item_id, name, qty, and unitPrice values for each item. For each XML item entry in the mydoc.order.items entry, fills one row of the query with the item's id attribute and the text in the name, quantity, and unitprice entries that the it contains.
<cfquery name="discountQuery" datasource="cfdocexamples"> SELECT * FROM employee WHERE Emp_Id = #accountNum# </cfquery> <cfset drate = 0> <cfif #discountQuery.RecordCount# is 1> <cfset drate = 10> </cfif>	If the account number is the same as an employee ID in the cfdocexamples database Employee table, the query returns one record. and RecordCount equals 1. In this case, sets a discount rate of 10%. Otherwise, sets a discount rate of 0%.
<cfquery name="priceQuery" dbType="query"> SELECT SUM(qty*unitPrice) AS totalPrice FROM orderquery </cfquery> <cfset discountPrice = priceQuery.totalPrice * (1 - drate/100)>	Uses a query of queries with the SUM operator to calculate the total cost before discount of the ordered items, then applies the discount to the price. The result of the query is a single value, the total price.
<cfxml variable="receiptxml"> <receipt num = "34"> <cfoutput> <price>#discountPrice#</price> <cfif drate GT 0 > <discountRate>#drate#</discountRate> </cfif> </cfoutput> <itemsFilled> <cfoutput query="orderQuery"> <name>#name# </name> <qty> #qty# </qty> <price> #qty*unitPrice# </price> </cfoutput> </itemsFilled> </receipt> </cfxml>	Creates an XML document object as a receipt. The receipt has a root element named receipt, which has the receipt number as an attribute. The receipt element contains a price element with the order cost and an itemsFilled element with one item element for each item.

Moving complex data across the web with WDDX

WDDX is an XML vocabulary for describing a complex data structure, such as an array, associative array (such as a ColdFusion structure), or a recordset, in a generic fashion. It lets you use HTTP to move the data between different application server platforms and between application servers and browsers. Target platforms for WDDX include ColdFusion, Active Server Pages (ASP), JavaScript, Perl, Java, Python, COM, Macromedia Flash, and PHP.

The WDDX XML vocabulary consists of a document type definition (DTD) that describes the structure of standard data types and a set of components for each of the target platforms to do the following:

- **Serialize** the data from its native representation into a WDDX XML document or document fragment.
- **Deserialize** a WDDX XML document or document fragment into the native data representation, such as a CFML structure.

This vocabulary creates a way to move data, its associated data types, and descriptors that allow the data to be manipulated on a target system, between arbitrary application servers.

Note: The WDDX DTD, which includes documentation, is located at www.openwddx.org/downloads/dtd/wddx_dtd_10.txt.

WDDX is a valuable tool for ColdFusion developers, however, its usefulness is not limited to CFML. If you serialize a common programming data structure (such as an array, recordset, or structure) into WDDX format, you can use HTTP to transfer the data across a range of languages and platforms. Also, you can use WDDX to store complex data in a database, file, or even a client variable.

WDDX has two features that make it useful for transferring data in a web environment:

- It is lightweight. The JavaScript used to serialize and deserialize data, including a debugging function to dump WDDX data, occupies less than 22K.
- Unlike traditional client-server approaches, the source and target system can have minimal-to-no prior knowledge of each other. They only need to know the structure of the data that is being transferred.

WDDX was created in 1998, and many applications now expose WDDX capabilities. The best source of information about WDDX is www.openwddx.org. This site offers free downloads of the WDDX DTD and SDK and a number of resources, including a WDDX FAQ, a developer forum, and links to additional sites that provide WDDX resources.

Uses of WDDX

WDDX is useful for transferring complex data between applications. For example, you can use it to exchange data between a CFML application and a CGI or PHP application. WDDX is also useful for transferring data between the server and client-side JavaScript.

Exchanging data across application servers

WDDX is useful for the transfer of complex, structured data seamlessly between different application server platforms. For example, an application based on ColdFusion at one business could use `cfwddx` to convert a purchase order structure to WDDX. It could then use `cfhttp` to send the WDDX to a supplier running a CGI-based system.

The supplier could then deserialize the WDDX to its native data form, the extract information from the order, and pass it to a shipping company running an application based on ASP.

Transferring data between the server and browser

You can use WDDX for server-to-browser and browser-to-server data exchanges. You can transfer server data to the browser in WDDX format and convert it to JavaScript objects on the browser. Similarly, your application pages can serialize JavaScript data generated on the browser into WDDX format and transfer the data to the application server. You then deserialize the WDDX XML into CFML data on the server.

On the server you use the `cfwddx` tag to serialize and deserialize WDDX data. On the browser, you use WddxSerializer and WddxRecordset JavaScript utility classes to serialize the JavaScript data to WDDX. (ColdFusion installs these utility classes on your server as webroot/CFIDE/scripts/wddx.js.)

WDDX and web services

WDDX does not compete with web services. It is a complementary technology focused on solving simple problems of application integration by sharing data on the web in a pragmatic, productive manner at very low cost.

WDDX offers the following advantages:

- It can be used by lightweight clients, such as browsers or the Macromedia Flash player.
- It can be used to store complex data structures in files and databases.

Applications that take advantage of WDDX can continue to do so if they start to use web services. These applications could also be converted to use web services standards exclusively; only the service and data interchange formats—not the application model—must change.

How WDDX works

The following example shows how WDDX works. A simple structure with two string variables might have the following form after it is serialized into a WDDX XML representation:

```
<var name='x'>
  <struct>

    <var name='a'>
      <string>Property a</string>
    </var>
    <var name='b'>
      <string>Property b</string>
    </var>
```

```
  </struct>
</var>
```

When you deserialize this XML into CFML or JavaScript, the result is a structure that is created by either of the following scripts:

JavaScript	CFScript
x = new Object(); x.a = "Property a"; x.b = "Property b";	x = structNew(); x.a = "Property a"; x.b = "Property b";

Conversely, when you serialize the variable x produced by either of these scripts into WDDX, you generate the XML listed above.

ColdFusion provides a tag and JavaScript objects that convert between CFML, WDDX, and JavaScript. Serializers and deserializers for other data formats are available on the web. For more information, see www.openwddx.org.

Note: The `cfwddx` tag and the wddx.js JavaScript functions use UTF-8 encoding to represent data. Any tools that deserialize ColdFusion-generated WDDX must accept UTF-8 encoded characters. UTF-8 encoding is identical to the ASCII and ISO 8859 single-byte encodings for the standard 128 "7-bit" ASCII characters. However, UTF-8 uses a two-byte representation for "high-ASCII" ISO 8859 characters where the initial bit is 1.

WDDX data type support

The following sections describe the data types that WDDX supports. This information is a distillation of the description in the WDDX DTD. For more detailed information, see the DTD at www.openwddx.org.

Basic data types

WDDX can represent the following basic data types:

Data type	Description
Null	Null values in WDDX are not associated with a type such as number or string. The `cfwddx` tag converts WDDX Nulls to empty strings.
Numbers	WDDX documents use floating point numbers to represent all numbers. The range of numbers is restricted to +/-1.7E+/-308. The precision is restricted to 15 digits after the decimal point.
Date-time values	Date-time values are encoded according to the full form of ISO8601; for example, 2002-9-15T09:05:32+4:0.
Strings	Strings can be of arbitrary length and must not contain embedded nulls. Strings can be encoded using double-byte characters.

Complex data types

WDDX can represent the following complex data types:

Data type	Description
Array	Arrays are integer-indexed collections of objects of arbitrary type. Because most languages start array indexes at 0, while CFML array indexes start at 1, working with array indices can lead to nonportable data.
Structure	Structures are string-indexed collections of objects of arbitrary type, sometimes called associative arrays. Because some of the languages supported by WDDX are not case-sensitive, no two variable names in a structure can differ only in their case.
Recordset	Recordsets are tabular rows of named fields, corresponding to ColdFusion query objects. Only simple data types can be stored in recordsets. Because some of the languages supported by WDDX are not case-sensitive, no two field names in a recordset can differ only in their case. Field names must satisfy the regular expression [_A-Za-z][_.0-9A-Za-z]* where the period (.) stands for a literal period character, not "any character".
Binary	The binary data type represents strings (blobs) of binary data. The data is encoded in MIME base64 format.

Data type comparisons

The following table compares the basic WDDX data types with the data types to which they correspond in the languages and technologies commonly used on the web:

WDDX	CFML	XML Schema	Java	ECMAScript/ JavaScript	COM
null	N/A	N/A	null	null	VT_NULL
boolean	Boolean	boolean	java.lang.Boolean	boolean	VT_BOOL
number	Number	number	java.lang.Double	number	VT_R8
dateTime	DateTime	dateTime	java.lang.Date	Date	VT_DATE
string	String	string	java.lang.String	string	VT_BSTR
array	Array	N/A	java.lang.Vector	Array	VT_ARRAY | VT_VARIANT
struct	Structure	N/A	java.lang. Hashtable	Object	IWDDXStruct
recordset	Query object	N/A	coldfusion.run time.QueryTable	WddxRecordset	IWDDXRecordset
binary	Binary	binary	byte[]	WddxBinary	V_ARRAY | UI1

Time zone processing

Producers and consumers of WDDX packets can be in geographically dispersed locations. Therefore, it is important to use time zone information when serializing and deserializing data, to ensure that date-time values are represented correctly.

The `cfwddx action=cfml2wddx` tag `useTimezoneInfo` attribute specifies whether to use time zone information in serializing the date-time data. In the JavaScript implementation, `useTimezoneInfo` is a property of the `WddxSerializer` object. In both cases the default `useTimezoneInfo` value is `True`.

Date-time values in WDDX are represented using a subset of the ISO8601 format. Time zone information is represented as an hour/minute offset from Coordinated Universal Time (UTC); for example, "2002-9-8T12:6:26-4:0".

When the `cfwddx` tag deserializes WDDX to CFML, it automatically uses available time zone information, and converts date-time values to local time. In this way, you do not need to worry about the details of time zone conversions.

However, when the JavaScript objects supplied with ColdFusion deserialize WDDX to JavaScript expressions, they do not use time zone information, because in JavaScript it is difficult to determine the time zone of the browser.

Using WDDX

The following sections describe how you can use WDDX in ColdFusion applications. The first two sections describe the tools that ColdFusion provides for creating and converting WDDX. The remaining sections show how you use these tools for common application uses.

Using the cfwddx tag

The `cfwddx` tag can do the following conversions:

From	To
CFML	WDDX
CFML	JavaScript
WDDX	CFML
WDDX	JavaScript

A typical `cfwddx` tag used to convert a CFML query object to WDDX looks like the following:

```
<cfwddx action="cfml2wddx" input="#MyQueryObject#" output="WddxTextVariable">
```

In this example, MyQueryObject is the name of the query object variable, and WddxTextVariable is the name of the variable in which to store the resulting WDDX XML. Note

For more information on the `cfwddx` tag, see *CFML Reference*.

Validating WDDX data

The `cfwddx` tag has a `Validate` attribute that you can use when converting WDDX to CFML or JavaScript. When you set this attribute to True, the XML parser uses the WDDX DTD to validate the WDDX data before deserializing it. If the WDDX is not valid, ColdFusion generates an error. By default, ColdFusion does not validate WDDX data before trying to convert it to ColdFusion or JavaScript data.

The `IsWDDX` function returns True if a variable is a valid WDDX data packet. It returns False otherwise. You can use this function to validate WDDX packets before converting them to another format. For example, you can use it instead of the `cfwddx validate` attribute, so that invalid WDDX is handled within conditional logic instead of error-handling code. You can also use it to pre-validate data that will be deserialized by JavaScript at the browser.

Using JavaScript objects

ColdFusion provides two JavaScript objects, `WddxSerializer object` and `WddxRecordset object`, that you can use in JavaScript to convert data to WDDX. These objects are defined in the file *webroot*/cfide/scripts/wddx.js.

CFML Reference describes these objects and their methods in detail. The example "Transferring data from the browser to the server" on page 881 shows how you can use these objects to serialize JavaScript to WDDX.

Converting CFML data to a JavaScript object

The following example demonstrates the transfer of a `cfquery` recordset from a ColdFusion page executing on the server to a JavaScript object that is processed by the browser.

The application consists of four principal sections:

- Running a data query
- Including the WDDX JavaScript utility classes
- Calling the conversion function
- Writing the object data in HTML

The following example uses the cfdocexamples data source that is installed with ColdFusion:

```
<!--- Create a simple query --->
<cfquery name = "q" datasource ="cfdocexamples">
  SELECT Message_Id, Thread_id, Username, Posted
  FROM messages
</cfquery>

<!--- Load the wddx.js file, which includes the dump function --->
<script type="text/javascript" src="/CFIDE/scripts/wddx.js"></script>

<script>
  // Use WDDX to move from CFML data to JavaScript
  <cfwddx action="cfml2js" input="#q#" topLevelVariable="qj">

  // Dump the recordset to show that all the data has reached
  // the client successfully.
  document.write(qj.dump(true));
</script>
```

Note: To see how `cfwddx Action="cfml2js"` works, save this code under your webroot directory, for example in wwwroot/myapps/wddxjavascript.cfm, run the page in your browser and select View Source in your browser.

Transferring data from the browser to the server

The following example serializes form field data, posts it to the server, deserializes it, and displays the data. For simplicity, it only collects a small amount of data. In applications that generate complex JavaScript data collections, you can extend this basic approach very effectively. This example uses the `WddxSerializer` JavaScript object to serialize the data, and the `cfwddx` tag to deserialize the data.

To use the example:

1. Save the file under your webroot directory, for example in wwwroot/myapps/wddxserializedeserialze.cfm.
2. Display http://localhost/myapps/wddxserializedeserialze.cfm in your browser.
3. Enter a first name and last name in the form fields.
4. Click Next.

   The name appears in the Names added so far box.
5. Repeat steps 3 and 4 to add as many names as you wish.
6. Click Serialize to serialize the resulting data.

   The resulting WDDX packet appears in the WDDX packet display box. This step is intended only for test purposes. Real applications handle the serialization automatically.
7. Click Submit to submit the data.

   The WDDX packet is transferred to the server-side processing code, which deserializes it and displays the information.

```
<!--- load the wddx.js file --->
<script type="text/javascript" src="/CFIDE/scripts/wddx.js"></script>

<!--- Data binding code --->
<script>

  // Generic serialization to a form field
  function serializeData(data, formField) {
    wddxSerializer = new WddxSerializer();
    wddxPacket = wddxSerializer.serialize(data);
    if (wddxPacket != null) {
      formField.value = wddxPacket;
    }
    else {
      alert("Couldn't serialize data");
    }
  }
  // Person info recordset with columns firstName and lastName
  // Make sure the case of field names is preserved
  var personInfo = new WddxRecordset(new Array("firstName", "lastName"),
  true);

  // Add next record to end of personInfo recordset
  function doNext() {
    // Extract data
```

```
    var firstName = document.personForm.firstName.value;
    var lastName = document.personForm.lastName.value;

    // Add names to recordset
    nRows = personInfo.getRowCount();
    personInfo.firstName[nRows] = firstName;
    personInfo.lastName[nRows] = lastName;

    // Clear input fields
    document.personForm.firstName.value = "";
    document.personForm.lastName.value = "";

    // Show added names on list
    // This gets a little tricky because of browser differences
    var newName = firstName + " " + lastName;
    if (navigator.appVersion.indexOf("MSIE") == -1) {
      document.personForm.names[length] =
        new Option(newName, "", false, false);
    }
    else {
      // IE version
      var entry = document.createElement("OPTION");
      entry.text = newName;
      document.personForm.names.add(entry);
    }
  }
</script>

<!--- Data collection form --->
<form action="#cgi.script_name#" method="Post" name="personForm">

  <!--- Input fields --->
  Personal information<br>
  First name: <input type=text name=firstName><br>
  Last name: <input type=text name=lastName><br>
  <br>

  <!--- Navigation & submission bar --->
  <input type="button" value="Next" onclick="doNext()">
  <input type="button" value="Serialize"
  onclick="serializeData(personInfo, document.personForm.wddxPacket)">
  <input type="submit" value="Submit">
  <br><br>
  Names added so far:<br>
  <select name="names" size="5">
  </select>
  <br>

  <!--- This is where the WDDX packet will be stored --->
  <!--- In a real application this would be a hidden input field. --->
  <br>
  WDDX packet display:<br>
  <textarea name="wddxPacket" rows="10" cols="80" wrap="Virtual">
  </textarea>
```

```
</form>


<!--- Server-side processing --->
<hr>
<b>Server-side processing</b><br>
<br>
<cfif isdefined("form.wddxPacket")>
  <cfif form.wddxPacket neq "">

    <!--- Deserialize the WDDX data --->
    <cfwddx action="wddx2cfml" input=#form.wddxPacket#
    output="personInfo">

    <!--- Display the query --->
    The submitted personal information is:<br>
    <cfoutput query=personInfo>
      Person #CurrentRow#: #firstName# #lastName#<br>
    </cfoutput>
  <cfelse>
    The client did not send a well-formed WDDX data packet!
  </cfif>
<cfelse>
  No WDDX data to process at this time.
</cfif>
```

Storing complex data in a string

The following simple example uses WDDX to store complex data, a data structure that contains arrays as a string in a client variable. It uses the `cfdump` tag to display the contents of the structure before serialization and after deserialization. It uses the `HTMLEditFormat` function in a `cfoutput` tag to display the contents of the client variable. The `HTMLEditFormat` function is required to prevent the browser from trying to interpret (and throwing away) the XML tags in the variable.

```
<!--- Enable client state management --->
<cfapplication name="relatives" clientmanagement="Yes">

<!--- Build a complex data structure --->
<cfscript>
  relatives = structNew();
  relatives.father = "Bob";
  relatives.mother = "Mary";
  relatives.sisters = arrayNew(1);
  arrayAppend(relatives.sisters, "Joan");
  relatives.brothers = arrayNew(1);
  arrayAppend(relatives.brothers, "Tom");
  arrayAppend(relatives.brothers, "Jesse");
</cfscript>

A dump of the original relatives structure:<br>
<br>
<cfdump var="#relatives#"><br>
<br>
```

```
<!--- Convert data structure to string form and save it in the client scope --
  ->
<cfwddx action="cfml2wddx" input="#relatives#" output="Client.wddxRelatives">

The contents of the Client.wddxRelatives variable:<br>
<cfoutput>#HtmlEditFormat(Client.wddxRelatives)#</cfoutput><br>

<!--- Now read the data from client scope into a new structure --->
<cfwddx action="wddx2cfml" input="#Client.wddxRelatives#"
  output="sameRelatives">
<br>
A dump of the sameRelatives structure generated from client.wddxRelatives<br>
<cfdump var="#sameRelatives#">
```

CHAPTER 36
Using Web Services

Web services let you publish and consume remote application functionality over the Internet. When you consume web services, you access remote functionality to perform an application task. When you publish a web service, you let remote users access your application functionality to build it into their own applications.

This chapter describes how to consume and publish web services.

Contents

Web services 885

Working with WSDL files 888

Consuming web services 891

Publishing web services 898

Handling complex data types 909

Troubleshooting SOAP requests and responses 914

Web services

Since its inception, the Internet has allowed people to access content stored on remote computers. This content can be static, such as a document represented by an HTML file, or dynamic, such as content returned from a Macromedia ColdFusion page or CGI script.

Web services let you access application functionality that someone created and made available on a remote computer. With a web service, you can make a request to the remote application to perform an action.

For example, you can request a stock quote, pass a text string to be translated, or request information from a product catalog. The advantage of web services is that you do not have to recreate application logic that someone else has already created and, therefore, you can build your applications faster.

Referencing a remote web service within your ColdFusion application is called *consuming* web services. Since web services adhere to a standard interface regardless of implementation technology, you can consume a web service implemented as part of a ColdFusion application, or as part of a .NET or Java application.

You can also create your own web services and make them available to others for remote access, called *publishing* web service. Applications that consume your web service can be implemented in ColdFusion MX or by any application that recognizes the web service standard.

Accessing a web service

In its simplest form, an access to a web service is similar to a function call. Instead of the function call referencing a library on your computer, it references remote functionality over the Internet.

One feature of web services is that they are *self-describing*. That means a person who makes a web service available also publishes a description of the API to the web service as a Web Services Description Language (WSDL) file.

A WSDL file is an XML-formatted document that includes information about the web service, including the following information:

- Operations that you can call on the web service
- Input parameters that you pass to each operation
- Return values from an operation

Consuming web services typically is a two-step process:

1. Parse the WSDL file of the web service to determine its interface.

   A web service makes its associated WSDL file available over the Internet. You must know the URL of the WSDL file defining the service. For example, you can access the WSDL file for the TemperatureService web service at the following URL:

   www.xmethods.net/sd/2001/TemperatureService.wsdl

   For an overview of WSDL syntax, see "Working with WSDL files" on page 888

2. Make a request to the web service.

   The following example invokes an operation on the Temperature web service to retrieve the temperature in zip code 55987:

```
<cfinvoke
  webservice="http://www.xmethods.net/sd/2001/TemperatureService.wsdl"
  method="getTemp"
  returnvariable="aTemp">
    <cfinvokeargument name="zipcode" value="55987"/>
</cfinvoke>
<cfoutput>The temperature at zip code 55987 is #aTemp#</cfoutput>
```

   For more information on consuming web services, see "Consuming web services" on page 891.

Basic web service concepts

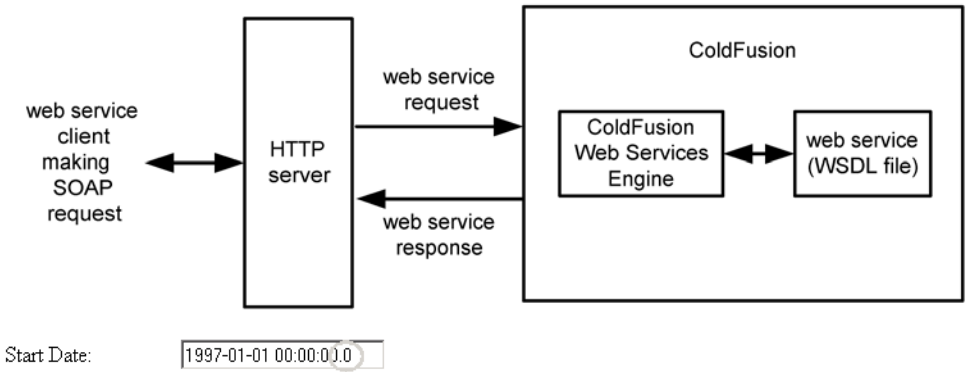
You must be familiar with the underlying architecture of a web service provider in order to fully understand how web services work.

Note: This section contains an overview of the architecture of web services. For detailed information, consult one of the many web services books.

The following are three primary components of the web services platform:

- SOAP (Simple Object Access Protocol)
- WSDL (Web Services Description Language)
- UDDI (Universal Description, Discovery, and Integration)

The following simple figure shows how the ColdFusion MX implementation of web services work:



The following sections describe the components shown in this figure.

Supporting web services with SOAP

SOAP provides a standard XML structure for sending and receiving web service requests and responses over the Internet. Usually you send SOAP messages using HTTP, but you also can send them using SMTP and other protocols. ColdFusion MX integrates the Apache Axis SOAP engine to support web services.

The ColdFusion Web Services Engine performs the underlying functionality to support web services, including generating WSDL files for web services that you create. In ColdFusion MX, to consume or publish web services does not require you to be familiar with SOAP or to perform any SOAP operations.

You can find additional information about SOAP in the W3C's SOAP 1.1 note at www.w3.org/TR/SOAP/.

Describing web services with WSDL

A WSDL document is an XML file that describes a web service's purpose, where it is located, and how to access it. The WSDL document describes the operations that you can invoke and their associated data types.

ColdFusion MX can generate a WSDL document from a web service, and you can publish the WSDL document at a URL to provide information to potential clients. For more information, see "Working with WSDL files" on page 888.

Finding web services with UDDI

As a consumer of web services, you want to know what web services are available. As a publisher of web services, you want others to be able to find information about your web services. Universal Description, Discovery and Integration (UDDI) provides a way for web service clients to dynamically locate web services that provide specific capabilities. You use a UDDI query to find service providers. A UDDI response contains information, such as business contact information, business category, and technical details, about how to invoke a web service.

Although ColdFusion MX does not directly support UDDI, you can manually register or find a web service using a public UDDI registry, such as the IBM UDDI Business Registry at https://www-3.ibm.com/services/uddi/protect/registry.html.

You can find additional information about UDDI at www.uddi.org/about.htm.

Working with WSDL files

WSDL files define the interface to a web service. To consume a web service, you access the service's WSDL file to determine information about it. If you publish your application logic as a web service, you must create a WSDL file for it.

WSDL is a draft standard supported by the World Wide Web Consortium. You can access the specification at www.w3.org/TR/wsdl.

Creating a WSDL file

To publish a web service, you construct the service's functionality and then create the WSDL file defining the service. In ColdFusion MX, you use components to create web services. ColdFusion automatically generates the WSDL file for a component that you use to produce a web service. For more information on creating web services, see "Publishing web services" on page 898.

For more information on components, see Chapter 10, "Building and Using ColdFusion Components," on page 201.

Accessing web services using Macromedia Dreamweaver MX 2004

The Dreamweaver MX 2004 Components tab lets you view web services, including operation names, parameter names, and parameter data types.

To open the Components tab in the Dreamweaver MX 2004 and add a web service:

1. Select Window > Components, or use Control+F7, to open the Components panel.
2. In the Components panel, select Web Services from the drop-down list in the upper-left of the panel.
3. Click the Plus (+) button.

   The Add Using WSDL dialog box appears.
4. Specify the URL of the WSDL file.

After the web service is defined to Dreamweaver MX 2004, you can drag it onto a page to call it using the `cfinvoke` tag.

For more information on using Dreamweaver MX 2004, see its online Help system.

Note: The Web Services option is not available if you are running Dreamweaver MX 2004 on the Macintosh. However, you can still use web services by writing code manually.

Reading a WSDL file

A WSDL file takes practice to read. You can view the WSDL file in a browser, or you can use a tool such as Dreamweaver MX 2004, which contains a built-in utility for displaying WSDL files in an easy-to-read format.

The following example shows a WSDL file for the TemperatureService web service:

```
<?xml version="1.0"?>
<definitions name="TemperatureService" targetNamespace="http://
  www.xmethods.net/sd/TemperatureService.wsdl"  xmlns:tns="http://
  www.xmethods.net/sd/TemperatureService.wsdl"   xmlns:xsd="http://www.w3.org/
  2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="getTempRequest">
    <part name="zipcode" type="xsd:string"/>
  </message>
  <message name="getTempResponse">
    <part name="return" type="xsd:float"/>
  </message>
  <portType name="TemperaturePortType">
    <operation name="getTemp">
      <input message="tns:getTempRequest"/>
      <output message="tns:getTempResponse"/>
    </operation>
  </portType>
  <binding name="TemperatureBinding" type="tns:TemperaturePortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/
  http"/>
    <operation name="getTemp">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded" namespace="urn:xmethods-Temperature"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
```

```
        <soap:body use="encoded" namespace="urn:xmethods-Temperature"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>
  <service name="TemperatureService">
    <documentation>Returns current temperature in a given U.S. zipcode  </
  documentation>
    <port name="TemperaturePort" binding="tns:TemperatureBinding">
      <soap:address location="http://services.xmethods.net:80/soap/servlet/
  rpcrouter"/>
    </port>
  </service>
</definitions>
```

The following are the major components of the WSDL file:

Component	Definition
`definitions`	The root element of the WSDL file. This area contains namespace definitions that you use to avoid naming conflicts between multiple web services.
`types`	(*Not shown*) Defines data types used by the service's messages.
`message`	Defines the data transferred by a web service operation, typically the name and data type of input parameters and return values.
`port type`	Defines one or more operations provided by the web service.
`operation`	Defines an operation that can be remotely invoked.
`input`	Specifies an input parameter to the operation using a previously defined message.
`output`	Specifies the return values from the operation using a previously defined message.
`fault`	(*not shown*) Optionally specifies an error message returned from the operation.
`binding`	Specifies the protocol used to access a web service including SOAP, HTTP GET and POST, and MIME.
`service`	Defines a group of related operations.
`port`	Defines an operation and its associated inputs and outputs.

For additional descriptions of the contents of this WSDL file, see "Consuming web services" on page 891.

Consuming web services

ColdFusion MX provides a variety of methods for consuming web services. The method that you choose depends on your ColdFusion programming style and application.

The following table describes these methods:

Method	CFML operator	Description
CFScript	`CreateObject()`	Consumes a web service from within a CFScript block.
CFML tag	`cfinvoke`	Consumes a web service from within a block of CFML code.
CFML tag	`cfobject`	Consumes a web service from within a block of CFML code.

One important consideration is that all consumption methods use the same underlying technology and offer the same performance.

About the examples in this section

The examples in this section reference the TemperatureService web service from XMethods. This web service returns the temperature for a given zip code. You can read the WSDL file for this web service in "Reading a WSDL file" on page 889.

The TemperatureService web service has one input parameter, a string that contains the requested zip code. It returns a float that contains the temperature for the specified zip code.

Passing parameters to a web service

The message and operation elements in the WSDL file contains subelements that define the web service operations and the input and output parameters of each operation, including the data type of each parameter. The following example shows a portion of the WSDL file for the TemperatureService web service:

```
<message name="getTempRequest">
  <part name="zipcode" type="xsd:string"/>
</message>
<message name="getTempResponse">
  <part name="return" type="xsd:float"/>
</message>
<portType name="TemperaturePortType">
  <operation name="getTemp">
    <input message="tns:getTempRequest"/>
    <output message="tns:getTempResponse"/>
  </operation>
</portType>
```

The operation name used in the examples in this section is getTemp. This operation takes a single input parameter defined as a message of type getTempRequest.

You can see that the message element named getTempRequest contains one string parameter: `zipcode`. When you call the getTemp operation, you pass the parameter as input.

Handling return values from a web service

Web service operations often return information back to your application. You can determine the name and data type of returned information by examining subelements of the message and operation elements in the WSDL file.

The following example shows a portion of the WSDL file for the TemperatureService web service:

```
<message name="getTempRequest">
  <part name="zipcode" type="xsd:string"/>
</message>
<message name="getTempResponse">
  <part name="return" type="xsd:float"/>
</message>
<portType name="TemperaturePortType">
  <operation name="getTemp">
    <input message="tns:getTempRequest"/>
    <output message="tns:getTempResponse"/>
  </operation>
</portType>
```

The operation getTemp returns a message of type getTempResponse. The message statement in the WSDL file defines the getTempResponse message as containing a single string parameter named `return`.

Using cfinvoke to consume a web service

This section describes how to consume a web service using the `cfinvoke` tag. With the `cfinvoke` tag, you reference the WSDL file and invoke an operation on the web service with a single tag.

The `cfinvoke` tag includes attributes that specify the URL to the WSDL file, the method to invoke, the return variable, and input parameters. For complete `cfinvoke` syntax, see *CFML Reference*.

Note: You can pass parameters to a web service using the `cfinvokeargument` tag or by specifying parameter names in the `cfinvoke` tag itself. For more information, see "Passing parameters to methods using the cfinvoke tag" on page 225.

To access a web service using cfinvoke:

1. Create a ColdFusion page with the following content:

```
<cfinvoke
  webservice="http://www.xmethods.net/sd/2001/TemperatureService.wsdl"
  method="getTemp"
  returnvariable="aTemp">
    <cfinvokeargument name="zipcode" value="55987"/>
</cfinvoke>
<cfoutput>The temperature at zip code 55987 is #aTemp#</cfoutput>
```

2. Save the page as wscfc.cfm in your web root directory.
3. View the page in your browser.

You can omit a parameter by setting the `cfinvokeargument omit` attribute to `"yes"`. If the WSDL specifies that the argument is nillable, ColdFusion MX sets the associated argument to null. If the WSDL specifies minoccurs=0, ColdFusion MX omits the argument from the WSDL. However, CFC web services must still specify `required="true"` for all arguments.

You can also use an attribute collection to pass parameters. An attribute collections is a structure where each structure key corresponds to a parameter name and each structure value is the parameter value passed for the corresponding key. The following example shows an invocation of a web service using an attribute collection:

```
<cfscript>
  stArgs = structNew();
  stArgs.zipcode = "55987";
</cfscript>

<cfinvoke
  webservice="http://www.xmethods.net/sd/2001/TemperatureService.wsdl"
  method="getTemp"
  argumentcollection="#stArgs#"
  returnvariable="aTemp">
<cfoutput>The temperature at zip code 55987 is #aTemp#</cfoutput>
```

In this example, you create the structure in a CFScript block, but you can use any ColdFusion method to create the structure.

Using CFScript to consume a web service

The example in this section uses CFScript to consume a web service. In CFScript, you use the `CreateObject` function to connect to the web service. After connecting, you can make requests to the service. For `CreateObject` syntax, see *CFML Reference*.

After creating the web service object, you can call operations of the web service using dot notation, in the following form:

```
webServiceName.operationName(inputVal1, inputVal2, ... )
```

You can handle return values from web services by writing them to a variable, as the following example shows:

```
resultVar = webServiceName.operationName(inputVal1, inputVal2, ... );
```

Or, you can pass the return values directly to a function, such as the `WriteOutput` function, as the following example shows:

```
writeoutput(webServiceName.operationName(inputVal1, inputVal2, ...) );
```

To access a web service from CFScript:

1. Create a ColdFusion page with the following content:

```
<cfscript>
  ws = CreateObject("webservice",
    "http://www.xmethods.net/sd/2001/TemperatureService.wsdl");
  xlatstring = ws.getTemp("55987");
  writeoutput(xlatstring);
</cfscript>
```

2. Save the page as wscfscript.cfm in your web root directory.
3. View the page in your browser.

You can also use named parameters to pass information to a web service. The following example performs the same operation as above, except that it uses named parameters to make the web service request:

```
<cfscript>
  ws = CreateObject("webservice",
    "http://www.xmethods.net/sd/2001/TemperatureService.wsdl");
  xlatstring = ws.getTemp(zipcode = "55987");
  writeoutput("The temperature at 55987 is " & xlatstring);
</cfscript>
```

Consuming web services that are not generated by Macromedia ColdFusion MX

To consume a web service that is implemented in a technology other than ColdFusion MX, the web service must have one of the following sets of options:

- rpc as the SOAP binding style and encoding as the encodingStyle
- document as the SOAP binding style and literal as the encodingStyle

The following example shows a portion of the WSDL file for the TemperatureService web service:

```
<binding name="TemperatureBinding" type="tns:TemperaturePortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/
  >
  <operation name="getTemp">
    <soap:operation soapAction=""/>
    <input>
      <soap:body use="encoded" namespace="urn:xmethods-Temperature"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <soap:body use="encoded" namespace="urn:xmethods-Temperature"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>
```

The WSDL file for the TemperatureService web service is compatible with ColdFusion MX because it uses rpc as the binding style, and encoding as the encodingStyle.

Calling web services from a Macromedia Flash client

The Flash Remoting service lets you call ColdFusion pages from a Flash client, but it does not let you call web services directly. To call web services from a Flash client, you can use Flash Remoting to call a ColdFusion component that calls the web service. The Flash client can pass input parameters to the component, and the component can return to the Flash client any data returned by the web service.

For more information, see Chapter 33, "Using the Flash Remoting Service," on page 807.

Catching errors when consuming web services

Web services might throw errors, including SOAP faults, during processing that you can catch in your application. If uncaught, these errors propagate to the browser.

To catch errors, you specify an error type of application to the ColdFusion `cfcatch` tag, as the following example shows:

```
<cftry>
  Put your application code here ...
  <cfcatch type="application">
    <!--- Add exception processing code here ... --->
  </cfcatch>
  ...
  <cfcatch type="Any">
    <!--- Add exception processing code appropriate for all other
  exceptions here ... --->
  </cfcatch>
</cftry>
```

For more information on error handling, see Chapter 14, "Handling Errors," on page 307.

Handling inout and out parameters

Some web services define inout and out parameters. You use *out* parameters to pass a placeholder for a return value to a web service. The web service then returns its result by writing it to the out parameter. *Inout* parameters let you pass a value to a web service and lets the web service return its result by overwriting the parameter value.

The following example shows a web service that takes as input an inout parameter containing a string and writes its results back to the string:

```
<cfset S="foo">
<cfscript>
  ws=createobject("webservice", "URLtoWSDL")
  ws.modifyString("S");
<cfscript>
<cfoutput>#S#</cfoutput>
```

Even though this web service takes as input the value of S, because you pass it as an inout parameter, you do not enclose it in number signs.

Note: ColdFusion MX supports the use of inout and out parameters to consume web services. However, ColdFusion MX does not support inout and out parameters when creating web services for publication.

Configuring web services in the ColdFusion MX Administrator

The ColdFusion MX Administrator lets you register web services so that you do not have to specify the entire WSDL URL when you reference the web service.

Note: The first time you reference a web service, ColdFusion MX automatically registers it in the Administrator.

For example, the following code references the URL to the TemperatureService WSDL file:

```
<cfscript>
  ws = CreateObject("webservice",
    "http://www.xmethods.net/sd/2001/TemperatureService.wsdl");
  xlatstring = ws.getTemp("55987");
  writeoutput(xlatstring);
</cfscript>
```

If you register the TemperatureService web service in the Administrator using (for example, the name wsTemp), you can then reference the web service as follows:

```
<cfscript>
  ws = CreateObject("webservice", "wsTemp");
  xlatstring = ws.getTemp("55987");
  writeoutput("wsTemp: " & xlatstring);
</cfscript>
```

Not only does this enable you to shorten your code, registering a web service in the Administrator lets you change a web service's URL without modifying your code. So, if the TemperatureService web service moves to a new location, you only update the administrator setting, not your application code.

For more information, see the ColdFusion MX Administrator online Help.

Data conversions between ColdFusion and WSDL data types

A WSDL file defines the input and return parameters of an operation, including data types. For example, the TemperatureService web service contains the following definition of input and return parameters:

```
<message name="getTempRequest">
  <part name="zipcode" type="xsd:string"/>
</message>
<message name="getTempResponse">
  <part name="return" type="xsd:float"/>
</message>
```

As part of consuming web services, you must understand how ColdFusion MX converts WSDL defined data types to ColdFusion data types. The following table shows this conversion:

ColdFusion data type	WSDL data type
numeric	SOAP-ENC:double
boolean	SOAP-ENC:boolean
string	SOAP-ENC:string

ColdFusion data type	WSDL data type
array	SOAP-ENC:Array
binary	xsd:base64Binary
numeric	xsd:float
string	xsd:enumeration
date	xsd:dateTime
void (operation returns nothing)	
struct	complex type
query	tns1:QueryBean (Returned by CFCs)

For many of the most common data types, such as string and numeric, a WSDL data type maps directly to a ColdFusion data type. For complex WSDL data types, the mapping is not as straight forward. In many cases, you map a complex WSDL data type to a ColdFusion structure. For more information on handling complex data types, see "Handling complex data types" on page 909.

Consuming ColdFusion web services

Your application can consume web services created in ColdFusion. You do not have to perform any special processing on the input parameters or return values because ColdFusion handles data mappings automatically when consuming a ColdFusion web service.

For example, when ColdFusion publishes a web service that returns a query, or takes a query as an input, the WSDL file for that service lists its data type as QueryBean. However, a ColdFusion application consuming this web service can pass a ColdFusion query object to the function as an input, or write a returned QueryBean to a ColdFusion query object.

Note: For a list of how ColdFusion data types map to WSDL data types, see "Data conversions between ColdFusion and WSDL data types" on page 896.

The following example shows a ColdFusion component that takes a query as input and echoes the query back to the caller:

```
<cfcomponent>
  <cffunction name='echoQuery' returnType='query' access='remote'>
    <cfargument name='input' type='query'>
    <cfreturn #arguments.input#>
  </cffunction>
</cfcomponent>
```

In the WSDL file for the echotypes.cfc component, you see the following definitions that specify the type of the function's input and output as QueryBean:

```
<wsdl:message name="echoQueryResponse">
  <wsdl:part name="echoQueryReturn" type="tns1:QueryBean"/>
</wsdl:message>
<wsdl:message name="echoQueryRequest">
  <wsdl:part name="input" type="tns1:QueryBean"/>
</wsdl:message>
```

For information on displaying WSDL, see "Producing WSDL files" on page 900.

Since ColdFusion automatically handles mappings to ColdFusion data types, you can call this web service as the following example shows:

```
<head>
<title>Passing queries to web services</title>
</head>
<body>
<cfquery name="GetEmployees" datasource="cfdocexamples">
  SELECT FirstName, LastName, Salary
  FROM Employee
</cfquery>

<cfinvoke
  webservice = "http://localhost/echotypes.cfc?wsdl"
  method = "echoQuery"
  input="#GetEmployees#"
  returnVariable = "returnedQuery">

<cfoutput>
  Is returned result a query? #isQuery(returnedQuery)# <br><br>
</cfoutput>

<cfoutput query="returnedQuery">
  #FirstName#, #LastName#, #Salary#<br>
</cfoutput>
</body>
```

Publishing web services

To publish web services for consumption by remote applications, you create the web service using ColdFusion components. For more information on components, see Chapter 10, "Building and Using ColdFusion Components," on page 201.

Creating components for web services

ColdFusion components (CFCs) encapsulate application functionality and provide a standard interface for client access to that functionality. A component typically contains one or more functions defined by the `cffunction` tag.

For example, the following component contains a single function:

```
<cfcomponent>
  <cffunction name="echoString" returnType="string" output="no">
    <cfargument name="input" type="string">
    <cfreturn #arguments.input#>
  </cffunction>
</cfcomponent>
```

The function, named echoString, echoes back any string passed to it. To publish the function as a web service, you must modify the function definition to add the `access` attribute and specify remote, as the following example shows:

```
<cffunction name="echoString" returnType="string" output="no" access="remote">
```

By defining the function as remote, ColdFusion includes the function in the WSDL file. Only those functions marked as remote are accessible as a web service.

The following list defines the requirements for how to create web services for publication:

1. The value of the `access` attribute of the `cffunction` tag must be `remote`.
2. The `cffunction` tag must include the `returnType` attribute to specify a return type.
3. The `output` attribute of the `cffunction` tag must be set to `No` because ColdFusion converts all output to XML to return it to the consumer.
4. The attribute setting `required="false"` for the `cfargument` tag is ignored. ColdFusion considers all parameters as required.

Specifying data types of function arguments and return values

The `cffunction` tag lets you define a single return value and one or more input parameters passed to a function. As part of the function definition, you include the data type of the return value and input parameters.

The following example shows a component that defines a function with a return value of type string, one input parameter of type string, and one input parameter of type numeric:

```
<cfcomponent>
  <cffunction name="trimString" returnType="string" output="no">
    <cfargument name="inString" type="string">
    <cfargument name="trimLength" type="numeric">
  </cffunction>
</cfcomponent>
```

As part of publishing the component for access as a web service, ColdFusion generates the WSDL file that defines the component where the WSDL file includes definitions for how ColdFusion data types map to WSDL data types. The following table shows this mapping:

ColdFusion data type	WSDL data type published
numeric	SOAP-ENC:double
boolean	SOAP-ENC:boolean
string	SOAP-ENC:string
array	SOAP-ENC:Array
binary	xsd:base64Binary
date	xsd:dateTime
guid	SOAP-ENC:string
uuid	SOAP-ENC:string
void (operation returns nothing)	
struct	Map
query	QueryBean

ColdFusion data type	WSDL data type published
any	complex type
component definition	complex type

In most cases, consumers of ColdFusion web services can easily pass data to and return results from component functions by mapping their data types to the WSDL data types shown in the preceding table.

Note: Document-literal web services use XML schema data types, not SOAP-ENC data types. For more information, see "Publishing document-literal style web services" on page 904.

For ColdFusion structures and queries, clients might have to perform some processing to map their data to the correct type. For more information, see "Publishing web services that use complex data types" on page 912.

You can also define a data type in one ColdFusion component based on another component definition. For more information on using components to specify a data type, see "Using ColdFusion components to define data types for web services" on page 902.

Producing WSDL files

ColdFusion automatically creates a WSDL file for any component referenced as a web service. For example, if you have a component named echo.cfc in your web root directory, you can view its corresponding WSDL file by requesting the component as follows:

```
http://localhost/echo.cfc?wsdl
```

For example, you define a ColdFusion component as follows:

```
<cfcomponent>
  <cffunction
  name = "echoString"
  returnType = "string"
  output = "no"
  access = "remote">
    <cfargument name = "input" type = "string">
    <cfreturn #arguments.input#>
  </cffunction>
</cfcomponent>
```

Tip: If you register the component in Dreamweaver MX 2004, it appears in the Components tab of the Application panel.

Requesting the WSDL file in a browser returns the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://ws"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:apachesoap="http://xml.apache.org/xml-soap"
  xmlns:impl="http://ws"
  xmlns:intf="http://ws"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tns1="http://rpc.xml.coldfusion"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

```
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Macromedia ColdFusion MX version 7,0,0,87956-->
<wsdl:types>
  <schema targetNamespace="http://rpc.xml.coldfusion"
    xmlns="http://www.w3.org/2001/XMLSchema">
  <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
  <complexType name="CFCInvocationException">
    <sequence/>
  </complexType>
  </schema>
</wsdl:types>
<wsdl:message name="CFCInvocationException">
  <wsdl:part name="fault" type="tns1:CFCInvocationException"/>
</wsdl:message>
<wsdl:message name="echoStringResponse">
  <wsdl:part name="echoStringReturn" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="echoStringRequest">
  <wsdl:part name="input" type="xsd:string"/>
 </wsdl:message>
 <wsdl:portType name="echo">
   <wsdl:operation name="echoString" parameterOrder="input">
     <wsdl:input message="impl:echoStringRequest" name="echoStringRequest"/>
     <wsdl:output message="impl:echoStringResponse"
       name="echoStringResponse"/>
     <wsdl:fault message="impl:CFCInvocationException"
name="CFCInvocationException"/>
   </wsdl:operation>
 </wsdl:portType>
 <wsdl:binding name="echo.cfcSoapBinding" type="impl:echo">
   <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/
     http"/>
   <wsdl:operation name="echoString">
     <wsdlsoap:operation soapAction=""/>
     <wsdl:input name="echoStringRequest">
       <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/
         encoding/" namespace="http://ws" use="encoded"/>
     </wsdl:input>
     <wsdl:output name="echoStringResponse">
       <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/
         encoding/" namespace="http://ws" use="encoded"/>
     </wsdl:output>
     <wsdl:fault name="CFCInvocationException">
       <wsdlsoap:fault encodingStyle="http://schemas.xmlsoap.org/soap/
         encoding/" name="CFCInvocationException" namespace=
         "http://ws" use="encoded"/>
     </wsdl:fault>
   </wsdl:operation>
 </wsdl:binding>
 <wsdl:service name="echoService">
   <wsdl:port binding="impl:echo.cfcSoapBinding" name="echo.cfc">
     <wsdlsoap:address location="http://localhost:8500/ws/echo.cfc"/>
   </wsdl:port>
```

```
   </wsdl:service>
</wsdl:definitions>
```

To publish a web service:

1. Create a ColdFusion page with the following content:

```
<cfcomponent output="false">
  <cffunction
       name = "echoString"
       returnType = "string"
       output = "no"
       access = "remote">
    <cfargument name = "input" type = "string">
    <cfreturn #arguments.input#>
  </cffunction>
</cfcomponent>
```

2. Save this file as echo.cfc in your web root directory.
3. Create a ColdFusion page with the following content:

```
<cfinvoke webservice ="http://localhost/echo.cfc?wsdl"
  method ="echoString"
  input = "hello"
  returnVariable="foo">

<cfoutput>#foo#</cfoutput>
```

4. Save this file as echoclient.cfm in your web root directory.
5. Request echoclient.cfm in your browser.

   The following string appears in your browser:

```
hello
```

You can also invoke the web service using the following code:

```
<cfscript>
  ws = CreateObject("webservice", "http://localhost/echo.cfc?wsdl");
  wsresults = ws.echoString("hello");
  writeoutput(wsresults);
</cfscript>
```

Using ColdFusion components to define data types for web services

ColdFusion lets you define components that contain only properties. Once defined, you can use components to define data types for web services. The following code defines a component in the file address.cfc that contains properties that represent a street address:

```
<cfcomponent>
  <cfproperty name="AddrNumber" type="numeric">
  <cfproperty name="Street" type="string">
  <cfproperty name="City" type="string">
  <cfproperty name="State" type="string">
  <cfproperty name="Country" type="string">
</cfcomponent>
```

The following code defines a component in the file name.cfc that defines first and last name properties:

```
<cfcomponent>
  <cfproperty name="Firstname" type="string">
  <cfproperty name="Lastname" type="string">
</cfcomponent>
```

You can then use address and name to define data types in a ColdFusion component created to publish a web service, as the following example shows:

```
<cfcomponent>
  <cffunction
     name="echoName" returnType="name" access="remote" output="false">
       <cfargument name="input" type="name">
       <cfreturn #arguments.input#>
  </cffunction>

  <cffunction
     name="echoAddress" returnType="address" access="remote" output="false">
       <cfargument name="input" type="address">
       <cfreturn #arguments.input#>
  </cffunction>
</cfcomponent>
```

Note: If the component files are not in a directory under your web root, you must create a web server mapping to the directory that contains them. You cannot use ColdFusion mappings to access web services.

The WSDL file for the web service contains data definitions for the complex types name and address. Each definition consists of the elements that define the type as specified in the ColdFusion component file for that type. For example, the following example shows the definition for name:

```
<complexType name="name">
  <sequence>
    <element name="firstname" nillable="true" type="soapenc:string"/>
    <element name="lastname" nillable="true" type="soapenc:string"/>
  </sequence>
</complexType>
```

You can also specify an array of CFCs in the `returnType` attribute, as the following example shows:

```
<cfcomponent>
  <cffunction
     name="allNames" returnType="name[]" access="remote" output="false">
    <cfset var returnarray = ArrayNew(1)>
    <cfset var temp = "">
    <cfquery name="empinfo" datasource="cfdocexamples">
     SELECT firstname, lastname
     FROM employee
    </cfquery>
    <cfloop query="empinfo" >
       <cfobject component="name" name="tempname">
       <cfset tempname.Firstname = #empinfo.firstname#>
```

```
      <cfset tempname.Lastname = #empinfo.lastname#>
      <cfset temp = ArrayAppend(returnarray, tempname)>
    </cfloop>
    <cfreturn returnarray>
  </cffunction>
</cfcomponent>
```

When you invoke the web service, it returns an array of CFCs. Access the properties in the CFC by using dot notation, as the following example shows:

```
<cfinvoke webservice ="http://localhost:8500/ws/cfcarray.cfc?wsdl"
  method ="allNames"
  returnVariable="thearray">

<cfif IsArray(thearray)>
  <h1>loop through the employees</h1>
  <p>thearray has <cfoutput>#ArrayLen(thearray)#</cfoutput> elements.</p>
  <cfloop index="i" from="1" to="#ArrayLen(thearray)#">
   <cfoutput>#thearray[i].firstname#, #thearray[i].lastname# </cfoutput><br>
  </cfloop>
<cfelse>
  <h1>Error: thearray is not an array</h1>
</cfif>
```

Publishing document-literal style web services

In addition to RPC-oriented operations, for which consumers specify a method and arguments, ColdFusion also lets you publish web services using the document-literal style. When you use document-literal style, the WSDL for the web service tells the client to use XML schemas rather than RPC calling conventions.

In most cases, the publisher of a web services identifies it as document-literal or RPC style. To identify the type, open the WSDL document and find the `soap:binding` element and examine its `style` attribute, as the following example shows:

```
<wsdl:binding name="WeatherForecastSoap" type="tns:WeatherForecastSoap">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http"
  style="document" />
```

In this example, the style is document-literal. You must further examine the WSDL to determine the methods you can call and the parameters for each method.

On the client side, the `cfinvoke` tag and other ColdFusion methods for calling web services handle this automatically. In most cases, no modifications are necessary. Similarly, when publishing CFCs as document-literal style web services, ColdFusion automatically creates and manages the appropriate WSDL.

To publish CFCs as document-literal style web services, specify `cfcomponent style="document"`, along with the other attributes required for document-literal style web services. For example, ColdFusion MX publishes the following CFC using document-literal style:

```
<cfcomponent style="document" >
  <cffunction
      name = "getEmp"
      returntype="string"
```

```
      output = "no"
      access = "remote">
  <cfargument name="empid" required="yes" type="numeric">
  <cfset var fullname = "">
  <cfquery name="empinfo" datasource="cfdocexamples">
     SELECT emp_id, firstname, lastname
     FROM employee
     WHERE emp_id = <cfqueryparam cfsqltype="cf_sql_integer"
      value="#arguments.empid#">
  </cfquery>
  <cfif empinfo.recordcount gt 0>
     <cfset fullname = empinfo.lastname & ", " & empinfo.firstname>
  <cfelse>
      <cfset fullname = "not found">
  </cfif>
    <cfreturn #fullname#>
  </cffunction>
</cfcomponent>
```

The `cfcomponent` tag includes optional attributes that you can use to control the WSDL that ColdFusion generates. You can use these attributes to create meaningful WSDL attribute names, as the following example shows:

```
<cfcomponent style="document"
    namespace = "http://www.mycompany.com/"
  serviceportname = "RestrictedEmpInfo"
  porttypename = "RestrictedEmpInfo"
  bindingname = "myns:RestrictedEmpInfo"
  displayname = "RestrictedEmpInfo"
  hint = "RestrictedEmpInfo">
```

Tip: For complete control of the WSDL, advanced users can specify the `cfcomponent wsdlFile` attribute to use a predefined WSDL file.

Securing your web services

You can restrict access to your published web services to control the users allowed to invoke them. You can use your web server to control access to the directories containing your web services, or you can use ColdFusion security in the same way that you would to control access to any ColdFusion page.

Controlling access to component CFC files

To browse the HTML description of a CFC file, you request the file by specifying a URL to the file in your browser. By default, ColdFusion secures access to all URLs that directly reference a CFC file, and prompts you to enter a password upon the request. Use the ColdFusion RDS password to view the file.

To disable security on CFC file browsing, use the ColdFusion MX Administrator to disable the RDS password.

For more information, see Chapter 10, “Building and Using ColdFusion Components,” on page 201.

Using your web server to control access

Most web servers, including IIS and Apache, implement directory access protection using the basic HTTP authentication mechanism. When a client attempts to access one of the resources under a protected directory, and has not properly authenticated, the web server automatically sends back an authentication challenge, typically an HTTP Error 401 Access Denied error.

In response, the client's browser opens a login prompt containing a username and password field. When the user submits this information, the browser sends it back to the web server. If authentication passes, the web server allows access to the directory. The browser also caches the authentication data as long as it is open, so subsequent requests automatically include the authentication data.

Web service clients can also pass the username and password information as part of the request. The `cfinvoke` tag includes the `username` and `password` attributes that let you pass login information to a web server using HTTP basic authentication. You can include these attributes when invoking a web service, as the following example shows:

```
<cfinvoke
  webservice = "http://some.cfc?wsdl"
  returnVariable = "foo"
  ...
  username="aName"
  password="aPassword">
<cfoutput>#foo#</cfoutput>
```

ColdFusion inserts the username/password string in the `authorization` request header as a base64 binary encoded string, with a colon separating the username and password. This method of passing the username/password is compatible with the HTTP basic authentication mechanism used by web servers.

The ColdFusion MX Administrator lets you predefine web services. As part of defining the web service, you can specify the username and password that ColdFusion includes as part of the request to the web service. Therefore, you do not have to encode this information using the `cfinvoke` tag. For information on defining a web service in the ColdFusion MX Administrator, see "Configuring web services in the ColdFusion MX Administrator" on page 896.

Using ColdFusion to control access

Instead of letting the web server control access to your web services, you can handle the username/password string in your Application.cfc or Application.cfm file as part of your own security mechanism. In this case, you use the `cflogin` tag to retrieve the username/password information from the `authorization` header, decode the binary string, and extract the username and password, as the following excerpt from an Application.cfc `onRequestStart` method shows:

```
<cflogin>
  <cfset isAuthorized = false>

  <cfif isDefined("cflogin")
    <!--- verify user name from cflogin.name and password from
  cflogin.password using your authentication mechanism --->
    >
    <cfset isAuthorized = true>
```

```
  </cfif>
</cflogin>

<cfif not isAuthorized>
  <!--- If the user does not pass a username/password, return a 401 error.
    The browser then prompts the user for a username/password. --->
  <cfheader statuscode="401">
  <cfheader name="WWW-Authenticate" value="Basic realm=""Test""">
  <cfabort>
</cfif>
```

This example does not show how to perform user verification. For more information on verification, see Chapter 16, "Securing Applications," on page 373.

Best practices for publishing web services

ColdFusion web services provide a powerful mechanism for publishing and consuming application functionality. However, before you produce web services for publication, you might want to consider the following best practices:

1. Minimize the use of ColdFusion complex types, such as query and struct, in the web services you create for publication. These types require consumers, especially those consuming the web service using a technology other than ColdFusion, to create special data structures to handle complex types.
2. Locally test the ColdFusion components implemented for web services before publishing them over the Internet.

Using request and response headers

ColdFusion MX includes a set of functions that enable your web service to get and set request and response headers. You use these functions to retrieve the response headers from a web service request and to create SOAP headers in a request that has the `mustUnderstand` attribute set to be True.

You typically use different functions in web services clients and in the web service CFC, itself:

In the client:

- `AddSOAPRequestHeader`, called before the request to set a SOAP header.
- `GetSOAPResponseHeader`, called after the request to retrieve a SOAP header.

In the web service CFC:

- `IsSOAPRequest`, called to determine whether the CFC method is being called as a web service.
- `GetSOAPRequestHeader`, called to retrieve a SOAP header set by the client.
- `AddSOAPResponseHeader`, called to set a SOAP header that is returned to the client.

Note: When used in a CFC, you can only use these functions in CFC methods if they are being used as web services. Use the `IsSOAPRequest` function to determine whether the CFC method is being called as a web service.

The following example CFM page uses the `AddSOAPRequestHeader`, `getSOAPRequest`, and `GetSOAPResponse` functions:

```
<cfsavecontent variable="my_xml">
  <Request xmlns="http://www.oasis-open.org/asap/0.9/asap.xsd">
    <SenderKey>ss</SenderKey>
    <ReceiverKey>zz</ReceiverKey>
    <ResponseRequired>Yes</ResponseRequired>
    <RequestID>id</RequestID>
  </Request>
</cfsavecontent>
<cfset xml_obj = xmlparse(my_xml)>

<cfscript>
  ws = CreateObject("webservice", "http://localhost:8500/soapexamples/
  HeaderFuncs.cfc?WSDL");
  addSOAPRequestHeader(ws, "http://www.cfdevguide.com/", "testrequestheader",
  "#xml_obj#");
</cfscript>

<cfscript>
  ret=ws.showSOAPHeaders();
  inxml = getSOAPRequest(ws);
  outxml = getSOAPResponse(ws);
</cfscript>

<cfoutput>
<h2>Return Value</h2>
<!--- This is XML, so use HTMLCodeFormat. --->
The return value was #ret#
<h2>Complete Request XML</h2>
#htmlcodeformat(inxml)#
<h2>Complete Response XML</h2>
#htmlcodeformat(outxml)#
</cfoutput>
```

The following example CFC uses the `IsSOAPRequest` and `AddSOAPResponseHeader` functions:

```
<cfcomponent>
  <cffunction
     name = "showSOAPHeaders"
     returnType = "string"
     output = "no"
     access = "remote"
   hint="After calling this function, use GetSOAPRequest and GetSOAPResponse
  to view headers">
  <cfset var xml_obj = "">
  <cfset var ret = "">

    <cfif IsSOAPRequest()>
      <!--- Define a response header --->
      <cfsavecontent variable="response_xml">
         <ThisResponseHeader xmlns="http://www.cfdevguide.com">
           <CreatedDateTime><cfoutput>#now()#</cfoutput></CreatedDateTime>
           <ExpiresInterval>6000</ExpiresInterval>
         </ThisResponseHeader>
```

```
      </cfsavecontent>
      <cfset xml_obj = xmlparse(response_xml)>
      <!--- Add the response header --->
      <cfscript>
        addSOAPResponseHeader("http://www.cfdevguide.com/",
  "testresponseheader", "#xml_obj#");
       ret = "Invoked as a web service. Use GetSOAPRequest and GetSOAPResponse
  to view headers.";
    </cfscript>
    <cfelse>
      <cfset ret = "Not invoked as a web service">
    </cfif>
    <cfreturn ret>
  </cffunction>
</cfcomponent>
```

Handling complex data types

When dealing with web services, handling complex types falls into the following categories:

- Mapping the data types of a web service to consume to ColdFusion data types
- Understanding how clients will reference your ColdFusion data types when you publish a web service

This section describes both categories.

Consuming web services that use complex data types

The following table shows how WSDL data types are converted to ColdFusion data types:

ColdFusion data type	WSDL data type
numeric	SOAP-ENC:double
boolean	SOAP-ENC:boolean
string	SOAP-ENC:string
array	SOAP-ENC:Array
numeric	SOAP-ENC:float
binary	xsd:base64Binary
date	xsd:dateTime
void (operation returns nothing)	
structure	complex type

This table shows that complex data types map to ColdFusion structures. ColdFusion structures offer a flexible way to represent data. You can create structures that contain single-dimension arrays, multi-dimensional arrays, and other structures.

The ColdFusion mapping of complex types to structures is not automatic. You have to perform some processing on the data in order to access it as a structure. The next sections describe how to pass complex types to web services, and how to handle complex types returned from web services.

Passing input parameters to web services as complex types

A web service can take a complex data type as input. In this situation, you can construct a ColdFusion structure that models the complex data type, then pass the structure to the web service.

For example, the following excerpt from a WSDL file shows the definition of a complex type named Employee:

```
<s:complexType name="Employee">
 <s:sequence>
   <s:element minOccurs="1" maxOccurs="1" name="fname" type="s:string" />
   <s:element minOccurs="1" maxOccurs="1" name="lname" type="s:string" />
   <s:element minOccurs="1" maxOccurs="1" name="active" type="s:boolean" />
   <s:element minOccurs="1" maxOccurs="1" name="age" type="s:int" />
   <s:element minOccurs="1" maxOccurs="1" name="hiredate" type="s:dateTime" />
   <s:element minOccurs="1" maxOccurs="1" name="number" type="s:double" />
 </s:sequence>
</s:complexType>
```

The Employee data type definition includes six elements, the data type of each element, and the name of each element.

Another excerpt from the WSDL file shows a message definition using the Employee data type. This message defines an input parameter, as the following code shows:

```
<message name="updateEmployeeInfoSoapIn">
  <part name="thestruct" type="s0:Employee" />
</message>
```

A third excerpt from the WSDL file shows the definition of an operation, named updateEmployeeInfo, possibly one that updates the employee database with the employee information. This operation takes as input a parameter of type Employee, as the following code shows:

```
<operation name="updateEmployeeInfo">
  <input message="s0:updateEmployeeInfoSoapIn" />
</operation>
```

To call the updateEmployeeInfo operation, create a ColdFusion structure, initialize six fields of the structure that correspond to the six elements of Employee, and then call the operation, as the following code shows:

```
<!--- Create a structure using CFScript, then call the web service. --->
<cfscript>
  stUser = structNew();
  stUser.active = TRUE;
  stUser.fname = "John";
  stUser.lname = "Smith";
  stUser.age = 23;
  stUser.hiredate = createDate(2002,02,22);
  stUser.number = 123.321;

  ws = createObject("webservice", "http://somehost/EmployeeInfo.asmx?wsdl");
  ws.updateEmployeeInfo(stUser);
</cfscript>
```

You can use structures for passing input parameters as complex types in many situations. However, to build a structure to model a complex type, you have to inspect the WSDL file for the web service to determine the layout of the complex type. This can take some practice.

Handling return values as complex types

When a web service returns a complex type, you can write that returned value directly to a ColdFusion variable.

The previous section used a complex data type named Employee to define an input parameter to an operation. A WSDL file can also define a return value using the Employee type, as the following code shows:

```
<message name="updateEmployeeInfoSoapOut">
  <part name="updateEmployeeInfoResult" type="s0:Employee" />
</message>
<operation name="updateEmployeeInfo">
  <input message="s0:updateEmployeeInfoSoapIn" />
  <output message="s0:updateEmployeeInfoSoapOut" />
</operation>
```

In this example, the operation updateEmployeeInfo takes a complex type as input and returns a complex type as output. To handle the input parameter, you create a structure. To handle the returned value, you write it to a ColdFusion variable, as the following example shows:

```
<!--- Create a structure using CFScript, then call the web service. --->
<!--- Write the returned value to a ColdFusion variable. --->
<cfscript>
  stUser = structNew();
  stUser.active = TRUE;
  stUser.fname = "John";
  stUser.lname = "Smith";
  stUser.age = 23;
  stUser.hiredate = createDate(2002,02,22);
  stUser.number = 123.321;

  ws = createObject("webservice", "http://somehost/echosimple.asmx?wsdl");
  myReturnVar = ws.echoStruct(stUser);

</cfscript>

<!--- Output the returned values. --->
<cfoutput>
  <br>
  <br>Name of employee is: #myReturnVar.fname#  #myReturnVar.lname#
  <br>Active status: #myReturnVar.active#
  <br>Age:  #myReturnVar.age#
  <br>Hire Date: #myReturnVar.hiredate#
  <br>Favorite Number: #myReturnVar.number#
</cfoutput>
```

You access elements of the variable myReturnVar using dot notation in the same way that you access structure fields. If a complex type has nested elements, in the way a structure can have multiple levels of nested fields, you use dot notation to access the nested elements, as in a.b.c.d, to whatever nesting level is necessary.

However, the variable myReturnVar is not a ColdFusion structure. It is a container for the complex type, but has none of the attributes of a ColdFusion structure. Calling the ColdFusion function `isStruct` on the variable returns False.

You can copy the contents of the variable to a ColdFusion structure, as the following example shows:

```
<cfscript>
...
  ws = createObject("webservice", "http://somehost/echosimple.asmx?wsdl");
  myReturnVar = ws.echoStruct(stUser);

  realStruct = structNew();
  realStruct.active = #myReturnVar.active#;
  realStruct.fname = "#myReturnVar.fname#";
  realStruct.lname = "#myReturnVar.lname#";
  realStruct.age = #myReturnVar.age#;
  realStruct.hiredate = #myReturnVar.hiredate#;
  realStruct.number = #myReturnVar.number#;

</cfscript>
```

Calling `IsStruct` on realStruct returns True and you can use all ColdFusion structure functions to process it.

This example shows that ColdFusion variables and structures are useful for handling complex types returned from web services. To understand how to access the elements of a complex type written to a ColdFusion variable, you have to inspect the WSDL file for the web service. The WSDL file defines the API to the web service and will provide you with the information necessary to handle data returned from it.

Publishing web services that use complex data types

The two ColdFusion data types that do not map exactly to WSDL data types are struct and query. When you publish a ColdFusion web service that uses parameters of type struct or query, the consuming application needs to be able to handle the data.

Note: If the consumer of a ColdFusion web service is another ColdFusion application, you do not have to perform any special processing. ColdFusion correctly maps struct and query data types in the web service publisher with the consumer. For more information, see "Consuming ColdFusion web services" on page 897.

Publishing structures

A ColdFusion structure can hold an unlimited number of key-value pairs where the values can be of any ColdFusion data type. While it is a very useful and powerful way to represent data, it cannot be directly mapped to any XML data types defined in the SOAP 1.1 encoding and XML Schema specification. Therefore, ColdFusion structures are treated as a custom type and the complex type XML schema in WSDL looks like the following:

```
<complexType name="mapItem">
  <sequence>
    <element name="key" nillable="true" type="xsd:anyType"/>
    <element name="value" nillable="true" type="xsd:anyType"/>
  </sequence>
</complexType>
<complexType name="Map">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="item"
  type="apachesoap:mapItem"/>
  </sequence>
</complexType>
```

This complex type defines a representation of a structure, where the structure keys and values can be any type.

In the WSDL mapping of a ColdFusion structure, each key-value pair in the structure points to the next element in the structure except for the final field, which contains a value. Use dot notation to access the key-value pairs.

Publishing queries

ColdFusion publishes query data types as the WSDL type QueryBean. The QueryBean data type contains two elements, as the following excerpt from a WSDL file shows:

```
<complexType name="QueryBean">
  <all>
    <element name="data" nillable="true" type="intf:ArrayOf_SOAP-ENC_Array" /
  >
    <element name="ColumnList" nillable="true"
  type="intf:ArrayOf_SOAP-ENC_string" />
  </all>
</complexType>
```

The following table describes the elements of QueryBean:

Element name	Description
ColumnList	String array that contains column names
data	Two-dimensional array that contains query data

The WSDL file for a QueryBean defines these elements as follows:

```
<complexType name="ArrayOf_SOAP-ENC_Array">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
```

```
        <attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="SOAP-ENC:Array[]" /
    >
      </restriction>
    </complexContent>
  </complexType>
  <complexType name="ArrayOf_SOAP-ENC_string">
    <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="xsd:string[]" />
    </restriction>
    </complexContent>
  </complexType>
```

Troubleshooting SOAP requests and responses

ColdFusion MX provides the following facilities for troubleshooting SOAP requests and responses:

- The `getSOAPRequest` and `getSOAPResponse` functions.
- The TCP monitor.

Viewing SOAP requests and responses

You can use the `getSOAPRequest` and `getSOAPResponse` functions to retrieve and display the XML passed to and from a web service. Although advanced users may use this information for custom functionality, you typically use these functions for debugging.

Use these functions in the following places:

- `GetSOAPRequest` Clients call this function after the web service request; web service CFCs call this function in the web service CFC method.
- `GetSOAPResponse` Clients call this function after the web service request completes; web service CFCs cannot use this method.

The following example uses the `GetSOAPRequest` and `GetSOAPResponse` functions in a web service client:

```
<cfscript>
 ws = CreateObject("webservice", "http://localhost:8500/soapexamples/
   tester.cfc?WSDL");
 addSOAPRequestHeader(ws, "http://mynamespace/", "username", "randy");
 ret = ws.echo_me("value");
</cfscript>

<cfset soapreq = GetSOAPRequest(ws)>
<h2>SOAP Request</h2>
<cfdump var="#soapreq#">

<cfset soapresp = GetSOAPResponse(ws)>
<h2>SOAP Response</h2>
<cfdump var="#soapresp#">
...
```

The following example uses the `GetSOAPRequest` function in a web service CFC method:

```
<cfcomponent displayName="testerdebug" hint="Test for underscores">

<cffunction access="remote" name="echo_me" output="false" returntype="string"
  displayname="Echo Test" hint="Header test">
  <cfargument name="in_here" required="true" type="string">
  <cfset var soapreq = “”>

  <cfif IsSOAPRequest()>
    <cfset soapreq = GetSOAPRequest()>
  <cflog text="#soapreq#"
          log="APPLICATION"
          type="Information">
...
```

Using the TCP monitor

TCPMonitor is a swing-based application that lets you watch the request and response flow of HTTP traffic. You can also watch the request and response flow of SOAP traffic. TCPMonitor replaces the Sniffer service formerly used in Macromedia JRun.

To run TCPMonitor:

- On Windows and Unix platforms, you can execute the TCPMonitor by launching the sniffer utility in the jrun_root/bin directory.

  The TCP Monitor main window appears.

TCPMonitor is a swing-based application that lets you watch the request and response flow of HTTP traffic. However, you can also use it to watch the request and response flow of SOAP traffic.

To run TCPMonitor:

1. On Windows and Unix platforms, you can execute the TCPMonitor by launching the sniffer utility in the *cf_root*/bin (server configuration) or *jrun_root*/bin (multiserver configuration) directory.

   The TCP Monitor main window appears.

   Note: In the J2EE configuration, run the utility directly out of the JAR file by using the following command:

   ```
   java -cp cf_webapp_root/WEB-INF/cfusion/lib/axis.jar java
     org.apache.axis.utils.tcpmon [listening_port] [target_host] [target_port]
   ```

2. Enter the values in the main window as described in the following table:

Field	Description
Listen Port#	Enter a local port number, such as 8123, to monitor for incoming connections. Instead of requesting the usual port on which your server runs, you request this port. TCPMonitor intercepts the request and forwards it to the Target Port.
Listener	Select Listener to use TCPMonitor as a sniffer service in JRun.

Field	Description
Proxy	Select Proxy to enable proxy support for TCPMonitor.
Target Hostname	Enter the target host to which incoming connections are forwarded. For example, if you are monitoring a service running on a local server, the hostname is localhost.
Target Port#	Enter the port number on the target machine to which TCPMonitor connects. For example, if you are monitoring a service running on your local ColdFusion server in the server configuration, the default port number is 8500.
HTTP Proxy Support	Select this check box only to configure proxy support for TCPMonitor.

You can optionally specify the Listen Port#, Target Hostname and Target Port# values when invoking TCPMonitor on the command line. The following is the syntax for TCPMonitor:

```
java org.apache.axis.utils.tcpmon [listening_port] [target_host]
  [target_port]
```

3. To add this profile to your TCPMonitor session, click Add.

   A tab appears for your new tunneled connection.
4. Select the new tab. If there are port conflicts, TCPMonitor alerts you in the Request panel.
5. Request a page using the Listen Port defined in this TCPMonitor session. For example, if you entered 8123 for the Listen Port, enter the following URL in your browser:

   ```
   http://localhost:8123/
   ```

   TCPMonitor displays the current request and response information:

   For each connection, the request appears in the Request panel and the response appears in the Response panel. TCPMonitor keeps a log of all request-response pairs and lets you view any particular pair by selecting an entry in the top panel.
6. To save results to a file for later viewing, click Save. To clear the top panel of older requests that you do not want to save, click Remove Selected and Remove All.
7. To resend the request that you are currently viewing and view a new response, click Resend. You can edit the request in the Request panel before resending, and test the effects of different requests.
8. To change the ports, click Stop, change the port numbers, and click Start.
9. To add another listener, click the Admin tab and enter the values as described previously.
10. To end this TCPMonitor session, click Close.

CHAPTER 37
Integrating J2EE and Java Elements in CFML Applications

This chapter describes how to integrate J2EE elements, including JSP pages and servlets; JSP tags; and Java objects, including Enterprise JavaBeans (EJBs); into your ColdFusion application.

It does not explain J2EE concepts or how to program using Java or JSP. It does explain how to use existing Java and JSP elements in your ColdFusion applications.

Contents

About ColdFusion, Java, and J2EE 917
Using JSP tags and tag libraries 921
Interoperating with JSP pages and servlets 922
Using Java objects 928

About ColdFusion, Java, and J2EE

Macromedia ColdFusion MX is built on a J2EE-compliant Java technology platform. This lets ColdFusion applications take advantage of, and integrate with, J2EE elements. ColdFusion pages can do any of the following:

- Include JavaScript and client-side Java applets on the page.
- Use JSP tags.
- Interoperate with JSP pages.
- Use Java servlets.
- Use Java objects, including JavaBeans and Enterprise JavaBeans.

About ColdFusion and client-side JavaScript and applets

ColdFusion pages, like HTML pages, can incorporate client-side JavaScript and Java applets. To use JavaScript, you write the JavaScript code just as you do on any HTML page. ColdFusion ignores the JavaScript and sends it to the client.

The `cfapplet` tag simplifies using Java client-side applets.

To use an applet on a ColdFusion page:

1. Register the applet .class file in ColdFusion MX Administrator Java Applets Extensions page. (For information on registering applets, see the ColdFusion MX Administrator online Help.)
2. Use the `cfapplet` tag to call the applet. The `appletSource` attribute must be the Applet name assigned in ColdFusion MX Administrator.

For example, ColdFusion includes a Copytext sample applet that copies text from one text box to another. The ColdFusion Setup automatically registers the applet in the Administrator. To use this applet, incorporate it on your page. For example:

```
<cfform action = "copytext.cfm">
  <cfapplet appletsource = "copytext" name = "copytext">
</cfform>
```

About ColdFusion and JSP

ColdFusion MX supports JSP tags and pages in the following ways:

- Interoperates with JSP pages: ColdFusion pages can include or forward to JSP pages, JSP pages can include or forward to ColdFusion pages, and both types of pages can share data in persistent scopes.
- Imports and uses JSP tag libraries: the `cfimport` tag imports JSP tag libraries and lets you use its tags.

ColdFusion pages are not JSP pages, however, and you cannot use most JSP syntax on ColdFusion pages. In particular you *cannot* use the following features on ColdFusion pages:

Include, Taglib, and Page directives Instead, you use CFML `import` tag to import tag libraries, and the `include` (or `forward`) method of the page context object returned by the ColdFusion `GetPageContext` function to include pages. For more information, see "Using JSP tags and tag libraries" on page 921 and "Interoperating with JSP pages and servlets" on page 922.

Expression, Declaration, and Scriptlet JSP scripting elements Instead, you use CFML elements and expressions.

JSP comments Instead, you use CFML comments. (ColdFusion ignores JSP comments and passes them to the browser.)

Standard JSP tags Such as jsp:plugin, unless your J2EE server provides access to these tags in a JAR file. Instead, you use ColdFusion tags and the PageContext object.

About ColdFusion and servlets

Some Java servlets are not exposed as JSP pages; instead they are Java programs. You can incorporate JSP servlets in your ColdFusion application. For example, your enterprise might have an existing servlet that performs some business logic. To use a servlet, the ColdFusion page specifies the servlet by using the ColdFusion `GetPageContext` function.

When you access the servlet with the `GetPageContext` function, the ColdFusion page shares the Request, Application, and Session scopes with the servlet, so you can use these scopes for shared data.

ColdFusion pages can also access servlets by using the `cfhttp` tag, use the servlet URL in a `form` tag, or access an SHTML page that uses a `servlet` tag.

Note: The `cfservlet` tag, which provides access to servlets on JRun servers, is deprecated for ColdFusion MX.

About ColdFusion and Java objects

Java objects include the following:

- Standard Java classes and methods that make up the J2EE API
- Custom-written Java objects, including the following:
  - Custom classes, including JavaBeans
  - Enterprise JavaBeans

ColdFusion pages use the `cfobject` tag to access Java objects.

ColdFusion MX searches for the objects in the following order:

1. The ColdFusion Java Dynamic Class Load directories:
   - Java archive (.jar) files in *web_root*/WEB-INF/lib
   - Class (.class) files in *web_root*/WEB-INF/classes

   ColdFusion reloads classes from these directories, as described in the next section, "About class loading".
2. The classpath specified on the JVM and Java Settings page in the ColdFusion MX Administrator.
3. The default JVM classpath.

About class loading

ColdFusion dynamically loads classes that are either .class files in the *web_root*/WEB-INF/classes directory or in JAR files in the *web_root*/WEB-INF/lib directory. ColdFusion checks the time stamp on the file when it creates an object that is defined in either directory, even when the class is already in memory. If the file that contains the class is newer than the class in memory, ColdFusion loads the class from that directory.

To use this feature, make sure that the Java implementation classes that you modify are not in the general JVM classpath.

To disable automatic class loading of your classes, put the classes in the JVM classpath. Classes located on the JVM classpath are loaded once per server lifetime. To reload these classes, stop and restart ColdFusion MX.

Note: Because you put tag libraries in the *web_root*/WEB-INF/lib directory, ColdFusion automatically reloads these libraries if necessary when you import the library.

About GetPageContext and the PageContext object

Because ColdFusion pages are J2EE servlet pages, all ColdFusion pages have an underlying Java PageContext object. CFML includes the `GetPageContext` function that you can then use in your ColdFusion page.

The PageContext object exposes a number of fields and methods that can be useful in J2EE integration. In particular, it includes the `include` and `forward` methods that provide the equivalent of the corresponding standard JSP tags.

This chapter describes how to use the `include` and `forward` PageContext methods for calling JSP pages and servlets. It does not discuss the PageContext object in general. For more information on the object, see Java documentation. You can find the Javadoc description of this class at http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/jsp/PageContext.html.

About CFML variables and Java variables

Because ColdFusion variables are case-independent and Java variables are case-dependent, you must be careful about variable names. Use the following rules and guidelines when sharing data between ColdFusion and Java code, including JSP pages and servlets.

Rules

- If you use mixed case variables, all variable names must be unique, independent of case. For example, you must not have two Java variables, MyVariable and MYVARIABLE. ColdFusion cannot distinguish between the two.
- If you share Request scope variables between a CFML page and a JSP page or servlet, all shared Request scope variable names *must* be all-lowercase in the JSP page or servlet. Mixed case or all-uppercase variables will cause null pointer exceptions if CFML refers to these variables.
- If you share Application or Session scope variables between a CFML page and a JSP page or servlet and use a named ColdFusion application (the common usage), the variables on the JSP page or servlet are case-independent.
- If you share the Application or Session scope variables between a CFML page and a JSP page or servlet, and use an *unnamed* ColdFusion application, the variable names in the JSP page or servlet *must* be all lowercase.
- When you specify a class name in the `cfobject` tag or `CreateObject` function, the name must be case-correct.

Guidelines

- You can prevent problems by consistently using all-lowercase variable names.
- In your CFML, use the same case as you do in your Java or JSP. Doing so does not change how the application works, but does help prevent confusion.

Using JSP tags and tag libraries

You can use JSP tags from any JSP tag library. For example, you can use any of the custom tags in the open-source Apache Jakarta Project Taglibs project tag libraries, located at http://jakarta.apache.org/taglibs/index.html. This project consists of a number of individual JSP custom tag libraries for purposes ranging from JNDI access to generating random text strings.

Using a JSP tag in a ColdFusion page

JSP pages use a standard set of tags, such as `jsp:forward` and `jsp:include`. You can also import custom JSP tag libraries into a JSP application. You can use both the standard JSP tags and custom JSP tags in ColdFusion pages, as the following sections describe.

Standard JSP tags and ColdFusion

ColdFusion tags provide equivalent features to most standard JSP tags. For example, the `cfapplet` tag provides the same service as the `jsp:plugin` tag, and `cfobject` tag lets you use JavaBeans, as does the `jsp:usebean` tag. Similarly, you do not use the `jsp:getproperty` tag because ColdFusion automatically gets properties when you reference them. Therefore, ColdFusion does not support the use of standard JSP tags directly.

However, two standard JSP tags provide functionality that is useful in ColdFusion pages: the `forward` and `include` tags invoke JSP pages and Java servlets. The PageContext object described in "About GetPageContext and the PageContext object" on page 920 has `forward` and `include` methods that provide the same operations. For more information about using these methods see "Accessing a JSP page or servlet from a ColdFusion page" on page 922.

Using custom JSP tags in a ColdFusion page

Follow these steps to use a custom JSP tag on a ColdFusion page:

To use a custom tag:

1. Put the tag library, consisting of the *taglibname*.jar file, and the *taglibname*.tld file, if one is supplied, in the *web_root*/WEB-INF/lib directory. The JSP custom tag library must be in this directory for you to use the `cfimport` tag.
2. Restart ColdFusion.
3. In the ColdFusion page that uses a JSP tag from the tag library, specify the tag library name in a `cfimport` tag; for example:

   ```
   <cfimport taglib="/WEB-INF/lib/random.jar" prefix="random">
   ```

   If the TLD file is not included in the JAR file, use the .tld suffix in place of the .jar suffix.

   Note: The `cfimport` tag must be on the page that uses the imported tag. You cannot put the `cfimport` tag in Application.cfm.

4. Use the custom tag using the form *prefix*:*tagName*; for example:

   ```
   <random:number id="myNum" range="000000-999999" />
   ```

Note: You cannot use the `cfsavecontent` tag to suppress output of a custom JSP tag.

Example: using the random tag library

The following example uses the random tag library from the Apache Jakarta Taglibs project and calls the library's `number` tag, which initializes a random number generator that uses a secure algorithm to generate a six-digit random number. You get a new random number each time you reference the variable randPass.random.

```
<cfimport taglib="/WEB-INF/lib/taglibs-random.jar" prefix="myrand">
<myrand:number id="randPass" range="000000-999999" algorithm="SHA1PRNG"
  provider="SUN" />
<cfset myPassword = randPass.random>
<cfoutput>
  Your password is #myPassword#<br>
</cfoutput>
```

For more information on the Jakarta random tag library and how to use its tags, see the documentation at the Apache Jakarta Taglibs project website, http://jakarta.apache.org/taglibs/index.html. The Taglibs project includes many open source custom tag libraries.

Interoperating with JSP pages and servlets

ColdFusion pages and JSP pages can interoperate in several ways:

- ColdFusion pages can invoke JSP pages and servlets.
- JSP pages can invoke ColdFusion pages.
- ColdFusion pages, JSP pages, and servlets can share data in three scopes.

The following sections show how you can use these techniques.

Integrating JSP and servlets in a ColdFusion application

You can integrate JSP pages and servlets in your ColdFusion application. For example, you can write some application pages in JSP and write others in CFML. ColdFusion pages can access JSP pages by using the JSP `include` and `forward` methods to call the page. As with any web application, you can use `href` links in ColdFusion pages to open JSP pages.

The ability to use JSP lets you incorporate legacy JSP pages in your ColdFusion application, or conversely, use CFML to expand an existing JSP application using ColdFusion pages.

If you have a JSP page that must call a ColdFusion page, you also use a `jsp:forward` or `jsp:include` tag to call the ColdFusion page. For an example of calling a ColdFusion page from a JSP page, see "Calling a JSP page from a ColdFusion page" on page 925.

Accessing a JSP page or servlet from a ColdFusion page

To access a JSP page or servlet from a ColdFusion page, you use the `GetPageContext` function with the `forward` or the `include` method. For example, to include a JSP "Hello World" page in your ColdFusion application, use the following line:

```
GetPageContext().include("hello.jsp");
```

To pass parameters to the JSP page, include the parameters in the page URL.

For example, you might want to integrate an existing JSP customer response component into a new ColdFusion order processing application. The order processing application provides the order number, total cost, and expected shipping date, and the customer response component sends the response to the e-mail address on file for the particular customer number. The ColdFusion application might use the following CFScript code to call the response JSP page:

```
urlParams = "UID=#order.uid#&cost=#order.total#&orderNo=#order.orderNo#
  &shipDate=#order.shipDateNo#"
getPageContext().forward(URLEncodedFormat("/responsegen/responsegen.jsp
  ?#urlParams#"));
```

To access a servlet that exposes the same functionality, you use the same code, although the URL would change. For example, to run a servlet called HelloWorldServlet, you put the servlet .java or .class file in the *serverroot*/WEB-INF/classes directory and refer to the servlet with the URL /servlet/HelloWorldServlet.

Sharing data between ColdFusion pages and JSP pages or servlets

If an application includes ColdFusion pages and JSP pages or servlets, they can share data in the Request, Session and Application scopes. The following table lists the ways that you can access JSP pages with which you want to share the scope data:

Scope	Can share data using
Request	forward, include **Note:** Shared Request scope variable names in the JSP page or servlet must be all-lowercase.
Session	href, cfhttp, forward, include
Application	href, cfhttp, forward, include

Note: When you share data between ColdFusion pages and JSP pages, you must be careful about data type conversion issues. For more information, see "Java and ColdFusion data type conversions" on page 933.

To share session variables, you must specify J2EE session management in the ColdFusion MX Administrator. For more information on configuring and using J2EE Session scope management, see Chapter 15, "ColdFusion and J2EE session management," on page 352.

For example, you could put the customer order structure used in the previous example in the Session scope. Then, you would not have to pass the order values as a set of parameters. Instead, the JSP pages could access the Session scope variables directly, and the ColdFusion page would only require a line like the following to call the JSP page:

```
getPageContext().forward(URLEncodedFormat("/responsegen/responsegen.jsp"));
```

For examples of using the Request, Session, and Application scopes to share data between ColdFusion pages and JSP pages, including samples of the appropriate JSP code, see the following section, "Examples: using JSP with CFML".

Note: When running in the server configuration, ColdFusion MX also shares the Form scope when calling a JSP or servlet. In the J2EE configuration, however, sharing the Form scope is dependant on the J2EE application server. For example, Macromedia JRun shares the Form scope, IBM WebSphere does not. ColdFusion always shares the Request, Session, and Application scopes.

Accessing ColdFusion application and session variables in JSP pages

ColdFusion MX runs as a J2EE application on the J2EE application server. The J2EE application ServletContext is a data structure that stores objects as attributes. A ColdFusion Application scope is represented as an attribute named by the Application scope name. The attribute contains the scope values as a hash table. Therefore, you access ColdFusion Application scope variable in a JSP page or servlet using the following format:

```
((Map)application.getAttribute("CFApplicationName"))).get("appVarName")
```

Similarly, the ColdFusion Session scope is a structure within the J2EE session. Because ColdFusion MX identifies sessions by the application name. the session structure is contained in an attribute of the J2EE session that is identified by the application name. Therefore, you access ColdFusion session variables as follows:

```
((Map)(session.getAttribute("CFApplicationName"))).get("sessionVarName")
```

Unnamed ColdFusion Application and Session scopes

If you do not specify an application name in the This.name variable in the Application.cfc initialization code or by using the ColdFusion `cfapplication` tag, the application is unnamed, and the Application scope corresponds to the ColdFusion MX J2EE servlet context. ColdFusion MX, therefore, supports only a single unnamed application. If multiple `cfapplication` tags and Application.cfc files do not specify an application name, all pages in these applications share the servlet context as their Application scope.

All sessions of unnamed applications correspond directly to the J2EE application server's session object. (If you do not use J2EE session variables, ColdFusion MX ensures that the J2EE session lasts at least as long as the session time-out.)

You access an Application scope variable from a ColdFusion unnamed application in a JSP page using the following format:

```
application.getAttribute("applicationVariableName")
```

You access Session scope variables in a ColdFusion unnamed application as follows:

```
session.getAttribute("sessionVariableName")
```

Note: When you use application and session variables for the unnamed ColdFusion application in JSP pages and servlets, the variable names must be case-correct. That is, the characters in the variable name must have the same case as you used when you created the variable in ColdFusion. You do not have to use case-correct application and session variable names for named ColdFusion applications.

Examples: using JSP with CFML

The following simple examples show how you can integrate JSP pages, servlets, and ColdFusion pages. They also show how you can use the Request, Application, and Session scopes to share data between ColdFusion pages, JSP pages, and servlets.

Calling a JSP page from a ColdFusion page

The following page sets Request, Session, and application variables and calls a JSP page, passing it a name parameter:

```
<cfapplication name="myApp" sessionmanagement="yes">
<cfscript>
Request.myVariable = "This";
Session.myVariable = "is a";
Application.myVariable = "test.";
GetPageContext().include("hello.jsp?name=Bobby");
</cfscript>
```

Reviewing the code

The following table describes the CFML code and its function:

Code	Description
`<cfapplication name="myApp" sessionmanagement="yes">`	Specifies the application name as myApp and enables session management. In most applications, this tag is in the Application.cfm page.

`<cfscript>`
`Request.myVariable = "This";`
`Session.myVariable = "is a";`
`Application.myVariable = "test.";`
	Sets ColdFusion Request, Session, and Application, scope variables. Uses the same name, myVariable, for each variable.

`GetPageContext().include ("hello.jsp?name=Bobby");` `</cfscript>`	Uses the `GetPageContext` function to get the current servlet page context for the ColdFusion page. Uses the `include` method of the page context object to call the hello.jsp page. Passes the name parameter in the URL.

The hello.jsp page is called by the ColdFusion page. It displays the `name` parameter in a header and the three variables in the remainder of the body.

```
<%@page import="java.util.*" %>
<h2>Hello <%= request.getParameter("name")%>!</h2>

<br>Request.myVariable: <%= request.getAttribute("myVariable")%>
<br>session.myVariable: <%=
  ((Map)(session.getAttribute("myApp"))).get("myVariable")%>
<br>Application.myVariable: <%=
  ((Map)(application.getAttribute("myApp"))).get("myVariable")%>
```

Reviewing the code

The following table describes the JSP code and its function (line breaks added for clarity):

Code	Description
`<%@page import="java.util.*" %>`	Imports the java.util package. This contains methods required in the JSP page.
`<h2>Hello <%= request.getParameter ("name")%>!</h2>`	Displays the name passed as a URL parameter from the ColdFusion page. The parameter name is case-sensitive, ***Note***: The `getParameter` request method cannot get all ColdFusion page request parameter values on some application servers. For example, on IBM WebSphere, you cannot use `getParameter` to get form fields.
` request.myVariable: <%= request. getAttribute("myvariable")%>`	Uses the `getAttribute` method of the JSP request object to displays the value of the Request scope variable myVariable. The JSP page must use all lowercase characters to refer to all request scope variables that it shares with CFML pages. You can use any case on the CFML page, but if you use mixed case to all uppercase on the JSP page, the variable will not get its value ColdFusion page.
` session.myVariable: <%= ((Map)(session.getAttribute("myApp "))). get("myVariable")%>`	Uses the `getAttribute` method of the JSP session object to get the myApp object (the Application scope). Casts this to a Java Map object and uses the get method to obtain the myVariable value for display. CFML pages and JSP pages share Session variables independent of the variable name case. The variable on the JSP page can have any case mixture and still receive the value from the ColdFusion page. For example, instead of myVariable, you could use MYVARIABLE or myvariable on this line.
` Application.myVariable: <%= ((Map)(application.getAttribute("m yApp"))) .get("myVariable")%>`	Uses the `getAttribute` method of the JSP myApp application object to obtain the value of myVariable in the Application scope. CFML pages and JSP pages share Application variables independent of the variable name case. The variable on the JSP page can have any case mixture and still receive the value from the ColdFusion page. For example, instead of myVariable, you could use MYVARIABLE or myvariable on this line.

Calling a ColdFusion page from a JSP page

The following JSP page sets Request, Session, and application variables and calls a ColdFusion page, passing it a name parameter:

```
<%@page import="java.util.*" %>

<% request.setAttribute("myvariable", "This");%>
<% ((Map)session.getAttribute("myApp")).put("myVariable", "is a");%>
<% ((Map)application.getAttribute("myApp")).put("myVariable", "test.");%>
```

```
<jsp:include page="hello.cfm">
  <jsp:param name="name" value="Robert" />
</jsp:include>
```

Reviewing the code

The following table describes the JSP code and its function:

Code	Description
`<%@page import="java.util.*" %>`	Imports the java.util package. This contains methods required in the JSP page.
`<% request.setAttribute("myvariable", "This");%>`	Uses the `setAttribute` method of the JSP request object to set the value of the Request scope variable myVariable. The JSP page must use all lowercase characters to refer to all request scope variables that it shares with CFML pages. You can use any case on the CFML page, but if you use mixed case to all uppercase on the JSP page, the JSP page will not share it with the ColdFusion page.
`<% ((Map)session.getAttribute("myApp")) .put("myVariable", "is a");%>`	Uses the `getAttribute` method of the JSP session object to get the myApp object (the Application scope). Casts this to a Java Map object and uses the `set` method to set the myVariable value. CFML pages and JSP pages share Session variables independent of the variable name case. The variable on the JSP page can have any case mixture and still share the value with the ColdFusion page. For example, instead of myVariable, you could use MYVARIABLE or myvariable on this line.
`<% ((Map)application.getAttribute ("myApp")).put("myVariable", "test.");%>`	Uses the `getAttribute` method of the JSP application object to get myApp object (the Application scope) and casts it to a Map object. It then sets the value of myVariable in the myApp application scope object. CFML pages and JSP pages share Application variables independent of the variable name case. The variable on the JSP page can have any case mixture and still share the value with the ColdFusion page. For example, instead of myVariable, you could use MYVARIABLE or myvariable on this line.
`<jsp:include page="hello.cfm"> <jsp:param name="name" value="Robert" /> </jsp:include>`	Sets the name parameter to Robert and calls the ColdFusion page hello.cfm.

The following hello.cfm page is called by the JSP page. It displays the Name parameter in a heading and the three variables in the remainder of the body.

```
<cfapplication name="myApp" sessionmanagement="yes">
<cfoutput>
<h2>Hello #URL.name#!</h2>
```

```
Request.myVariable: #Request.myVariable#<br>
Session.myVariable: #Session.myVariable#<br>
Application.myVariable: #Application.myVariable#<br>

</cfoutput>
```

Reviewing the code

The following table describes the CFML code and its function:

Code	Description
`<cfapplication name="myApp" sessionmanagement="yes">`	Specifies the application name as myApp and enables session management. In most applications, this tag is in the Application.cfm page.
`<cfoutput>` `<h2>Hello #URL.name#!</h2>`	Displays the name passed using the `jsp:param` tag on the JSP page. The parameter name is *not* case-sensitive.
`Request.myVariable: #Request.myVariable# ` `Session.myVariable: #Session.myVariable# ` `Application.myVariable: #Application.myVariable# ` `</cfoutput>`	Displays the Request.myVariable, Session. myVariable, and Application.myVariable values. Note that all variable names on CFML pages are case independent.

Using Java objects

You use the `cfobject` tag to create an instance of a Java object. You use other ColdFusion tags, such as `cfset` and `cfoutput`, or CFScript to invoke properties (attributes), and methods (operations) on the object.

Method arguments and return values can be any valid Java type; for example, simple arrays and objects. ColdFusion does the appropriate conversions when strings are passed as arguments, but not when they are received as return values. For more information on type conversion issues, see "Java and ColdFusion data type conversions" on page 933.

The examples in the following sections assume that the `name` attribute in the `cfobject` tag specified the value obj, and that the object has a property called Property, and methods called Method1, Method2, and Method3.

Note: The `cfdump` tag displays an object's public methods and data.

Using basic object techniques

The following sections describe how to invoke Java objects.

Invoking objects

The `cfobject` tag makes Java objects available in ColdFusion MX. It can access any Java class that is available on the JVM classpath or in either of the following locations:

- In a Java archive (.jar) file in *web_root*/WEB-INF/lib
- In a class (.class) file in *web_root*/WEB-INF/classes

For example:

```
<cfobject type="Java" class="MyClass" name="myObj">
```

Although the `cfobject` tag loads the class, it does **not** create an instance object. Only static methods and fields are accessible immediately after the call to `cfobject`.

If you call a public non-static method on the object without first calling the `init` method, there ColdFusion makes an implicit call to the default constructor.

To call an object constructor explicitly, use the special ColdFusion `init` method with the appropriate arguments after you use the `cfobject tag`; for example:

```
<cfobject type="Java" class="MyClass" name="myObj">
<cfset ret=myObj.init(arg1, arg2)>
```

Note: The `init` method is *not* a method of the object, but a ColdFusion identifier that calls the `new` function on the class constructor. So, if a Java object has an `init` method, a name conflict exists and you cannot call the object's `init` method.

To have persistent access to an object, you must use the `init` function, because it returns a reference to an instance of the object, and `cfobject` does not.

An object created using `cfobject` or returned by other objects is implicitly released at the end of the ColdFusion page execution.

Using properties

Use the following coding syntax to access properties if the object does either of the following actions:

- Exposes the properties as public properties.
- Does not make the properties public, but is a JavaBean that provides public getter and setter methods of the form get*PropertyName*() and set*PropertyName*(*value*). For more information, see the following "Calling JavaBean get and set methods" section.

To set a property:

```
<cfset obj.property = "somevalue">
```

To get a property:

```
<cfset value = obj.property>
```

Note: ColdFusion does not require that property and method names be consistently capitalized. However, you should use the same case in ColdFusion as you do in Java to ensure consistency.

Calling methods

Object methods usually take zero or more arguments. Some methods return values, while others might not. Use the following techniques to call methods:

- If the method has no arguments, follow the method name with empty parentheses, as in the following `cfset` tag:

  ```
  <cfset retVal = obj.Method1()>
  ```

- If the method has one or more arguments, put the arguments in parentheses, separated by commas, as in the following example, which has one integer argument and one string argument:

```
<cfset x = 23>
<cfset retVal = obj.Method1(x, "a string literal")>
```

Note: When you invoke a Java method, the type of the data being used is important. For more information see "Java and ColdFusion data type conversions" on page 933.

Calling JavaBean get and set methods

ColdFusion can automatically invoke get*PropertyName*() and set*PropertyName*(*value*) methods if a Java class conforms to the JavaBeans pattern. As a result, you can set or get the property by referencing it directly, without having to explicitly invoke a method.

For example, if the myFishTank class is a JavaBean, the following code returns the results of calling the getTotalFish() method on the myFish object:

```
<cfoutput>
  There are currently #myFish.TotalFish# fish in the tank.
</cfoutput>
```

The following example adds one guppy to a myFish object by implicitly calling the setGuppyCount(int *number*) method:

```
<cfset myFish.GuppyCount = myFish.GuppyCount + 1>
```

Note: You can use the direct reference method to get or set values in some classes that have *getProperty* and *setProperty* methods but do not conform fully to the JavaBean pattern. However, you cannot use this technique for all classes that have *getProperty* and *setProperty* methods. For example, you cannot directly reference any of the following standard Java classes, or classes derived from them: Date, Boolean, Short, Integer, Long, Float, Double, Char, Byte, String, List, Array.

Calling nested objects

ColdFusion supports nested (scoped) object calls. For example, if an object method returns another object and you must invoke a property or method on that object, you can use the following syntax:

```
<cfset prop = myObj.X.Property>.
```

Similarly, you can use code such as the following CFScript line:

```
GetPageContext().include("hello.jsp?name=Bobby");
```

In this code, the ColdFusion `GetPageContext` function returns a Java PageContext object, and the line invokes the PageContext object's `include` method.

Creating and using a simple Java class

Java is a strongly typed language, unlike ColdFusion, which does not enforce data types. As a result, there are some subtle considerations when calling Java methods. The following sections create and use a Java class to show how to use Java effectively in ColdFusion pages.

The Employee class

The Employee class has four data members: FirstName and LastName are public, and Salary and JobGrade are private. The Employee class has three overloaded constructors and a overloaded SetJobGrade method.

Save the following Java source code in the file Employee.java, compile it, and place the resulting Employee.class file in a directory that is specified in the classpath:

```
public class Employee {

public String FirstName;
public String LastName;
private float Salary;
private int JobGrade;

public Employee() {
  FirstName ="";
  LastName ="";
  Salary = 0.0f;
  JobGrade = 0;
}

public Employee(String First, String Last) {
  FirstName = First;
  LastName = Last;
  Salary = 0.0f;
  JobGrade = 0;
}

public Employee(String First, String Last, float salary, int grade) {
  FirstName = First;
  LastName = Last;
  Salary = salary;
  JobGrade = grade;
}

public void SetSalary(float Dollars) {
  Salary = Dollars;
}

public float GetSalary() {
  return Salary;
}

public void SetJobGrade(int grade) {
  JobGrade = grade;
}

public void SetJobGrade(String Grade) {
  if (Grade.equals("CEO")) {
    JobGrade = 3;
  }
  else if (Grade.equals("MANAGER")) {
    JobGrade = 2;
```

```
  }
  else if (Grade.equals("DEVELOPER")) {
    JobGrade = 1;
  }
}

public int GetJobGrade() {
  return JobGrade;
}

}
```

A CFML page that uses the Employee class

Save the following text as JEmployee.cfm:

```
<html>
<body>
<cfobject action="create" type="java" class="Employee" name="emp">
<!--- <cfset emp.init()> --->
<cfset emp.firstname="john">
<cfset emp.lastname="doe">
<cfset firstname=emp.firstname>
<cfset lastname=emp.lastname>
</body>

<cfoutput>
  Employee name is #firstname# #lastname#
</cfoutput>
</html>
```

When you view the page in your browser, you get the following output:

Employee name is john doe

Reviewing the code

The following table describes the CFML code and its function:

Code	Description
`<cfobject action=create type=java class=Employee name=emp>`	Loads the Employee Java class and gives it an object name of emp.
`<!--- <cfset emp.init()> --->`	Does not call a constructor. ColdFusion invokes the default constructor when it first uses the class; in this case, when it processes the next line.
`<cfset emp.firstname="john">` `<cfset emp.lastname="doe">`	Sets the public fields in the emp object to your values.
`<cfset firstname=emp.firstname>` `<cfset lastname=emp.lastname>`	Gets the field values back from emp object.

`<cfoutput>`
`  Employee name is #firstname#`
`  #lastname#`
`</cfoutput>`
	Displays the retrieved values.

Java considerations

Keep the following points in mind when you write a ColdFusion page that uses a Java class object:

- The Java class name is case-sensitive. You must ensure that the Java code and the CFML code use Employee as the class name.
- Although Java method and field names are case-sensitive, ColdFusion variables are not case-sensitive, and ColdFusion does any necessary case conversions. As a result, the sample code works even though the CFML uses emp.firstname and emp.lastname; the Java source code uses FirstName and LastName for these fields.
- If you do not call the constructor (or, as in this example, comment it out), ColdFusion automatically invokes the default constructor when it first uses the class.

Using an alternate constructor

The following ColdFusion page explicitly calls one of the alternate constructors for the Employee object:

```
<html>
<body>

<cfobject action="create" type="java" class="Employee" name="emp">
<cfset emp.init("John", "Doe", 100000.00, 10)>
<cfset firstname=emp.firstname>
<cfset lastname=emp.lastname>
<cfset salary=emp.GetSalary()>
<cfset grade=emp.GetJobGrade()>

<cfoutput>
  Employee name is #firstname# #lastname#<br>
  Employee salary #DollarFormat(Salary)#<br>
  Employee Job Grade #grade#
</cfoutput>

</body>
</html>
```

In this example, the constructor takes four arguments: the first two are strings, the third is a float, and the fourth is an integer.

Java and ColdFusion data type conversions

ColdFusion does not use explicit types for variables, while Java is strongly typed. However, ColdFusion data does use a number of underlying types to represent data.

Under most situations, when the method names are not ambiguous, ColdFusion can determine the data types that are required by a Java object, and often it can convert ColdFusion data to the required types. For example, ColdFusion text strings are implicitly converted to the Java String type. Similarly, if a Java object contains a doIt method that expects a parameter of type int, and CFML is issuing a doIt call with a CFML variable x that contains an integer value, ColdFusion converts the variable x to Java int type. However, ambiguous situations can result from Java method overloading, where a class has multiple implementations of the same method that differ only in their parameter types.

The following sections describe how ColdFusion handles the unambiguous situations, and how it provides you with the tools to handle ambiguous ones.

Default data type conversion

Whenever possible, ColdFusion automatically matches Java types to ColdFusion types.

The following table lists how ColdFusion converts ColdFusion data values to Java data types when passing arguments. The left column represents the underlying ColdFusion representation of its data. The right column indicates the Java data types into which ColdFusion can automatically convert the data:

CFML	Java
Integer	short, int, long (short and int might result in a loss of precision).
Real number	float double (float might result in a loss of precision.
Boolean	boolean
Date-time	java.util.Date
String, including lists	String short, int, long, float, double, java.util.Date, when a CFML string represents a number or date. boolean, for strings with the value Yes, No, True, and False (case-insensitive).
Array	java.util.Vector (ColdFusion Arrays are internally represented using an instance of a java.util.Vector object.) ColdFusion can also map a CFML array to any of the following when the CFML array contains consistent data of a type that can be converted to the Java array's data type: byte[], char[], boolean[], int[], long[], float[], double[], String[], or Object[]. When a CFML array contains data of different of types, the conversion to a simple array type might fail.
Structure	java.util.Map
Query object	java.util.Map
XML document object	Not supported.
ColdFusion component	Not applicable.

The following table lists how ColdFusion converts data returned by Java methods to ColdFusion data types:

Java	CFML
boolean/Boolean	Boolean
byte/Byte	String
char/Char	String
short/Short	Integer
int/Integer	Integer
long/Long	Integer
float/Float	Real Number
double/Double	Real Number
String	String
java.util.Date	Date-time
java.util.List	Comma-delimited list
byte[]	Array
char[]	Array
boolean[]	Array
String[]	Array
java.util.Vector	Array
java.util.Map	Structure

Resolving ambiguous data types with the JavaCast function

You can overload Java methods so a class can have several identically named methods. At runtime, the JVM resolves the specific method to use based on the parameters passed in the call and their types.

In the section "The Employee class" on page 931, the Employee class has two implementations for the SetJobGrade method. One method takes a string variable, the other an integer. If you write code such as the following, which implementation to use is ambiguous:

```
<cfset emp.SetJobGrade("1")>
```

The "1" could be interpreted as a string or as a number, so there is no way to know which method implementation to use. When ColdFusion encounters such an ambiguity, it throws a user exception.

The ColdFusion `JavaCast` function helps you resolve such issues by specifying the Java type of a variable, as in the following line:

```
<cfset emp.SetJobGrade(JavaCast("int", "1"))>
```

The JavaCast function takes two parameters: a string representing the Java data type, and the variable whose type you are setting. You can specify the following Java data types: boolean, int, long, float, double, and String.

For more information about the JavaCast function, see *CFML Reference*.

Handling Java exceptions

You handle Java exceptions just as you handle standard ColdFusion exceptions, with the cftry and cfcatch tags. You specify the name of the exception class in the cfcatch tag that handles the exception. For example, if a Java object throws an exception named myException, you specify myException in the cfcatch tag.

Note: To catch any exception generated by a Java object, specify java.lang.Exception for the cfcatch type attribute. To catch any Throwable errors, specify java.lang.Throwable in the cfcatch tag type attribute.

The following sections show an example of throwing and handling a Java exception.

For more information on exception handling in ColdFusion MX, see Chapter 14, "Handling Errors," on page 307.

Example: exception-throwing class

The following Java code defines the testException class that throws a sample exception. It also defines a myException class that extends the Java built-in Exception class and includes a method for getting an error message.

The myException class has the following code. It throws an exception with a message that is passed to it, or if no argument is passed, it throws a canned exception.

```
//class myException
public class myException extends Exception
{
  public myException(String msg) {
    super(msg);
  }
  public myException() {
    super("Error Message from myException");
  }
}
```

The testException class contains one method, doException, which throws a myException error with an error message, as follows:

```
public class testException {
  public testException ()
  {
  }
  public void doException() throws myException {
    throw new myException("Throwing an exception from testException class");
  }
}
```

Example: CFML Java exception handling code

The following CFML code calls the testException class doException method. The `cfcatch` block handles the resulting exception.

```
<cfobject action=create type=java class=testException name=Obj>
<cftry>
  <cfset Obj.doException() >
  <cfcatch type="myException">
    <cfoutput>
      <br>The exception message is: #cfcatch.Message#<br>
    </cfoutput>
  </cfcatch>
</cftry>
```

Examples: using Java with CFML

The following sections show several examples of using Java objects in CFML. They include examples of using a custom Java class, a standard Java API class in a user-defined function, a JavaBean, and an Enterprise JavaBean (EJB).

Using a Java API in a UDF

The following example of a user-defined function (UDF) is functionally identical to the `GetHostAddress` function from the NetLib library of UDFs from the Common Function Library Project, www.cflib.org. It uses the `InetAddress` class from the standard Java 2 java.net package to get the Internet address of a specified host:

```
<cfscript>
function GetHostAddress(host) {
  // Define the function local variables.
  var iaddrClass="";
  var address="";
  // Initialize the Java class.
  iaddrClass=CreateObject("java", "java.net.InetAddress");
  // Get the address object.
  address=iaddrClass.getByName(host);
  // Return the address
  return address.getHostAddress();
}
</cfscript>
<cfoutput>#gethostaddress("macromedia.com")#</cfoutput>
```

Using an EJB

ColdFusion MX can use EJBs that are served by JRun 4.0 servers. The JRun server jrun.jar file must have the same version as the jrun.jar file in ColdFusion.

To call an EJB, you use `cfobject type="Java"` to create and call the appropriate objects. Before you can use an EJB you must do the following:

1. Have a properly deployed EJB running on a J2EE server. The bean must be registered with the JNDI server.
2. Have the following information:
   - Name of the EJB server
   - Port number of the JNDI naming service on the EJB server
   - Name of the EJB, as registered with the naming service
3. Install the EJB home and component interface compiled classes on your ColdFusion web server, either as class files in the *web_root*/WEB-INF/classes directory or packaged in a JAR file the *web_root*/WEB-INF/lib directory.

Note: To use an EJB served by a JRUN server, your ColdFusion installation and the JRun server that hosts the EJB must have the same version of the jrun.jar file (located in *cf_root**runtime**lib* directory in ColdFusion).

Although the specific steps for using an EJB depend on the EJB server and on the EJB itself, they generally correspond to the following order.

To use an EJB:

1. Use the `cfobject` tag to create an object of the JNDI naming context class (javax.naming.Context). You will use fields from this class to define the information that you use to locate the EJB. Because you only use fields, you do not initialize the object.
2. Use the `cfobject` tag to create a java.util.Properties class object that will contain the context object properties.
3. Call the `init` method to initialize the Properties object.
4. Set the Properties object to contain the properties that are required to create an initial JNDI naming context. These include the INITIAL_CONTEXT_FACTORY and PROVIDER_URL properties. You might also need to provide SECURITY_PRINCIPAL and SECURITY_CREDENTIALS values required for secure access to the naming context. For more information on these properties, see the JNDI documentation.
5. Use the `cfobject` tag to create the JNDI InitialContext (javax.naming. InitialContext) object.
6. Call the `init` method for the InitialContext object with the Properties object values to initialize the object.
7. Call the InitialContextext object's `lookup` method to get a reference to the home interface for the bean that you want. Specify the JNDI name of the bean as the `lookup` argument.
8. Call the `create` method of the bean's home object to create a new instance of the bean. If you are using Entity beans, you typically use a finder method instead. A finder method locates one or more existing entity beans.

9. Now you can use the bean's methods as required by your application.
10. When finished, call the context object's `close` method to close the object.

The following code shows this process using a simple Java Entity bean on a JRun 4.0 server. It calls the bean's `getMessage` method to obtain a message.

```
<html>
<head>
  <title>cfobject Test</title>
</head>

<body>
<H1>cfobject Test</H1>
<!--- Create the Context object to get at the static fields. --->
<CFOBJECT
  action=create
  name=ctx
  type="JAVA"
  class="javax.naming.Context">

<!--- Create the Properties object and call an explicit constructor--->
<CFOBJECT
  action=create
  name=prop
  type="JAVA"
  class="java.util.Properties">

<!--- Call the init method (provided by cfobject)
    to invoke the Properties object constructor. --->
<cfset prop.init()>

<!--- Specify the properties These are required for a remote server only --->
<cfset prop.put(ctx.INITIAL_CONTEXT_FACTORY,
  "jrun.naming.JRunContextFactory")>
<cfset prop.put(ctx.PROVIDER_URL, "localhost:2908")>
<!--- <cfset prop.put(ctx.SECURITY_PRINCIPAL, "admin")>
    <cfset prop.put(ctx.SECURITY_CREDENTIALS, "admin")>
 --->
<!--- Create the InitialContext --->
<CFOBJECT
  action=create
  name=initContext
  type="JAVA"
  class="javax.naming.InitialContext">

<!--- Call the init method (provided through cfobject)
    to pass the properties to the InitialContext constructor. --->
<cfset initContext.init(prop)>

<!--- Get reference to home object. --->
<cfset home = initContext.lookup("SimpleBean")>

<!--- Create new instance of entity bean.
    (hard-wired account number). Alternatively,
```

```
    you would use a find method to locate an existing entity bean. --->
<cfset mySimple = home.create()>

<!--- Call a method in the entity bean. --->
<cfset myMessage = mySimple.getMessage()>

<cfoutput>
  #myMessage#<br>
</cfoutput>

<!--- Close the context. --->
<cfset initContext.close()>

</body>
</html>
```

Using a custom Java class

The following code provides a more complex custom class than in the example "Creating and using a simple Java class" on page 930. The Example class manipulates integer, float, array, Boolean, and Example object types.

The Example class

The following Java code defines the Example class. The Java class Example has one public integer member, `mPublicInt`. Its constructor initializes `mPublicInt` to 0 or an integer argument. The class has the following public methods:

Method	Description
ReverseString	Reverses the order of a string.
ReverseStringArray	Reverses the order of elements in an array of strings.
Add	Overloaded: Adds and returns two integers or floats or adds the `mPublicInt` members of two Example class objects and returns an Example class object.
SumArray	Returns the sum of the elements in an integer array.
SumObjArray	Adds the values of the `mPublicInt` members of an array of Example class objects and returns an Example class object.
ReverseArray	Reverses the order of an array of integers.
Flip	Switches a Boolean value.

```
public class Example {
  public int mPublicInt;

  public Example() {
    mPublicInt = 0;
  }

  public Example(int IntVal) {
    mPublicInt = IntVal;
  }
```

```
  public String ReverseString(String s) {
    StringBuffer buffer = new StringBuffer(s);
    return new String(buffer.reverse());
  }

  public String[] ReverseStringArray(String [] arr) {
    String[] ret = new String[arr.length];
    for (int i=0; i < arr.length; i++) {
      ret[arr.length-i-1]=arr[i];
    }
    return ret;
  }

  public int Add(int a, int b) {
    return (a+b);
  }

  public float Add(float a, float b) {
    return (a+b);
  }

  public Example Add(Example a, Example b) {
    return new Example(a.mPublicInt + b.mPublicInt);
  }

  static public int SumArray(int[] arr) {
    int sum=0;
    for (int i=0; i < arr.length; i++) {
      sum += arr[i];
    }
    return sum;
  }

  static public Example SumObjArray(Example[] arr) {
    Example sum= new Example();
    for (int i=0; i < arr.length; i++) {
      sum.mPublicInt += arr[i].mPublicInt;
    }
    return sum;
  }

  static public int[] ReverseArray(int[] arr) {
    int[] ret = new int[arr.length];
    for (int i=0; i < arr.length; i++) {
      ret[arr.length-i-1]=arr[i];
    }
    return ret;
  }

  static public boolean Flip(boolean val) {
    System.out.println("calling flipboolean");
    return val?false:true;
  }
}
```

The useExample ColdFusion page

The following useExample.cfm page uses the `Example` class to manipulate numbers, strings, Booleans, and Example objects. The `JavaCast` CFML function ensures that CFML variables convert into the appropriate Java data types.

```
<html>
<head>
  <title>CFOBJECT and Java Example</title>
</head>
<body>

<!--- Create a reference to an Example object --->
<cfobject action=create type=java class=Example name=obj>
<!--- Create the object and initialize its public member to 5 --->
<cfset x=obj.init(JavaCast("int",5))>

<!--- Create an array and populate it with string values,
     then use the Java object to reverse them. --->
<cfset myarray=ArrayNew(1)>
<cfset myarray[1]="First">
<cfset myarray[2]="Second">
<cfset myarray[3]="Third">
<cfset ra=obj.ReverseStringArray(myarray)>

<!--- Display the results --->
<cfoutput>
  <br>
  original array element 1: #myarray[1]#<br>
  original array element 2: #myarray[2]#<br>
  original array element 3: #myarray[3]#<br>
  after reverse  element 1: #ra[1]#<br>
  after reverse  element 2: #ra[2]#<br>
  after reverse  element 3: #ra[3]#<br>
  <br>
</cfoutput>


<!--- Use the Java object to flip a Boolean value, reverse a string,
     add two integers, and add two float numbers --->
<cfset c=obj.Flip(true)>
<cfset StringVal=obj.ReverseString("This is a test")>
<cfset IntVal=obj.Add(JavaCast("int",20),JavaCast("int",30))>
<cfset FloatVal=obj.Add(JavaCast("float",2.56),JavaCast("float",3.51))>

<!--- Display the results --->
<cfoutput>
  <br>
  StringVal: #StringVal#<br>
  IntVal: #IntVal#<br>
  FloatVal: #FloatVal#<br>
  <br>
</cfoutput>
```

```
<!--- Create a two-element array, sum its values,
     and reverse its elements --->
<cfset intarray=ArrayNew(1)>
<cfset intarray[1]=1>
<cfset intarray[2]=2>
<cfset IntVal=obj.sumarray(intarray)>
<cfset reversedarray=obj.ReverseArray(intarray)>

<!--- Display the results --->
<cfoutput>
  <br>
  IntVal1 :#IntVal#<br>
  array1: #reversedarray[1]#<br>
  array2: #reversedarray[2]#<br>
  <br>
</cfoutput><br>

<!--- Create a ColdFusion array containing two Example objects.
     Use the SumObjArray method to add the objects in the array
     Get the public member of the resulting object--->
<cfset oa=ArrayNew(1)>
<cfobject action=create type=java class=Example name=obj1>
<cfset VOID=obj1.init(JavaCast("int",5))>
<cfobject action=create type=java class=Example name=obj2>
<cfset VOID=obj2.init(JavaCast("int",10))>
<cfset oa[1] = obj1>
<cfset oa[2] = obj2>
<cfset result = obj.SumObjArray(oa)>
<cfset intval = result.mPublicInt>

<!--- Display the results --->
<cfoutput>
  <br>
  intval1: #intval#<br>
  <br>
</cfoutput><br>
</body>
</html>
```

CHAPTER 38
Integrating COM and CORBA Objects in CFML Applications

This chapter describes how to use the `cfobject` tag to invoke COM (Component Object Model) or DCOM (Distributed Component Object Model) and CORBA (Common Object Request Broker) objects.

Contents

About COM and CORBA 945
Creating and using objects 947
Getting started with COM and DCOM 948
Creating and using COM objects 951
Getting started with CORBA 961
Creating and using CORBA objects 961
CORBA example 969

About COM and CORBA

This section provides some basic information on COM and CORBA objects supported in Macromedia ColdFusion MX and provides resources for further inquiry.

About objects

COM and CORBA are two of the *object* technologies supported by ColdFusion MX. Other object technologies include Java and ColdFusion components. For more information on ColdFusion components see Chapter 10, "Building and Using ColdFusion Components," on page 201.

An object is a self-contained module of data and its associated processing. An object is a building block that you can put together with other objects and integrate into ColdFusion code to create an application.

An object is represented by a handle, or name. Objects have *properties* that represent information. Objects also provide *methods* for manipulating the object and getting data from it. The exact terms and rules for using objects vary with the object technology.

You create instances of objects using the `cfobject` tag or the `CreateObject` function. You then use the object and its methods in ColdFusion tags, functions, and expressions. For more information on the ColdFusion syntax for using objects, see "Creating and using objects" on page 947.

About COM and DCOM

COM (Component Object Model) is a specification and a set of services defined by Microsoft to enable component portability, reusability, and versioning. DCOM (Distributed Component Object Model) is an implementation of COM for distributed services, which allows access to components residing on a network.

COM objects can reside locally or on any network node. COM is supported on Microsoft Windows platforms.

For more information on COM, go to the Microsoft COM website, www.microsoft.com/com.

About CORBA

CORBA (Common Object Request Broker Architecture) is a distributed computing model for object-oriented applications defined by the Object Management Group (OMG). In this model, an object is an encapsulated entity whose services are accessed only through well-defined interfaces. The location and implementation of each object is hidden from the client requesting the services. ColdFusion supports CORBA 2.3 on both Windows and UNIX.

CORBA uses an Object Request Broker (ORB) to send requests from applications on one system to objects executing on another system. The ORB allows applications to interact in a distributed environment, independent of the computer platforms on which they run and the languages in which they are implemented. For example, a ColdFusion application running on one system can communicate with an object that is implemented in C++ on another system.

CORBA follows a client-server model. The client invokes operations on objects that are managed by the server, and the server replies to requests. The ORB manages the communications between the client and the server using the Internet Inter-ORB Protocol (IIOP).

Each CORBA object has an interface that is defined in the CORBA Interface Definition Language (IDL). The CORBA IDL describes the operations that can be performed on the object, and the parameters of those operations. Clients do not have to know anything about how the interface is implemented to make requests.

To request a service from the server, the client application gets a handle to the object from the ORB. It uses the handle to call the methods specified by the IDL interface definition. The ORB passes the requests to the server, which processes the requests and returns the results to the client.

For information about CORBA, see the following OMG website, which is the main web repository for CORBA information: www.omg.com.

Creating and using objects

You use the `cfobject` tag or the `CreateObject` function to create a named instance of an object. You use other ColdFusion tags, such as `cfset` and `cfoutput`, to invoke the object's properties and methods.

The following sections provide information about creating and using objects that applies to both COM and CORBA objects. The examples assume a sample object named "obj", and that the object has a property called "Property", and methods called "Method1", "Method2", and "Method3".

Creating objects

You create, or *instantiate* (create a named instance of) an object in ColdFusion with the `cfobject` tag or `CreateObject` function. The specific attributes or parameters that you use depend on the type of object you use, and are described in detail in "Creating and using COM objects" on page 951 and "Creating CORBA objects" on page 961. The following examples use a `cfobject` tag to create a COM object and a `CreateObject` function to create a CORBA object:

```
<cfobject type="COM" action="Create" name="obj" class="sample.MyObject">
obj = CreateObject("CORBA", "d:\temp\tester.ior", "IOR", "Visibroker")
```

ColdFusion releases any object created by `cfobject` or `CreateObject`, or returned by other objects, at the end of the ColdFusion page execution.

Using properties

Use standard ColdFusion statements to access properties as follows:

- To set a property, use a statement or `cfset` tag, such as the following:
  ```
  <cfset obj.property = "somevalue">
  ```
- To get a property, use a statement or `cfset` tag, such as the following:
  ```
  <cfset value = obj.property>
  ```

As shown in this example, you do not use parentheses on the right side of the equation to get a property value.

Calling methods

Object methods usually take zero or more arguments. You send In arguments, whose values are not returned to the caller by value. You send Out and In,Out arguments, whose values are returned to the caller, by reference. Arguments sent by reference usually have their value changed by the object. Some methods have return values, while others might not.

Use the following techniques to call methods:

- If the method has no arguments, follow the method name with empty parentheses, as in the following `cfset` tag:

```
<cfset retVal = obj.Method1()>
```

- If the method has one or more arguments, put the arguments in parentheses, separated by commas, as in the following example, which has one integer argument and one string argument:

```
<cfset x = 23>
<cfset retVal = obj.Method1(x, "a string literal")>
```

- If the method has reference (Out or In,Out) arguments, use double quotation marks (") around the name of the variable you are using for these arguments, as shown for the variable x in the following example:

```
<cfset x = 23>
<cfset retVal = obj.Method2("x",  "a string literal")>
<cfoutput> #x#</cfoutput>
```

  In this example, if the object changes the value of x, it now contains a value other than 23.

Calling nested objects

ColdFusion supports nested (scoped) object calls. For example, if an object method returns another object, and you must invoke a property or method on that object, you can use the syntax in either of the following examples:

```
<cfset prop = myObj.X.Property>
```

or

```
<cfset objX = myObj.X>
<cfset prop = objX.Property>
```

Getting started with COM and DCOM

ColdFusion is an automation (late-binding) COM client. As a result, the COM object must support the IDispatch interface, and arguments for methods and properties must be standard automation types. Because ColdFusion is a typeless language, it uses the object's type information to correctly set up the arguments on call invocations. Any ambiguity in the object's data types can lead to unexpected behavior.

In ColdFusion, you should only use server-side COM objects, which do not have a graphical user interface. If your ColdFusion application invokes an object with a graphical interface in a window, the component might appear on the web server desktop, not on the user's desktop. This can take up ColdFusion server threads and prevent further web server requests from being serviced.

ColdFusion can call Inproc, Local, or Remote COM objects. The attributes specified in the `cfobject` tag determine which type of object is called.

COM requirements

To use COM components in your ColdFusion application, you need at least the following items:

- The COM objects (typically DLL or EXE files) that you want to use in your ColdFusion application pages. These components should allow late binding; that is, they should implement the IDispatch interface.
- Microsoft OLE/COM Object Viewer, available from Microsoft at www.microsoft.com/com/resources/oleview.asp. This tool lets you view registered COM objects.

  Object Viewer lets you view an object's class information so that you can properly define the `class` attribute for the `cfobject` tag. It also displays the object's supported interfaces, so you can discover the properties and methods (for the IDispatch interface) of the object.

Registering the object

After you acquire an object, you must register it with Windows for ColdFusion (or any other program) to find it. Some objects have setup programs that register objects automatically, while others require manual registration.

You can register Inproc object servers (.dll or .ocx files) manually by running the regsvr32.exe utility using the following form:

```
regsvr32 c:\path\servername.dll
```

You typically register Local servers (.exe files) either by starting them or by specifying a command line parameters, such as the following:

```
C:\pathname\servername.exe -register
```

Finding the component ProgID and methods

Your COM object supplier should provide documentation that explains each of the component's methods and the ProgID. If you do not have documentation, use either the ColdFusion `cfdump` tag or the OLE/COM Object Viewer to view the component's interface.

Using the cfdump tag to view COM object interfaces

Effective with ColdFusion MX, the ColdFusion `cfdump` tag displays the following information about a COM object:

- Public methods
- Put properties
- Get properties

The method and property information includes the parameter or property types and whether they are in, out, optional, or retval values. The `cfdump` tag output does not include the object's ProgID.

Note: The dump header indicates the ColdFusion object class, which is coldfusion.runtime.com.ComProxy, and the COM object CLSID.

Using the OLE/COM Object Viewer

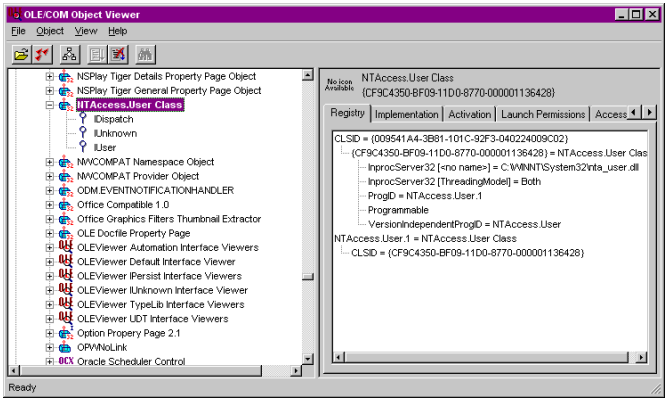
The OLE/COM Object Viewer installation installs the executable, by default, as \mstools\bin\oleview.exe. You use the Object Viewer to retrieve a COM object's ProgID, as well as its methods and properties.

To find an object in the Object Viewer, it must be registered, as described in "Registering the object" on page 949. The Object Viewer retrieves all COM objects and controls from the Registry, and presents the information in a simple format, sorted into groups for easy viewing.

By selecting the category and then the component, you can see the ProgID of a COM object. The Object Viewer also provides access to options for the operation of the object.

To view an object's properties:

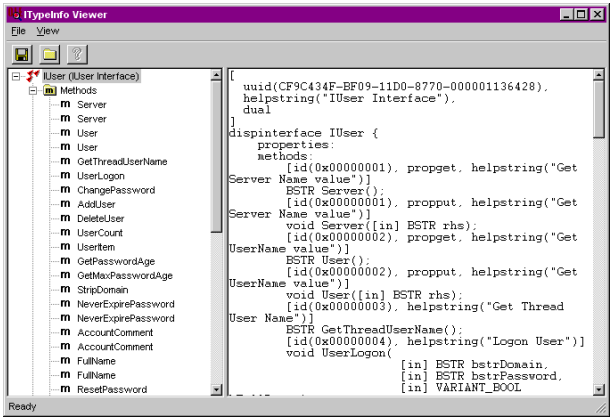
1. Open the Object Viewer, as shown in the following figure, and scroll to the object that you want to examine.



2. Select and expand the object in the left pane of the Object Viewer.

3. Right-click the object to view it, including the TypeInfo.

   If you view the TypeInfo, you see the object's methods and properties, as shown in the following figure. Some objects do not have access to the TypeInfo area, which is determined when an object is built and by the language used.



Creating and using COM objects

You must use the `cfobject` tag or the `CreateObject` function to create an instance of the COM object (component) in ColdFusion before your application pages can invoke any methods or assign any properties in the component.

For example, the following code uses the `cfobject` tag to create the Windows CDO (Collaborative Data Objects) for NTS NewMail object to send mail:

```
<cfobject type="COM"
     action="Create"
     name="Mailer"
     class="CDONTS.NewMail">
```

The following line shows how to use the corresponding `CreateObject` function in CFScript:

```
Mailer = CreateObject("COM", "CDONTS.NewMail");
```

The examples in later sections in this chapter use this object.

Note: CDO is installed by default on all Windows NT and 2000 operating systems that have installed the Microsoft SMTP server. In Windows NT Server environments, the SMTP server is part of the Option Pack 4 setup. In Windows 2000 Server and Workstation environments, it is bundled with the operating system. For more information on CDO for NTS, see http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/cdo/_olemsg_overview_of_cdo.htm.

The CDO for NTS NewMail component includes a number of methods and properties to perform a wide range of mail-handling tasks. (In the OLE/COM Object Viewer, methods and properties might be grouped together, so you could find it difficult to distinguish between them at first.)

The CDO for NTS NewMail object includes the following properties:

```
Body [ String ]
Cc  [ String ]
From  [ String ]
Importance  [ Long ]
Subject  [ String ]
To  [ String ]
```

You use these properties to define elements of your mail message. The CDO for NTS NewMail object also includes a `send` method which has a number of optional arguments to send messages.

Connecting to COM objects

The `action` attribute of the `cfobject` tag provides the following two ways to connect to COM objects:

Create method (`cfobject action="Create"`) Takes a COM object, typically a DLL, and instantiates it prior to invoking methods and assigning properties.

Connect method (`cfobject action="Connect"`) Links to an object, typically an executable, that is already running on the server.

You can use the optional `cfobject context` attribute to specify the object context. If you do not specify a context, ColdFusion uses the setting in the Registry. The following table describes the `context` attribute values:

Attribute value	Description
InProc	An in-process server object (typically a DLL) that is running in the same process space as the calling process, such as ColdFusion.
local	An out-of-process server object (typically an EXE file) that is running outside the ColdFusion process space but running locally on the same server.
remote	An out-of-process server object (typically an EXE file) that is running remotely on the network. If you specify `remote`, you must also use the `server` attribute to identify where the object resides.

Setting properties and invoking methods

The following example, which uses the sample Mailer COM object, shows how to assign properties to your mail message and how to execute component methods to handle mail messages.

In the example, form variables contain the method parameters and properties, such as the name of the recipient, the desired e-mail address, and so on:

```
<!--- First, create the object --->
<cfobject type="COM"
  action="Create"
  name="Mailer"
  class="CDONTS.NewMail">
```

```
<!--- Second, use the form variables from the user entry form to populate a
  number
    of properties necessary to create and send the message. --->
<cfset Mailer.From = "#Form.fromName#">
<cfset Mailer.To = "#Form.to#">
<cfset Mailer.Subject = "#Form.subject#">
<cfset Mailer.Importance = 2>
<cfset Mailer.Body = "#Form.body#">
<cfset Mailer.Cc = "#Form.cc#">

<!--- Last, use the Send() method to send the message.
  Invoking the Send() method destroys the object.--->
<cfset Mailer.Send()>
```

Note: Use the `cftry` and `cfcatch` tags to handle exceptions thrown by COM objects. For more information on exception handling, see Chapter 14, "Handling runtime exceptions with ColdFusion tags," on page 322.

Releasing COM objects

By default, COM object resources are released when the Java garbage collector cleans them. You can use the `ReleaseCOMObject` function to immediately release resources if an object is no longer needed.

Use the `ReleaseCOMObject` function to release COM objects that are launched as an external process, such as Microsoft Excel. The garbage collector might not clean these processes in a short time, resulting in multiple external processes running, which drains system resources.

If the COM object has an end method, such as a quit method that terminates the program, call this method before you call the `ReleaseComObject` function. If you use the `ReleaseComObject` function on an object that is in use, the object is prematurely released and your application will get exceptions.

Example

The following example creates a Microsoft Excel application object, uses it, then releases the object when it is no longer needed:

```
<h3>ReleaseComObject Example</h3>
<cfscript>
obj = CreateObject("Com", "excel.application.9");
//code that uses the object goes here
obj.quit();
ReleaseComObject(obj);
</cfscript>
```

General COM object considerations

When you use COM objects, consider the following to prevent and resolve errors:

- Ensuring correct threading
- Using input and output arguments
- Understanding common COM-related error messages

The following sections describe these issues.

Ensuring correct threading

Improper threading can cause serious problems when using a COM object in ColdFusion. Make sure that the object is *thread-safe*. An object is thread-safe if it can be called from many programming threads simultaneously, without causing errors.

Visual Basic ActiveX DLLs are typically not thread-safe. If you use such a DLL in ColdFusion, you can make it thread-safe by using the OLE/COM Object Viewer to change the object's threading model to the Apartment model.

If you are planning to store a reference to the COM object in the Application, Session, or Server scope, do not use the Apartment threading model. This threading model is intended to service only a single request. If your application requires you to store the object in any of these scopes, keep the object in the Both threading model, and lock all code that accesses the object, as described in Chapter 15, "Locking code with cflock," on page 360.

To change the threading model of a COM Object:

1. Open the OLE/COM Object Viewer.
2. Select All Objects under Object Classes in the left pane.
3. Locate your COM object. The left pane lists these by name.
4. Select your object.
5. Select the Implementation tab in the right pane.
6. Select the Inproc Server tab, below the App ID field.
7. Select the Threading Model drop down menu and select Apartment or Both, as appropriate.

Using input and output arguments

COM object method in arguments are passed by value. The COM object gets a copy of the variable value, so you can specify a ColdFusion variable without surrounding it with quotation marks.

COM object out method arguments are passed by reference. The COM object modifies the contents of the variable on the calling page, so the calling page can use the resulting value. To pass a variable by reference, surround the name of an existing ColdFusion variable with quotation marks. If the argument is a numeric type, assign the variable a valid number before you make the call. For example:

```
<cfset inStringArg="Hello Object">
<cfset outNumericArg=0>
<cfset result=myCOMObject.calculate(inStringArg, "outNumericArg")>
```

The string "Hello Object" is passed to the object's calculate method as an input argument. The value of outNumericArg is set by the method to a numeric value.

Understanding common COM-related error messages

The following table described some error messages you might encounter when using COM objects:

Error	Cause
Error Diagnostic Information Error trying to create object specified in the tag. COM error 0x800401F3. Invalid class string.	The COM object is not registered or does not exist.
Error Diagnostic Information Error trying to create object specified in the tag. COM error 0x80040154. Class not registered.	The COM object is not registered or does not exist. This error usually occurs when an object existed previously, but was uninstalled.
Error Diagnostic Information Failed attempting to find "SOMEMETHOD" property/method on the object COM error 0x80020006. Unknown name.	The COM object was instantiated correctly, but the method you specified does not exist.

Accessing Complex COM Objects using Java proxies

ColdFusion MX supports Java proxies to access COM objects. If you do not create Java proxies in advance, ColdFusion must dynamically discover the COM interface. This technique can have two disadvantages:

- Dynamic discovery takes time and can reduce server performance with frequently used complex COM objects.
- Dynamic discovery uses the IDispatcher interface to determine the COM object features, and might not handle some complex COM interfaces.

To overcome these problems, ColdFusion MX includes a utility, com2java.exe, that creates static Java stub proxy classes for COM objects. ColdFusion can use these Java stubs to access COM objects more efficiently than when it creates the proxies dynamically. Additionally, the com2java.exe utility can create stubs for features that the dynamic proxy generator might miss.

ColdFusion MX ships with pregenerated stubs for the Windows XP, Windows 2000, and Windows 97 editions of Microsoft Excel, Microsoft Word, and Microsoft Access. ColdFusion MX is configured to automatically use these stubs.

If you create Java stub files for a COM object, you continue to use the `cfobject` tag with a `type` attribute value of `COM`, or the `CreateObject` function with a first argument of `COM`, and you access the object properties and methods as you normally do for COM objects in ColdFusion.

Use the following steps to use the com2java.exe utility. This procedure uses Microsoft Outlook as an example.

To create Java stub files for COM objects:

1. Configure your system as follows:
   a. Ensure that a JDK (Java Development Kit) is correctly installed, including proper configuration of the CLASSPATH and the command prompt PATH variable.
   b. Add *CF_root*\lib\jintegra.jar to your CLASSPATH.
2. Make a new directory for the Java stub files; for example:

   ```
   mkdir C:\src\outlookXP
   ```

   This directory can be temporary. You add files from the directory to a ColdFusion JAR file.
3. Run the *CF_root*\Jintegra\bin\com2java.exe program from a command line or the Windows Start Menu. A window appears.
   a. If a COM class implements multiple interfaces that define methods with the same names, click the Options button and clear the Implement interfaces that may conflict option. The generated Java stub classes do not implement the additional, conflicting, interfaces. You can still access the interfaces using the getAs*XXX* method that is generated. See the generated comments in the Java files.
   b. Click on the `Select` button.
   c. Select your COM object's Type Library or DLL. For Microsoft Outlook in Windows XP, it is normally Program Files\Microsoft Office\Office10\MSOUTL.OLB.
   d. Enter a package name (for example, *outlookXP*) in the Java package field in the com2java dialog box. This package will contain all the classes for the Java stubs for the COM object.

      Note: Macromedia uses a package name that starts with *coldfusion.runtime.com.com2java* for the packages that contain the preinstalled Java stubs for Microsoft Excel, Microsoft Word, and Microsoft Access. For example, the name for the package containing the Microsoft Word XP Java stub classes is *coldfusion.runtime.com.com2java.wordXP*. This package name hierarchy results in the wordXP classes having a path inside the msapps.jar file of coldfusion\runtime\com\com2java\wordXP\className.class. Although this naming convention is not necessary, consider using a similar package naming convention for clarity, if you use many COM objects.

   e. Click the Generate Proxies button to display the File browser. Select the directory you created in step 2., and click the file browser OK button to generate the stub files.
   f. Click Close to close the com2java.exe utility.

   The files generated in your directory include the following:
   - A Java interface and proxy class for each COM interface
   - A Java class for each COM class
   - A Java interface for each ENUM (a set of constant definitions)

4. Compile your Java code. In a command prompt, do the following:
   a Make the directory that contains the Java stubs (in this example, C:\src\outlookXP) your working directory.
   b Enter the following line:

   ```
   javac -J-mx100m -J-ms100m *.java
   ```

   The compiler switches ensure that you have enough memory to compile all the necessary files.

   Note: If you did not put jintegra.jar on your CLASSPATH in step 1b, add the switch `-classpath:/cf_root/lib/jintegra.jar`, where *cf_root* is the directory where ColdFusion is installed, to the command.

5. Ensure that the ColdFusion server is not running. To stop the ColdFusion server, open the Services control panel, select ColdFusion MX application server, and click Stop.
6. Add your .class files to the ColdFusion Microsoft application Java stubs file by doing the following:
   a In the Windows Command prompt, make the parent directory of the directory that contains your class files your working directory. In this example, make c:\src your working director by entering **cd ..** in the Command prompt from step 4.
   b Enter the following command:

   ```
   jar -uvf cf_root\lib\msapps.jar directoryName\*.class
   ```

   Where *cf_root* is the directory where ColdFusion MX is installed and *directoryName* is the name of the directory that contains the class files. For the OutlookXP example, enter the following line:

   ```
   jar -uvf C:\CFusionMX7\lib\msapps.jar outlookXP\*.class
   ```

7. Update the *cf_root* /lib/neo-comobjmap.xml file by appending your object definition to the list. The object definition consists of the following lines:

   ```
   <var name="progID">
   <string>PackageName.mainClass</string>
   </var>
   ```

   Use the following values in these lines:

   ProgID The COM object's ProgID, as displayed in the OLE/COM object viewer.

   PackageName The package name you specified in step 3c.

   mainClass The main class of the COM object. The main class contains the methods you invoke. For many Microsoft applications, this class is Application. In general, the largest class file created in step 4. is the main class.

   For example, to add outlookXP to neo-comobjmap.xml, add the lines in bold text above the `</struct>` end tag:

   ```
   <var name="access.application.9">
   <string>coldfusion.runtime.com.com2java.access2k.Application</string>
   </var>
   <var name="outlook.application.10">
   <string>outlookXP.Application</string>
   ```

```
</var>
</struct>
```

In this example, *outlook.application.10* is the ProgID of the Outlook COM object, *outlookXP* is the package name you specified in step 3c, and *Application* is the COM object's main class.

8. Restart the ColdFusion server: Open the Services control panel, select ColdFusion MX application server, and click the Start button.
9. After you have installed the stubs, you can delete the directory you created in step 2., including all its contents.

Using the Application Scope to improve COM performance

The Java call to create a new COM object instance can take substantial time. As a result, creating COM objects in ColdFusion MX can be substantially slower than in ColdFusion 5. For example, on some systems, creating a Microsoft Word application object could take over one second using ColdFusion MX, while on the same system, the overhead of creating the Word object might be about 200 milliseconds.

Therefore, in ColdFusion MX, you can improve COM performance substantially if you can share a single COM object in the Application scope among all pages.

Use this technique only if the following are true:

- The COM object does not need to be created for every request or session. (For session-specific objects, consider using the technique described in this section with the Session scope in place of the Application scope.)
- The COM object is designed for sharing.

Because the object can be accessed from multiple pages and sessions simultaneously, you must also consider the following threading and locking issues:

- For best performance, the object should be multithreaded. Otherwise, only one request can access the object at a time.
- Lock the code that accesses and modifies common data. In general, you do not have to lock code that modifies a shared object's data, including writable properties or file contents, if the data (as opposed to the object) is not shared by multiple requests. However, specific locking needs depend on the COM object's semantics, interface, and implementation.
- All `cflock` tags in the application that use an Application scope lock share one lock. Therefore, code that accesses a frequently used COM object inside an Application scope lock can become a bottleneck and reduce throughput if many users request pages that use the object. You might be able to avoid some contention by putting code that uses the COM object in named locks; you must put the code that creates the object in an Application scope lock.

Note: You can also improve the performance of some COM objects by creating Java stubs, as described in "Accessing Complex COM Objects using Java proxies" on page 955. Using a Java stub does not improve performance as much as sharing the COM object, but the technique works with all COM objects. Also, you must generate Java stubs to correctly access complex COM objects that do not properly make all their features available through the COM IDispatcher interface. Therefore, to get the greatest performance increase and prevent possible problems, use both techniques.

Example 1: Using the FileSystem object

The following example uses the Microsoft FileSystem Scripting object in the Application scope. This code creates a user-defined function that returns a structure that consists of the drive letters and free disk space for all hard drives on the system.

```
<cfapplication name="comtest" clientmanagement="No" Sessionmanagement="yes">

<!--- Uncomment the following line if you must delete the object from the
Application scope during debugging. Then restore the comments.
This technique is faster than stopping and starting the ColdFusion server. --->
 <!--- <cfset structdelete(Application, "fso")> --->

<!--- The getFixedDriveSpace user-defined function returns a structure with
the drive letters as keys and the drive's free space as data for all fixed
drives on a system. The function does not take any arguments --->

<cffunction name="getFixedDriveSpace" returnType="struct" output=True>
  <!--- If the FileSystemObject does not exist in the Application scope,
  create it. --->
  <!--- For information on the use of initialization variables and locking in
    this code, see “Locking application variables efficiently” in Chapter 15,
    “Using Persistent Data and Locking” --->
  <cfset fso_is_initialized = False>
  <cflock scope="application" type="readonly" timeout="120">
    <cfset fso_is_initialized = StructKeyExists(Application, "fso")>
  </cflock>
  <cfif not fso_is_initialized >
    <cflock scope="Application" type="EXCLUSIVE" timeout="120">
      <cfif NOT StructKeyExists(Application, "fso")>
        <cfobject type="COM" action="create"
  class="Scripting.FileSystemObject"
          name="Application.fso" server="\\localhost">
      </cfif>
    </cflock>
  </cfif>

  <!--- Get the drives collection and loop through it to populate the
    structure. --->
  <cfset drives=Application.fso.drives()>
  <cfset driveSpace=StructNew()>
  <cfloop collection="#drives#" item="curDrive">
    <!--- A DriveType of 2 indicates a fixed disk --->
    <cfif curDrive.DriveType IS 2>
    <!--- Use dynamic array notation with the drive letter for the struct key
    --->
      <cfset driveSpace["#curDrive.DriveLetter#"]=curDrive.availablespace>
    </cfif>
  </cfloop>
  <cfreturn driveSpace>
</cffunction>

<!--- Test the function. Get the execution time for running the function --->
<cfset start = getTickCount()>
<cfset DriveInfo=getFixedDriveSpace()>
```

```
<h3>Getting fixed drive available space</h3>
<cfoutput>Execution Time: #int(getTickCount()-start)# milliseconds</
  cfoutput><br><br>
<cfdump label="Drive Free Space" var="#driveInfo#">
```

Example 2: Using the Microsoft Word application object

The following example uses the Microsoft Word application COM object in the Application scope to convert a Word document to HTML. This example works with Word 2000 as written. To work with Word 97, change "Val(8)" to "Val(10)".

This example uses an Application scope lock to ensure that no other page interrupts creating the object. Once the Word object exists, the example uses a named lock to prevent simultaneous access to the file that is being converted.

```
<cfapplication name="comtest" clientmanagement="No" Sessionmanagement="yes">
<!--- Uncomment the following line if you need to delete the object from the
Application scope --->
<!--- <cfset structdelete(Application, "MyWordObj")> --->

<!--- use the GetTickCount function to get a current time indicator, used for
     displaying the total processing time. --->
<cfset start = GetTickCount()>
<!--- If necessary, create the Word.application object and put it in the
     Application scope --->
<cfset WordObj_is_initialized = False>
<cflock scope="application" type="readonly" timeout=120>
  <cfset WordObj_is_initialized = StructKeyExists(application, "MyWordObj")>
</cflock>
<cfif not WordObj_is_initialized >
  <cflock scope="Application" type="exclusive" timeout="120">
    <cfif not StructKeyExists(application, "MyWordObj")>

<!--- First try to connect to an existing Word object --->
       <cftry>
         <cfobject type="com"
           action="connect"
           class="Word.application"
           name="Application.MyWordobj"
           context="local">
         <cfcatch>
<!--- There is no existing object, create one --->
           <cfobject type="com"
             action="Create"
             class="Word.application"
             name="Application.MyWordobj"
             context="local">
         </cfcatch>
       </cftry>

       <cfset Application.mywordobj.visible = False>
    </cfif>
  </cflock>
</cfif>
```

```
<!--- Convert a Word document in temp.doc to an HTML file in temp.htm. --->
<!--- Because this example uses a fixed filename, multiple pages might try
  to use the file simultaneously. The lock ensures that all actions from
  reading the input file through closing the output file are a single "atomic"
  operation, and the next page cannot access the file until the current page
  completes all processing.
  Use a named lock instead of the Application scope lock to reduce lock
  contention. --->
<cflock name="WordObjLock" type="exclusive" timeout="120">
  <cfset docs = application.mywordobj.documents()>
  <cfset docs.open("c:\CFusionMX7\wwwroot\temp.doc")>
  <cfset converteddoc = application.mywordobj.activedocument>
  <!--- Val(8) works with Word 2000. Use Val(10) for Word 97 --->
  <cfset converteddoc.saveas("c:\CFusionMX7\wwwroot\temp.htm",val(8))>
  <cfset converteddoc.close()>
</cflock>

<cfoutput>
  Conversion of temp.htm Complete<br>
  Execution Time: #int(getTickCount()-start)# milliseconds<br>
</cfoutput>
```

Getting started with CORBA

The ColdFusion `cfobject` tag and `CreateObject` function support CORBA through the Dynamic Invocation Interface (DII). As with COM, the object's type information must be available to ColdFusion. Therefore, an IIOP-compliant Interface Repository (IR) must be running on the network, and the object's Interface Definition Language (IDL) specification must be registered in the IR. If your application uses a naming service to get references to CORBA objects, a naming service must also be running on the network.

ColdFusion loads ORB runtime libraries at startup using a connector, which does not tie ColdFusion customers to a specific ORB vendor. ColdFusion currently includes connectors for the Borland Visibroker 4.5 ORB. The source necessary to write connectors for other ORBs is available under NDA to select third-party candidates and ORB vendors

You must take several steps to configure and enable CORBA access in ColdFusion. For detailed instructions, see *Installing and Using ColdFusion MX*.

Note: When you enable CORBA access in ColdFusion, one step requires you to start the Interface Repository using an IDL file. This file must contain the IDL for *all* the CORBA objects that you invoke in ColdFusion applications on the server.

Creating and using CORBA objects

The following sections describe how to create, or instantiate, a CORBA object and how to use it in your ColdFusion application.

Creating CORBA objects

The `cfobject` tag and `CreateObject` functions create in ColdFusion a stub, or proxy object, for the CORBA object on the remote server. You use this stub object to invoke the remote object.

The following table describes the attributes you use in the `cfobject` tag to create a CORBA object:

Attribute	Description
type	Must be CORBA. COM is the default.
context	Specifies the CORBA binding method, that is, how the object is obtained, as follows: • `IOR` Uses a file containing the object's unique Interoperable Object Reference. • `NameService` Uses a naming service.
class	Specifies the information required for the binding method to access the object. If you set the `context` attribute to `IOR`, The `class` attribute must be to the full pathname of a file containing the string version of the IOR. ColdFusion must be able to read this IOR file at all times, so make it local to the server or put it on the network in an accessible location. If you set the `context` attribute to `NameService`, The `class` attribute must be a name delimited by forward slashes (/), such as MyCompany/Department/Dev. You can use period-delimited "kind" identifiers as part of the class attribute; for example, Macromedia.current/Eng.current/CF"
name	Specifies the name (handle) that your application uses to call the object's interface.
locale	(Optional) Identifies the connector configuration. You can omit this option if ColdFusion MX Administrator has only one connector configuration, or if it has multiple connector configurations and you want to use the one that is currently selected in the Administrator. If you specify this attribute, it must be an ORB name you specified in the CORBA Connector ORB Name field when you configured a CORBA connector in ColdFusion MX Administrator; for example, Visibroker.

For example, use the following CFML to invoke a CORBA object specified by the tester.ior file if you configured your ORB name as Visibroker:

```
<cfobject action = "create" type = "CORBA" context = "IOR"
  class = "d:\temp\tester.ior" name = "handle" locale = "Visibroker">
```

When you use the `CreateObject` function to invoke this CORBA object, specify the name as the function return variable, and specify the type, class, context, and locale as arguments. For example, the following line creates the same object as the preceding `cfobject` tag:

```
handle = CreateObject("CORBA", "d:\temp\tester.ior", "IOR", "Visibroker")
```

For the complete `cfobject` and `CreateObject` syntax, see *CFML Reference*.

Using a naming service

Currently, ColdFusion can only resolve objects registered in a CORBA 2.3-compliant naming service.

If you use a naming service, make sure that its naming context is identical to the naming context specified in the property file of the Connector configuration in use, as specified in the ColdFusion MX Administrator CORBA Connectors page. The property file must contain the line "SVCnameroot=*name*" where *name* is the naming context to be used. The server implementing the object must bind to this context, and register the appropriate name.

Using CORBA objects in ColdFusion

After you create the object, you can invoke attributes and operations on the object using the syntax described in "Creating and using objects" on page 947. The following sections describe the rules for using CORBA objects in ColdFusion pages. They include information on using methods in ColdFusion, which IDL types you can access from ColdFusion, and the ColdFusion data types that correspond to the supported IDL data types.

- Using CORBA interface methods in ColdFusion
- Using IDL types with ColdFusion variables

Using CORBA interface methods in ColdFusion

When you use the `cfobject` tag or the `CreateObject` function to create a CORBA object, ColdFusion creates a handle to a CORBA interface, which is identified by the `cfobject name` attribute or the `CreateObject` function return variable. For example, the following CFML creates a handle named myHandle:

```
<cfobject action = "create" type = "CORBA" context = "IOR"
  class = "d:\temp\tester.ior" name = "myHandle" locale="visibroker">
<cfset myHandle = CreateObject("CORBA", "d:\temp\tester.ior", "IOR",
  "visibroker")
```

You use the handle name to invoke all of the interface methods, as in the following CFML:

```
<cfset ret=myHandle.method(foo)>
```

The following sections describe how to call CORBA methods correctly in ColdFusion.

Method name case considerations

Method names in IDL are case-sensitive. However, ColdFusion is case-insensitive. Therefore, do not use methods that differ only in case in IDL.

For example, the following IDL method declarations correspond to two different methods:

```
testCall(in string a); // method #1
TestCall(in string a); // method #2
```

However, ColdFusion cannot differentiate between the two methods. If you call either method, you cannot be sure which of the two will be invoked.

Passing parameters by value (in parameters)

CORBA in parameters are always passed by value. When calling a CORBA method with a variable in ColdFusion, specify the variable name without quotation marks, as shown in the following example:

IDL	void method(in string a);
CFML	<cfset foo="my string"> <cfset ret=handle.method(foo)>

Passing variables by reference (out and inout parameters)

CORBA out and inout parameters are always passed by reference. As a result, if the CORBA object modifies the value of the variable that you pass when you invoke the method, your ColdFusion page gets the modified value.

To pass a parameter by reference in ColdFusion, specify the variable name in double-quotation marks in the CORBA method. The following example shows an IDL line that defines a method with a string variable, b, that is passed in and out of the method by reference. It also shows CFML that calls this method.

IDL	void method(in string a, inout string b);
CFML	<cfset foo = "My Initial String"> <cfset ret=handle.method(bar, "foo")> <cfoutput>#foo#</cfoutput>

In this case, the ColdFusion variable foo corresponds to the inout parameter b. When the CFML executes, the following happens:

1. ColdFusion calls the method, passing it the variable by reference.
2. The CORBA method replaces the value passed in, "My Initial String", with some other value. Because the variable was passed by reference, this modifies the value of the ColdFusion variable.
3. The `cfoutput` tag prints the new value of the foo variable.

Using methods with return values

Use CORBA methods that return values as you would any ColdFusion function; for example:

IDL	double method(out double a);
CFML	<cfset foo=3.1415> <cfset ret=handle.method("foo")> <cfoutput>#ret#</cfoutput>

Using IDL types with ColdFusion variables

The following sections describe how ColdFusion supports CORBA data types. They include a table of supported IDL types and information about how ColdFusion converts between CORBA types and ColdFusion data.

IDL support

The following table shows which CORBA IDL types ColdFusion supports, and whether they can be used as parameters or return variables. (NA means not applicable.)

CORBA IDL type	General support	As parameters	As return value
constants	No	No	No
attributes	Yes (for properties)	NA	NA
enum	Yes (as an integer)	Yes	Yes

CORBA IDL type	General support	As parameters	As return value
union	No	No	No
sequence	Yes	Yes	Yes
array	Yes	Yes	Yes
interface	Yes	Yes	Yes
typedef	Yes	NA	NA
struct	Yes	Yes	Yes
module	Yes	NA	NA
exception	Yes	NA	NA
any	No	No	No
boolean	Yes	Yes	Yes
char	Yes	Yes	Yes
wchar	Yes	Yes	Yes
string	Yes	Yes	Yes
wstring	Yes	Yes	Yes
octet	Yes	Yes	Yes
short	Yes	Yes	Yes
long	Yes	Yes	Yes
float	Yes	Yes	Yes
double	Yes	Yes	Yes
unsigned short	Yes	Yes	Yes
unsigned long	Yes	Yes	Yes
longlong	No	No	No
unsigned longlong	No	No	No
void	Yes	NA	Yes

Data type conversion

The following table lists IDL data types and the corresponding ColdFusion data types:

IDL type	ColdFusion type
boolean	Boolean
char	One-character string
wchar	One-character string
string	String
wstring	String

IDL type	ColdFusion type
octet	One-character string
short	Integer
long	Integer
float	Real number
double	Real number
unsigned short	Integer
unsigned long	Integer
void	Not applicable (returned as an empty string)
struct	Structure
enum	Integer, where 0 corresponds to the first enumerator in the enum type
array	Array (must match the array size specified in the IDL)
sequence	Array
interface	An object reference
module	Not supported (cannot dereference by module name)
exception	ColdFusion throws an exception of type `coldfusion.runtime.corba.CorbaUserException`
attribute	Object reference using dot notation

Boolean data considerations

ColdFusion treats any of the following as Boolean values:

True	"yes", "true", or 1
False	"no", "false", or 0

You can use any of these values with CORBA methods that take Boolean parameters, as the following code shows:

IDL

```
module Tester
   {
   interface TManager
   {
      void testBoolean(in boolean a);
      void testOutBoolean(out boolean a);
      void testInoutBoolean(inout boolean a);
      boolean returnBoolean();
   }
}
```

CFML

```
<CFSET handle = CreateObject("CORBA", "d:\temp\tester.ior", "IOR", "")
   >
<cfset ret = handle.testboolean("yes")>
<cfset mybool = True>
<cfset ret = handle.testoutboolean("mybool")>
<cfoutput>#mybool#</cfoutput>

<cfset mybool = 0>
<cfset ret = handle.testinoutboolean("mybool")>
<cfoutput>#mybool#</cfoutput>

<cfset ret = handle.returnboolean()>
<cfoutput>#ret#</cfoutput>
```

Struct data type considerations

For IDL struct types, use ColdFusion structures. You can prevent errors by using the same case for structure key names in ColdFusion as you do for the corresponding IDL struct field names.

Enum type considersations

ColdFusion treats the enum IDL type as an integer with the index starting at 0. As a result, the first enumerator corresponds to 0, the second to 1, and so on. In the following example, the IDL enumerator a corresponds to 0, b to 1 and c to 2:

IDL

```
module Tester
{
   enum EnumType {a, b, c};
   interface TManager
   {
      void testEnum(in EnumType a);
      void testOutEnum(out EnumType a);
      void testInoutEnum(inout EnumType a);
      EnumType returnEnum();
   }
}
```

CFML

```
<CFSET handle = CreateObject("CORBA", "d:\temp\tester.ior", "IOR", "")
   >
<cfset ret = handle.testEnum(1)>
```

In this example, the CORBA object gets called with the second (*not* first) entry in the enumerator, a.

Double-byte character considerations

If you are using an ORB that supports CORBA later than version 2.0, you do not have to do anything to support double-byte characters. Strings and characters in ColdFusion will appropriately convert to wstring and wchar when they are used. However, the CORBA 2.0 IDL specification does not support the wchar and wstring types, and uses the 8-bit Latin-1 character set to represent string data. In this case, you cannot pass parameters containing those characters, however, you can call parameters with char and string types using ColdFusion string data.

Handling exceptions

Use the `cftry` and `cfcatch` tags to catch CORBA object method exceptions thrown by the remote server, as follows:

1. Specify `type="coldfusion.runtime.corba.CorbaUserException"` in the `cfcatch` tag to catch CORBA exceptions.
2. Use the `cfcatch.getContents` method to get the contents of the exception object.

The `cfcatch.getContents` method returns a ColdFusion structure containing the data specified by the IDL for the exception.

The following code example shows the IDL for a CORBA object that raises an exception defined by the PrimitiveException exception type definition, and the CFML that catches the exception and displays the contents of the object.

IDL

```
interface myInterface
{
   exception PrimitiveException
   {
      long l;
      string s;
      float f;
   };
   void testPrimitiveException() raises (PrimitiveException);
}
```

CFML

```
<cftry>
   <cfset ret0 = handle.testPrimitiveException()>
   <cfcatch type=coldfusion.runtime.corba.CorbaUserException>
      <cfset exceptStruct= cfcatch.getContents()>
      <cfdump var ="#exceptStruct#">
   </cfcatch>
</cftry>
```

CORBA example

The following code shows an example of using a LoanAnalyzer CORBA object. This simplified object determines whether an applicant is approved for a loan based on the information that is supplied.

The LoanAnalyzer CORBA interface has one method, which takes the following two in arguments:

- An Account struct that identifies the applicant's account. It includes a Person struct that represents the account holder, and the applicant's age and income.
- A CreditCards sequence, which corresponds to the set of credit cards the user currently has. The credit card type is represented by a member of the CardType enumerator. (This example assumes the applicant has no more than one of any type of card.)

The object returns a Boolean value indicating whether the application is accepted or rejected.

The CFML does the following:

1. Initializes the values of the ColdFusion variables that are used in the object method. In a more complete example, the information would come from a form, query, or both.

   The code for the Person and Account structs is straightforward. The cards variable, which represents the applicant's credit cards, is more complex. The interface IDL uses a sequence of enumerators to represent the cards. ColdFusion represents an IDL sequence as an array, and an enumerator as 0-indexed number indicating the position of the selected item among the items in the enumerator type definition.

   In this case, the applicant has a Master Card, a Visa card, and a Diners card. Because Master Card (MC) is the first entry in the enumerator type definition, it is represented in ColdFusion by the number 0. Visa is the third entry, so it is represented by 2. Diners is the fifth entry, so it is represented by 4. These numbers must be put in an array to represent the sequence, resulting in a three-element, one-dimensional array containing 0, 2, and 4.
2. Instantiates the CORBA object.
3. Calls the approve method of the CORBA object and gets the result in the return variable, ret.

4. Displays the value of the ret variable, Yes or No.

IDL

```
struct Person
{
   long pid;
   string name;
   string middle;
   string last_name;
}

struct Account
{
   Person person;
   short age;
   double income;
}

double loanAmountl
enum cardType {AMEX, VISA, MC, DISCOVER, DINERS};

typedef sequence<cardType> CreditCards;

interface LoanAnalyzer
{
   boolean approve( in Account, in CreditCards);
}
```

CFML

```
<!--- Declare a "person" struct ---->
<cfset p = StructNew()>
<cfif IsStruct(p)>
   <cfset p.pid = 1003232>
   <cfset p.name = "Eduardo">
   <cfset p.middle = "R">
   <cfset p.last_name = "Doe">
</cfif>

<!---- Declare an "Account" struct --->
<cfset a = StructNew()>
<cfif IsStruct(a)>
   <cfset a.person = p>
   <cfset a.age = 34>
   <cfset a.income = 150120.50>
</cfif>

<!----- Declare a "CreditCards" sequence --->
<cfset cards = ArrayNew(1)>
<cfset cards[1] = 0>   <!--- corresponds to Amex --->
<cfset cards[2] = 2>   <!--- corresponds to MC --->
<cfset cards[3] = 4>   <!--- corresponds to Diners --->

<!---- Creating a CORBA handle using the Naming Service---->
<cfset handle = CreateObject("CORBA", "FirstBostonBank/MA/Loans",
"NameService") >

<cfset ret=handle.approve(a, cards)>
<cfoutput>Account approval: #ret#</cfoutput>
```

PART VII
Using External Resources

This part describes how you can use Macromedia ColdFusion MX to access and use the following external services: mail servers, remote HTTP and FTP servers, and files and directories.

The following chapters are included:

Chapter 39: Sending and Receiving E-Mail 973
Chapter 40: Interacting with Remote Servers 991
Chapter 41: Managing Files on the Server 1003
Chapter 42: Using Event Gateways 1019
Chapter 43: Using the Instant Messaging Event Gateways 1047
Chapter 44: Using the SMS Event Gateway 1065
Chapter 45: Creating Custom Event Gateways 1085

CHAPTER 39
Sending and Receiving E-Mail

You can add interactive e-mail features to your ColdFusion applications using the `cfmail` and `cfpop` tags. This complete two-way interface to mail servers makes the Macromedia ColdFusion MX e-mail capability a vital link to your users.

Contents

Using ColdFusion with mail servers 973
Sending e-mail messages 974
Sample uses of the cfmail tag 977
Using the cfmailparam tag 981
Receiving e-mail messages 983
Handling POP mail 985

Using ColdFusion with mail servers

Adding e-mail to your ColdFusion applications lets you respond automatically to user requests. You can use e-mail in your ColdFusion applications in many different ways, including the following:

- Trigger e-mail messages based on users' requests or orders.
- Allow users to request and receive additional information or documents through e-mail.
- Confirm customer information based on order entries or updates.
- Send invoices or reminders, using information pulled from database queries.

ColdFusion offers several ways to integrate e-mail into your applications. To send e-mail, you generally use the Simple Mail Transfer Protocol (SMTP). To receive e-mail, you use the Post Office Protocol (POP) to retrieve e-mail from the mail server. To use e-mail messaging in your ColdFusion applications, you must have access to an SMTP server and/or a valid POP account.

In your ColdFusion application pages, you use the `cfmail` and `cfpop` tags to send and receive e-mail, respectively. The following sections describe how to use the ColdFusion e-mail features and show examples of these tags.

How ColdFusion sends mail

The ColdFusion implementation of SMTP mail uses a spooled architecture. If you select to spool mail on the Mail page in the ColdFusion MX Administrator, when an application page processes a `cfmail` tag, the messages that are generated are not sent immediately. Instead, they are spooled to disk and processed in the background. This architecture has two advantages:

- End users of your application are not required to wait for SMTP processing to complete before a page returns to them. This design is especially useful when a user action causes more than a handful of messages to be sent.
- Messages sent using `cfmail` are delivered reliably, even in the presence of unanticipated events like power outages or server crashes.

You can set how frequently ColdFusion MX checks for spooled mail messages on the Mail page in the ColdFusion MX Administrator. If ColdFusion is extremely busy or has a large existing queue of messages, however, delivery can occur after the spool interval.

Some ColdFusion MX editions have advanced spooling options that let you fine tune how ColdFusion sends mail. For more information, see *Configuring and Administering ColdFusion MX*.

Error logging and undelivered messages

ColdFusion logs all errors that occur during SMTP message processing to the file mail.log in the ColdFusion log directory. The log entries contain the date and time of the error as well as diagnostic information about why the error occurred.

If a message is not delivered because of an error, ColdFusion writes it to this directory:

- In Windows: \CFusionMX7\Mail\Undelivr
- On UNIX: /opt/coldfusionmx7/mail/undelivr

The error log entry that corresponds to the undelivered message contains the name of the file written to the UnDelivr (or undelivr) directory.

Note: To have ColdFusion try to resend a message that it could not deliver, move the message file from the Undelivr directory to the Spool directory.

For more information about the mail logging settings in the ColdFusion MX Administrator, see *Configuring and Administering ColdFusion MX*.

Sending e-mail messages

Before you configure ColdFusion to send e-mail messages, you must have access to an SMTP e-mail server. Also, before you run application pages that refer to the e-mail server, you can configure the ColdFusion MX Administrator to use the SMTP server. If you need to override the ColdFusion MX Administrator SMTP server setting for any messages, you can specify a new mail server in the `server` attribute of the `cfmail` tag.

To configure ColdFusion for e-mail:

1. In the ColdFusion MX Administrator, select Server Settings > Mail.
2. In the Mail Server box, enter the name or IP address of your SMTP mail server.
3. (Optional) Change the Server Port and Connection Timeout default settings.
4. Select the Verify Mail Server Connection check box to make sure ColdFusion can access your mail server.
5. If your mail server does not use port 25, the default, SMTP port, change the Server Port default settings.
6. Depending on your ColdFusion MX edition, the Mail page in the Administrator has additional options that you can use to configure and optimize ColdFusion mail behavior. Select these as appropriate.
7. Click Submit Changes.

   ColdFusion saves the settings. The page displays a message indicating success or failure for connecting to the server.

ColdFusion MX Enterprise edition includes additional mail spooling and delivery features. For more information on these features, and for information on the Administrator's mail settings, see *Configuring and Administering ColdFusion MX*.

Sending SMTP e-mail with the cfmail tag

The `cfmail` tag provides support for sending SMTP e-mail from within ColdFusion applications. The `cfmail` tag is similar to the `cfoutput` tag, except that `cfmail` outputs the generated text as an SMTP mail message rather than to a page. The `cfmail` tag supports all the attributes and commands that you use with `cfoutput`, including `query`. The following table describes basic `cfmail` tag attributes that you might use to send a simple email message. For a complete list of attributes, see the `cfmail` description in *CFML Reference*.

Attribute	Description
subject	The subject of the message.
from	The e-mail address of the sender.
to	The e-mail address of the recipient. Use a comma-delimited list to specify multiple recipients.
cc	(Optional) The e-mail address of a carbon copy recipient. The recipient's address is visible to other recipients. Use a comma-delimited list to specify multiple cc recipients.
bcc	(Optional) The e-mail address of a blind carbon copy recipient. The recipient's address is not visible to other recipients. Use a comma-delimited list to specify multiple bcc recipients.

To send a simple e-mail message:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Sending a simple e-mail</title>
</head>
<body>
<h1>Sample e-mail</h1>
<cfmail
  from="Sender@Company.com"
  to="#URL.email#"
  subject="Sample e-mail from ColdFusion MX">

This is a sample e-mail message to show basic e-mail capability.

</cfmail>
The e-mail was sent.

</body>
</html>
```

2. Save the file as send_mail.cfm in the myapps directory under your *web_root* directory.
3. Open your browser and enter the following URL:

   http://localhost:8500/myapps/send_mail.cfm?email=*myname@mycompany.com*

   (Replace myname@mycompany.com with your e-mail address.)

   The page sends the e-mail message to you, through your SMTP server.

Note: If you do not receive an e-mail message, check whether you have configured ColdFusion to work with your SMTP server; for more information, see "Sending e-mail messages" on page 974.

The `cfmail` tag has many options that let you customize your mail or control how it is sent. Later sections in the chapter describe how to use some of these attributes. For a description of all attributes, including options to wrap mail text at a specified column, specify the mail character encoding, and specify the mail server, user name, and password, see the `cfmail` description in *CFML Reference*.

Sending HTML format e-mail

If you know all the mail recipients use mail applications that are capable of reading and interpreting HTML code in a mail message, you can use the `cfmail` tag to send an HTML message. The `cfmail` tag `type="HTML"` attribute informs the receiving e-mail client that the message contains embedded HTML tags that must be processed. For an example that sends HTML mail, see "Including images in a message" on page 982.

Sending multipart mail messages

The `cfmailpart` tag lets you create multipart mail messages, with each part having a different MIME type or character set. For example, if you do not know that all recipients can interpret HTML mail messages, you can send your message as a multipart mail with a text part and an HTML part. To do so use two `cfmailpart` tags, one with the HTML version of the message and one with the plain text message, as shown in the following example. To test this example, replace the `To` attribute value with a valid email address, save and run the page, and check the incoming email at the address you entered.

```
<cfmail from = "peter@domain.com" To = "paul@domain.com"
Subject = "Which version do you see?">
  <cfmailpart
    type="text"
    wraptext="74">
      You are reading this message as plain text, because your mail reader
      does not handle HTML text.
  </cfmailpart>>
  <cfmailpart
    type="html">
      <h3>HTML Mail Message</h3>
      <p>You are reading this message as <strong>HTML</strong>.</p>
      <p>Your mail reader handles HTML text.</p>
  </cfmailpart>
</cfmail>
```

Note: In the HTML version of the message, you must escape any number signs, such as those used to specify colors, by using two # characters; for example, `bgcolor="##C5D9E5"`.

Sample uses of the cfmail tag

An application page containing the `cfmail` tag dynamically generates e-mail messages based on the tag's settings. The following sections show some of the tasks that you can accomplish with `cfmail`:

- Sending a mail message in which the data the user enters in an HTML form determine the recipient and contents
- Using a query to send a mail message to a database-driven list of recipients
- Using a query to send a customized mail message, such as a billing statement, to a list of recipients that is dynamically populated from a database

Later sections in this chapter show additional ways you can use the `cfmail` tag to send e-mail messages.

Sending form-based e-mail

In the following example, the contents of a customer inquiry form submittal are forwarded to the marketing department. You could also use the same application page to insert the customer inquiry into the database. You include the following code on your form so that it executes when users enter their information and submit the form:

```
<cfmail
  from="#Form.EMailAddress#"
  to="marketing@MyCompany.com,sales@MyCompany.com"
  subject="Customer Inquiry">

A customer inquiry was posted to our web site:

Name: #Form.FirstName# #Form.LastName#
Subject: #Form.Subject#

#Form.InquiryText#
</cfmail>
```

Sending query-based e-mail

In the following example, a query (ProductRequests) retrieves a list of the customers who inquired about a product during the previous seven days. ColdFusion sends the list, with an appropriate header and footer, to the marketing department:

```
<cfmail
  query="ProductRequests"
  from="webmaster@MyCompany.com"
  to="marketing@MyCompany.com"
  subject="Widget status report">

Here is a list of people who have inquired about
MyCompany Widgets during the previous seven days:

<cfoutput>
#ProductRequests.FirstName# #ProductRequests.LastName#
  (#ProductRequests.Company#) - #ProductRequests.EMailAddress#&##013;
</cfoutput>

Regards,
The WebMaster
webmaster@MyCompany.com

</cfmail>
```

Reviewing the code

The following table describes the code:

Code	Description

`<cfoutput>`
`#ProductRequests.FirstName#`
`  #ProductRequests.LastName#`
`  (#ProductRequests.Company#) -`
`  #ProductRequests.EMailAddress#&##013;`
`</cfoutput>`
	Presents a dynamic list embedded within a normal `cfmail` message, repeating for each row in the ProductRequests query. Because the `cfmail` tag specifies a query, the `cfoutput` tag does not use a `query` attribute. The `&##013;` forces a carriage return between output records.

Sending e-mail to multiple recipients

In addition to simply using a comma-delimited list in the `to` attribute of the `cfmail` tag, you can send e-mail to multiple recipients by using the `query` attribute of the `cfmail` tag. The following examples show how you can send the same message to multiple recipients and how you can a customize each message for the recipient.

Sending a simple message to multiple recipients

In the following example, a query (BetaTesters) retrieves a list of people who are beta testing ColdFusion. This query then notifies each beta tester that a new release is available. The contents of the `cfmail` tag body are not dynamic. What is dynamic is the list of e-mail addresses to which the message is sent. Using the variable `#TesterEMail#`, which refers to the TesterEmail column in the Betas table, in the `to` attribute, enables the dynamic list:

```
<cfquery name="BetaTesters" datasource="myDSN">
  SELECT * FROM BETAS
</cfquery>

<cfmail query="BetaTesters"
  from="beta@MyCompany.com"
  to="#BetaTesters.TesterEMail#"
  subject="Widget Beta Four Available">

To all Widget beta testers:

Widget Beta Four is now available
for downloading from the MyCompany site.
The URL for the download is:

http://beta.mycompany.com

Regards,
Widget Technical Support
beta@MyCompany.com

</cfmail>
```

Customizing e-mail for multiple recipients

In the following example, a query (GetCustomers) retrieves the contact information for a list of customers. The query then sends an e-mail to each customer to verify that the contact information is still valid:

```
<cfquery name="GetCustomers" datasource="myDSN">
  SELECT * FROM Customers
</cfquery>

<cfmail query="GetCustomers"
  from="service@MyCompany.com"
  to="#GetCustomers.EMail#"
  subject="Contact Info Verification">

Dear #GetCustomers.FirstName# -

We'd like to verify that our customer
database has the most up-to-date contact
information for your firm. Our current
information is as follows:

Company Name: #GetCustomers.Company#
Contact: #GetCustomers.FirstName# #GetCustomers.LastName#

Address:
  #GetCustomers.Address1#
  #GetCustomers.Address2#
  #GetCustomers.City#, #GetCustomers.State# #GetCustomers.Zip#

Phone: #GetCustomers.Phone#
Fax: #GetCustomers.Fax#
Home Page: #GetCustomers.HomePageURL#

Please let us know if any of the above
information has changed, or if we need to
get in touch with someone else in your
organization regarding this request.

Thanks,
Customer Service
service@MyCompany.com

</cfmail>
```

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfquery name="GetCustomers"`
`  datasource="myDSN">`
`  SELECT * FROM Customers`
`</cfquery>`
	Retrieves all data from the Customers table into a query named GetCustomers.

`<cfmail query="GetCustomers"`
`  from="service@MyCompany.com"`
`  to="#GetCustomers.EMail#"`
`  subject="Contact Info`
`  Verification">`
	Uses the `to` attribute of `cfmail`, the `#GetCustomers.Email#` query column causes one message to be sent to the address listed in each row of the query. Therefore, the mail body does not use a `cfoutput` tag.

`Dear #GetCustomers.FirstName#`
`...`
`Company Name: #GetCustomers.Company#`
`Contact: #GetCustomers.FirstName#`
`  #GetCustomers.LastName#`
 	
`Address:`
`  #GetCustomers.Address1#`
`  #GetCustomers.Address2#`
`  #GetCustomers.City#,`
`  #GetCustomers.State#`
`  #GetCustomers.Zip#`
 	
`Phone: #GetCustomers.Phone#`
`Fax: #GetCustomers.Fax#`
`Home Page: #GetCustomers.HomePageURL#`
	Uses other query columns (`#GetCustomers.FirstName#`, `#GetCustomers.LastName#`, and so on) within the `cfmail` section to customize the contents of the message for each recipient.

Using the cfmailparam tag

You use the `cfmailparam` tag to include files in your message or add a custom header to an e-mail message. You can send files as attachments or display them inline in the message. You nest the `cfmailparam` tag within the `cfmail` tag.

Attaching files to a message

You can use one `cfmailparam` tag for each attachment, as the following example shows:

```
<cfmail from="daniel@MyCompany.com"
  to="jacob@YourCompany.com"
  subject="Requested Files">

Jake,

Here are the files you requested.

Regards,
Dan

<cfmailparam file="c:\widget_launch\photo_01.jpg">
<cfmailparam file="c:\widget_launch\press_release.doc">

</cfmail>
```

You must use a fully qualified system path for the `file` attribute of `cfmailparam`. The file must be located on a drive on the ColdFusion server machine (or a location on the local network), not the browser machine.

Including images in a message

You can use the `cfmailparam` to include images from other files in an HTML message, as follows:

1. Put a `cfmailparam` tag for each image following the `cfmail` start tag.
2. In each `cfmailparam` tag, do the following
   - Set the `file` attribute to the location of the image.
   - Specify `disposition="inline"`
   - Set the `contentID` attribute to a unique identifier; for example, myImage1.
3. In the location in your HTML where you want the message included, use an `img` tag such as the following:

   ```
   <img src="cid:myImage1">
   ```

The following example shows a simple mail message with an inline image. In this case, the image is located between paragraphs, but you could include it directly inline with the text. To test this example, replace the `cfmail to` parameter with a valid email address and change the `file` parameter to the path to a valid image.

```
<cfmail type="HTML"
     to = "Peter@myCo.com"
     from = "Paul@AnotherCo.com"
     subject = "Sample inline image">
  <cfmailparam file="C:\Inetpub\wwwroot\web.gif"
     disposition="inline"
     contentID="image1">
  <P>There should be an image here</p>
  <img src="cid:image1">
  <p> This text follows the picture</p>
</cfmail>
```

Adding a custom header to a message

When the recipient of an e-mail message replies to the message, the reply is sent, by default, to the address specified in the From field of the original message. You can use `cfmailparam` to provide a Reply-To e-mail address that tells the mail client to override the value in the From field. Using `cfmailparam`, the reply to the following example is addressed to widget_master@YourCompany.com:

```
<cfmail from="jacob@YourCompany.com"
  to="daniel@MyCompany.com"
  subject="Requested Files">
<cfmailparam name="Reply-To" value="widget_master@YourCompany.com">

Dan,
Thanks very much for the sending the widget press release and graphic.
I'm now the company's Widget Master and am accepting e-mail at
```

```
widget_master@YourCompany.com.

See you at Widget World 2002!

Jake
</cfmail>
```

Note: You can combine the two uses of `cfmailparam` within the same ColdFusion page. Write a separate `cfmailparam` tag for each header and for each attached file.

Receiving e-mail messages

You create ColdFusion pages to access a Post Office Protocol (POP) server to retrieve e-mail message information. ColdFusion can then display the messages (or just header information), write information to a database, or perform other actions.

The `cfpop` tag lets you add Internet mail client features and e-mail consolidation to applications. Although a conventional mail client provides an adequate interface for personal mail, there are many cases in which an alternative interface to some mailboxes is advantageous. You use `cfpop` to develop targeted mail clients to suit the specific needs of a wide range of applications. The `cfpop` tag does not work with the other major e-mail protocol, Internet Mail Access Protocol (IMAP).

Here are three instances in which implementing POP mail makes sense:

- If your site has generic mailboxes that are read by more than one person (*sales@yourcompany.com*), it can be more efficient to construct a ColdFusion mail front end to supplement individual user mail clients.
- In many applications, you can automate mail processing when the mail is formatted to serve a particular purpose; for example, when subscribing to a list server.
- If you want to save e-mail messages to a database.

Using `cfpop` with your POP server is like running a query on your mailbox contents. You set its `action` attribute to retrieve either headers (using the `GetHeaderOnly` value) or entire messages (using the `GetAll` value) and assign it a `name` value. You use the name to refer to the record set that `cfpop` returns, for example, when using `cfoutput`. To access a POP server, you also must define the `server`, `username`, and `password` attributes.

Note: If the `cfpop` tag encounters an error, such as an improperly formatted email message, when retrieving messages, it tries to ignore the error; it returns empty fields in the result structure and retrieves any available messages.

For more information on `cfpop` syntax and variables, see *CFML Reference*.

Using the cfpop tag

Use the following steps to add POP mail to your application.

To implement the cfpop tag in your application:

1. Choose the mailboxes to access within your ColdFusion application.
2. Determine which mail message components you must process: message header, message body, attachments, and so on.
3. Decide whether you must store the retrieved messages in a database.
4. Decide whether you must delete messages from the POP server after you retrieve them.
5. Incorporate the `cfpop` tag in your application and create a user interface for accessing a mailbox.
6. Build an application page to handle the output. Retrieved messages can include characters that do not display properly in the browser.

You use the `cfoutput` tag with the `HTMLCodeFormat` and `HTMLEditFormat` functions to control output to the browser. These functions convert characters with special meanings in HTML, such as the less than (<), greater than (>), and ampersand (&) symbols, into HTML-escaped characters, such as <, >, and &. The `HTMLCodeFormat` tag also surrounds the text in a `pre` tag block. The examples in this chapter use these functions.

The cfpop query variables

Like any ColdFusion query, each `cfpop` query returns variables that provide information about the record:

RecordCount The total number of records returned by the query.

ColumnList A list of the headings of the columns that are returned by the query

CurrentRow The current row of the query being processed by `cfoutput` or `cfloop` in a query-driven loop.

The query includes one variable that is not returned by the `cfquery` tag: the UID variable contains the unique identifier of the e-mail message file.

You can reference these properties in a `cfoutput` tag by prefixing the query variable with the query name in the `name` attribute of `cfpop`:

```
<cfoutput>
This operation returned #Sample.RecordCount# messages.
</cfoutput>
```

Handling POP mail

This section describes how to specify the message or messages that you want to get (or get information about) and provides an example of each of the following uses of POP mail:

- Retrieving message headers
- Retrieving messages
- Retrieving messages and attachments
- Deleting messages

Specifying the message or messages

For all `cfpop` actions, you can tel the tag to perform the action on all messages, or to do it on selected messages. To operate on all messages, for example to get all message headers, do not specify a `messageNumber` or `UID` attribute. To operate on specific messages, for example, to delete three selected messages, specify a `messageNumber` or `UID` attribute with a comma-delimited list of messages to act on.

Retrieving message headers

To retrieve message headers without getting the messages, specify `action="GetHeaderOnly"` in the `cfpop` tag. Whether you use `cfpop` to retrieve the header or the entire message, ColdFusion returns a query object that contains one row for each message in the specified mailbox. you specify the query object name in the `cfpop` tag `name` attribute. The query has the following fields:

- date
- from
- header (A string with all the mail header fields, including those that have separate fields in the query object)
- messageNumber (The sequential number of the message in the POP server; identical to the row number of the entry in the query object)
- messageID (The mail header Message-ID field)
- replyTo
- subject
- cc
- to
- UID (The mail header X-UID field)

The `cfpop` tag with the `getHeaderOnly` attribute retrieves any file attachments if you specify an `attachmentPath` attribute; otherwise, it does not get the attachments, and the attachmentfiles column contains empty strings.

To retrieve only the message header:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>POP Mail Message Header Example</title>
</head>

<body>
<h2>This example retrieves message header information:</h2>

<cfpop server="mail.company.com"
  username=#myusername#
  password=#mypassword#
  action="GetHeaderOnly"
  name="Sample">

<cfoutput query="Sample">
  MessageNumber: #HTMLEditFormat(Sample.messageNumber)# <br>
  To: #HTMLEditFormat(Sample.to)# <br>
  From: #HTMLEditFormat(Sample.from)# <br>
  Subject: #HTMLEditFormat(Sample.subject)# <br>
  Date: #HTMLEditFormat(Sample.date)#<br>
  Cc: #HTMLEditFormat(Sample.cc)# <br>
  ReplyTo: #HTMLEditFormat(Sample.replyTo)# <br><br>
</cfoutput>

</body>
</html>
```

2. Edit the following lines so that they refer to valid values for your POP mail server, username, and password:

```
<cfpop server="mail.company.com"
  username=#myusername#
  password=#mypassword#
```

3. Save the file as header_only.cfm in the myapps directory under your *web_root* and view it in your web browser:

   This code retrieves the message headers and stores them in a `cfpop` record set called Sample. For more information about working with record set data, see Chapter 22, "Using Query of Queries," on page 489.

The ColdFusion function `HTMLEditFormat` replaces characters that have meaning in HTML, such as the less than (<) and greater than (>) signs that can surround detailed e-mail address information, with escaped characters such as `<` and `>`.

In addition, you can process the date returned by `cfpop` with `ParseDateTime`, which accepts an argument for converting POP date/time objects into a CFML date-time object.

You can reference any of these columns in a `cfoutput` tag, as the following example shows:

```
<cfoutput>
  #ParseDateTime(queryname.date, "POP")#
  #HTMLCodeFormat(queryname.from)#
```

```
  #HTMLCodeFormat(queryname.messageNumber)#
</cfoutput>
```

For information on these ColdFusion functions, see *CFML Reference*.

Retrieving messages

When you use the `cfpop` tag with `action="GetAll"`, ColdFusion returns the same columns as with `getheaderonly`, plus the following additional columns:

- attachments (A tab-delimited list of attachment file names)
- attachmentfiles (A tab-delimited list of paths to the attachment files retrieved to the local server, if any. You get the files only if you specify an `attachmentpath` attribute.)
- body
- htmlbody
- textbody

If the message is multipart, the htmlbody and textbody fields contain the contents of the HTML and plain text parts, and the body field has the first part in the message. If the message has only one part, the body contains the message, and either the htmlbody or textbody field, depending on the message type, also has a copy of the message.

To retrieve entire messages:

1. Create a ColdFusion page with the following content:

```
<html>
<head><title>POP Mail Message Body Example</title></head>

<body>
<h2>This example adds retrieval of the message body:</h2>
<cfpop server="mail.company.com"
  username=#myusername#
  password=#mypassword#
  action="GetAll"
  name="Sample">

<cfoutput query="Sample">
  MessageNumber: #HTMLEditFormat(Sample.messageNumber)# <br>
  To: #Sample.to# <br>
  From: #HTMLEditFormat(Sample.from)# <br>
  Subject: #HTMLEditFormat(Sample.subject)# <br>
  Date: #HTMLEditFormat(Sample.date)#<br>
  Cc: #HTMLEditFormat(Sample.cc)# <br>
  ReplyTo: #HTMLEditFormat(Sample.replyTo)# <br>
  <br>
  Body:<br>
  #Sample.body#<br>
  <br>
  Header:<br>
  #HTMLCodeFormat(Sample.header)#<br>
  <hr>
  </cfoutput>
```

```
  </body>
</html>
```

2. Edit the following lines so that they refer to valid values for your POP mail server, username, and password:

```
<cfpop server="mail.company.com"
  username=#myusername#
  password=#mypassword#
```

3. Save the file as header_body.cfm in the myapps directory under your *web_root* and view it in your web browser:

This example does not use a CFML function to encode the body contents. As a result, the browser displays the formatted message as you would normally see it in a mail program that supports HTML messages.

Retrieving messages and attachments

When you use the `cfpop` tag with an `attachmentpath` attribute to specify the directory in which to store attachments, ColdFusion retrieves any attachment files from the POP server and saves them in the specified directory. The `cfpop` tag fills the attachmentfiles field with a tab-separated list of the locations of the attachment files. Use the `cffile` tag to delete these temporary files when they are no longer needed. ColdFusion MX creates the directory if it does not exist. (ColdFusion MX must have the appropriate rights on the system to create the directory.)

If a message has no attachments, the `attachments` and `attachmentfiles` columns contain empty strings.

Note: CFML does not provide a way to change the name of a mail attachment returned by `cfpop` before it tries to save the file. If the attachment name is invalid for the file system on which ColdFusion is running, the attachment cannot be saved.

To retrieve all parts of a message, including attachments:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>POP Mail Message Attachment Example</title>
</head>

<body>
<h2>This example retrieves message header,
body, and all attachments:</h2>

<cfpop server="mail.company.com"
  username=#myusername#
  password=#mypassword#
  action="GetAll"
  attachmentpath="c:\temp\attachments"
  name="Sample">

<cfoutput query="Sample">
```

```
    MessageNumber: #HTMLEditFormat(Sample.MessageNumber)# <br>
    To: #HTMLEditFormat(Sample.to)# <br>
    From: #HTMLEditFormat(Sample.from)# <br>
    Subject: #HTMLEditFormat(Sample.subject)# <br>
    Date: #HTMLEditFormat(Sample.date)# <br>
    Cc: #HTMLEditFormat(Sample.cc)# <br>
    ReplyTo: #HTMLEditFormat(Sample.ReplyTo)# <br>
    Attachments: #HTMLEditFormat(Sample.Attachments)# <br>
    Attachment Files: #HTMLEditFormat(Sample.AttachmentFiles)# <br>
    <br>
    Body:<br>
    #Sample.body# <br>
    <br>
    Header:<br>
    HTMLCodeFormat(Sample.header)# <br>
    <hr>
  </cfoutput>

  </body>
  </html>
```

2. Edit the following lines so that they refer to valid values for your POP mail server, username, and password:

```
<cfpop server="mail.company.com"
  username=#myusername#
  password=#mypassword#
```

3. Save the file as header_body_att.cfm in the myapps directory under your *web_root* and view it in your web browser:

Note: To avoid duplicate filenames when saving attachments, set the `generateUniqueFilenames` attribute of `cfpop` to `Yes`.

Deleting messages

Using the `cfpop` tag to delete a message permanently removes it from the server. By default, retrieved messages remain on the POP mail server. To delete the messages, set the `action` attribute of the `cfpop` tag to Delete. Specify use the `messagenumber` attribute to specify the messages to delete; omit the attribute to delete all the user's messages from the server.

Note: Message numbers are reassigned at the end of every POP mail server communication that contains a delete action. For example, if you retrieve four messages from a POP mail server, the server returns the message numbers 1,2,3,4. If you delete messages 1 and 2 with a single `cfpop` tag, messages 3 and 4 are assigned message numbers 1 and 2, respectively.

To delete messages:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>POP Mail Message Delete Example</title>
</head>

<body>
<h2>This example deletes messages:</h2>

<cfpop server="mail.company.com"
  username=#username#
  password=#password#
  action="Delete"
  messagenumber="1,2,3">

</body>
</html>
```

2. Edit the following lines so that they refer to valid values for your POP mail server, username, and password:

```
<cfpop server="mail.company.com"
  username=#username#
  password=#password#
```

3. Save the file as message_delete.cfm in the myapps directory under your *web_root* and view the file in your web browser.
4. Run the header_only.cfm page that you created to confirm that the messages have been deleted.

   Caution: When you view this page in your web browser, ColdFusion immediately deletes the messages from the POP server.

CHAPTER 40
Interacting with Remote Servers

This chapter describes how Macromedia ColdFusion MX wraps the complexity of Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP) communications in a simplified tag syntax that lets you extend your site's offerings across the web.

Contents

About interacting with remote servers 991

Using cfhttp to interact with the web 992

Creating a query object from a text file 994

Using the cfhttp Post method. 996

Performing file operations with cfftp 999

About interacting with remote servers

Transfer protocols are mechanisms for moving files and information from a source to one or more destinations. Two of the more popular protocols are the Hypertext Transfer Protocol (HTTP) and the File Transfer Protocol (FTP). ColdFusion has the `cfhttp` and `cfftp` tags that let you use these protocols to interact with remote servers.

The `cfhttp` tag lets you receive a web page or web-based file, just as a web browser uses HTTP to transport web pages. When you type a URL into a web browser, you make an HTTP request to a web server. With the `cfhttp` tag, you can display a web page, send variables to a ColdFusion or CGI application, retrieve specialized content from a web page, and create a ColdFusion query from a text file. You can use the Get or Post methods to interact with remote servers.

The `cfftp` tag takes advantage of FTP's main purpose—transporting files. Unlike HTTP, FTP was not designed to interact with other servers for processing and interacting with data. After you establish an FTP connection with the `cfftp` tag, you can use it to upload, download, and manage files and directories.

Using cfhttp to interact with the web

The `cfhttp` tag, which lets you retrieve information from a remote server, is one of the more powerful tags in the CFML tag set. You can use one of two methods—Get or Post—to interact with a remote server using the `cfhttp` tag:

- Using the Get method, you can only send information to the remote server in the URL. This method is often used for a one-way transaction in which `cfhttp` retrieves an object.
- Using the Post method, you can pass variables to a ColdFusion page or CGI program, which processes them and returns data to the calling page. The calling page then appears or further processes the data that was received. For example, when you use `cfhttp` to Post to another ColdFusion page, that page does not appear. It processes the request and returns the results to the original ColdFusion page, which then uses the information as appropriate.

Using the cfhttp Get method

You use Get to retrieve files, including text and binary files, from a specified server. The retrieved information is stored in a special variable, `cfhttp.fileContent`. The following examples show several common Get operations.

To retrieve a file and store it in a variable:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Use Get Method</title>
</head>
<body>
<cfhttp
  method="Get"
  url="http://www.macromedia.com"
  resolveurl="Yes">
<cfoutput>
  #cfhttp.FileContent# <br>
</cfoutput>

</body>
</html>
```

2. (Optional) Replace the value of the `url` attribute with another URL.
3. Save the file as get_webpage.cfm in the myapps directory under your *web_root* and view it in the web browser.

   The browser loads the web page specified in the `url` attribute.

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfhttp method="Get"`
`  url="http://www.macromedia.com"`
`  resolveurl="Yes">`
	Get the page specified in the URL and make the links absolute instead of relative so that they appear properly.

`<cfoutput>` `  #cfhttp.FileContent#  ` `</cfoutput>`	Display the page, which is stored in the variable `cfhttp.fileContent`, in the browser.

To get a web page and save it in a file:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Use Get Method</title>
</head>
<body>

<cfhttp
  method = "Get"
  url="http://www.macromedia.com/software"
  path="c:\temp"
  file="macr_software.htm">
</body>
</html>
```

2. (Optional) Replace the value of the `url` attribute with another URL and change the filename.
3. (Optional) Change the path from C:\temp to a path on your hard drive.
4. Save the page as save_webpage.cfm in the myapps directory under your *web_root* directory.
5. Go to the specified path and view the file that you specified in a text editor (using the values specified in step 1, this is C:\temp\macr_software.htm).

   The saved file does not appear properly in your browser because the Get operation saves only the specified web page HTML. It does not save the frame, image, or other files that the page might include.

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfhttp`
`method = "Get"`
`url="http://www.macromedia.com/`
`  software"`
`  path="c:\temp"`
`  file="macr_software.htm">`
	Get the page specified in the URL and save it in the file specified by the `path` and `file` attributes.
	When you use the `path` and `file` attributes, ColdFusion ignores any `resolveurl` attribute. As a result, frames and other included files cannot appear when you view the saved page.

To get a binary file and save it:

1. Create a ColdFusion page with the following content:

```
<cfhttp
  method="Get"
  url="http://www.macromedia.com/macromedia/accessibility/images/
  spotlight.jpg"
  path="c:\temp"
  file="My_SavedBinary.jpg">
<cfoutput>
  #cfhttp.MimeType#
</cfoutput>
```

2. (Optional) Replace the value of the `url` attribute with the URL of a binary file that you want to download.
3. (Optional) Change the path from C:\temp to a path on your hard drive.
4. Save the file as save_binary.cfm in the myapps directory under your *web_root* and open it in the web browser to view the MIME type.
5. (Optional) Verify that the binary file now exists at the location you specified in the `path` attribute.

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfhttp`
`method="Get"`
`url="http://www.macromedia.com/macromedia/`
`accessibility/images/spotlight.jpg"`
`path="c:\temp"`
`file="My_SavedBinary.jpg">`
	Get a binary file and save it in the `path` and `file` specified.

`<cfoutput>`
`#cfhttp.MimeType#`
`</cfoutput>`
	Display the MIME type of the file.

Creating a query object from a text file

You can create a query object from a delimited text file by using the `cfhttp` tag and specifying `method="Get"` and the `name` attribute. This is a powerful method for processing and handling text files. After you create the query object, you can easily reference columns in the query and perform other ColdFusion operations on the data.

ColdFusion processes text files in the following manner:

- You can specify a field delimiter with the `delimiter` attribute. The default is a comma.
- If data in a field might include the delimiter character, you must surround the entire field with the text qualifier character, which you can specify with the `textqualifier` attribute. The default text qualifier is the double-quotation mark (").
- The `textqualifier=""` specifies that there is no text qualifier. If you use `textqualifier="""""` (four " marks in a row), it explicitly specifies the double-quotation mark as the text qualifier.

- If there is a text qualifier, you must surround all field values with the text qualifier character.
- To include the text qualifier character in a field, use a double character. For example, if the text qualifier is ", use "" to include a quotation mark in the field.
- The first row of text is always interpreted as column headings, so that row is skipped. You can override the file's column heading names by specifying a different set of names in the `columns` attribute. You must specify a name for each column. You then use these new names in your CFML code. However, ColdFusion never treats the first row of the file as data.
- When duplicate column heading names are encountered, ColdFusion adds an underscore character to the duplicate column name to make it unique. For example, if two CustomerID columns are found, the second is renamed "CustomerID_".

To create a query from a text file:

1. Create a text file with the following content:

```
OrderID,OrderNum,OrderDate,ShipDate,ShipName,ShipAddress
001,001,01/01/01,01/11/01,Mr. Shipper,123 Main Street
002,002,01/01/01,01/28/01,Shipper Skipper,128 Maine Street
```

2. Save the file as text.txt in the myapps directory under your *web_root*.
3. Create a ColdFusion page with the following content:

```
<cfhttp method="Get"
  url="http://127.0.0.1/myapps/text.txt"
  name="juneorders"
  textqualifier="">

<cfoutput query="juneorders">
  OrderID: #OrderID#<br>
  Order Number: #OrderNum#<br>
  Order Date: #OrderDate#<br>
</cfoutput>
<!--- Now substitute different column names --->
<!--- by using the columns attribute --->
<hr>
Now using replacement column names<br>

<cfhttp method="Get"
  url="http://127.0.0.1/myapps/text.txt"
  name="juneorders"
  columns="ID,Number,ODate,SDate,Name,Address"
  textqualifier="">

<cfoutput query="juneorders">
  Order ID: #ID#<br>
  Order Number: #Number#<br>
  Order Date: #SDate#<br>
</cfoutput>
```

4. Save the file as query_textfile.cfm in the myapps directory under your *web_root* and view it in the web browser.

Using the cfhttp Post method

Use the Post method to send cookie, form field, CGI, URL, and file variables to a specified ColdFusion page or CGI program for processing. For Post operations, you must use the `cfhttpparam` tag for each variable you want to post. The Post method passes data to a specified ColdFusion page or an executable that interprets the variables being sent and returns data.

For example, when you build an HTML form using the Post method, you specify the name of the page to which form data is passed. You use the Post method in `cfhttp` in a similar way. However, with `cfhttp`, the page that receives the Post does not, itself, display anything.

To pass variables to a ColdFusion page:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>HTTP Post Test</title>
</head>
<body>
<h1>HTTP Post Test</h1>
<cfhttp method="Post"
  url="http://127.0.0.1:8500/myapps/post_test_server.cfm">
  <cfhttpparam type="Cookie"
    value="cookiemonster"
    name="mycookie6">
  <cfhttpparam type="CGI"
    value="cgivar "
    name="mycgi">
  <cfhttpparam type="URL"
    value="theurl"
    name="myurl">
  <cfhttpparam type="Formfield"
    value="twriter@macromedia.com"
    name="emailaddress">
  <cfhttpparam type="File"
    name="myfile"
    file="c:\pix\trees.gif">
</cfhttp>
<cfoutput>
File Content:<br>
  #cfhttp.filecontent#<br>
Mime Type:  #cfhttp.MimeType#<br>
</cfoutput>
</body>
</html>
```

2. Replace the path to the GIF file to a path on your server (just before the closing `cfhttp` tag).
3. Save the file as post_test.cfm in the myapps directory under your *web_root*.

Note: You must write a page to view the variables. This is the next procedure.

Reviewing the code

The following table describes the code and its function:

Code	Description
`<cfhttp method="Post" url="http://127.0.0.1:8500/myapps/ post_test_server.cfm">`	Post an HTTP request to the specified page.
`<cfhttpparam type="Cookie" value="cookiemonster" name="mycookie6">`	Send a cookie in the request.
`<cfhttpparam type="CGI" value="cgivar " name="mycgi">`	Send a CGI variable in the request.
`<cfhttpparam type="URL" value="theurl" name="myurl">`	Send a URL in the request.
`<cfhttpparam type="Formfield" value="twriter@macromedia.com" name="emailaddress">`	Send a Form field in the request.
`<cfhttpparam type="File" name="myfile" file="c"\pix\trees.gif">`	Send a file in the request. The `</cfhttp>` tag ends the http request.
`<cfoutput> File Content:  #cfhttp.filecontent# `	Display the contents of the file that the page that is posted to creates by processing the request. In this example, this is the output from the `cfoutput` tag in server.cfm.
`Mime Type: #cfhttp.MimeType#  </cfoutput>`	Display the MIME type of the created file.

To view the variables:

1. Create a ColdFusion page with the following content:

```
<html>
<head><title>HTTP Post Test</title> </head>
<body>
<h1>HTTP Post Test</h1>
<cffile destination="C:\temp\"
  nameconflict="Overwrite"
  filefield="Form.myfile"
  action="Upload"
  attributes="Normal">
<cfoutput>
  The URL variable is: #URL.myurl# <br>
  The Cookie variable is: #Cookie.mycookie6# <br>
  The CGI variable is: #CGI.mycgi#. <br>
  The Formfield variable is: #Form.emailaddress#. <br>
  The file was uploaded to #File.ServerDirectory#\#File.ServerFile#.
</cfoutput>
</body>
</html>
```

2. Replace C:\temp\ with an appropriate directory path on your hard drive.
3. Save the file as post_test_server.cfm in the myapps directory under your *web_root*.
4. View post_test.cfm in your browser and look for the file in C:\temp\ (or your replacement path).

Reviewing the code

The following table describes the code and its function:

Code	Description
`<cffile destination="C:\temp\" nameconflict="Overwrite" filefield="Form.myfile" action="Upload" attributes="Normal">`	Write the transferred document to a file on the server. You send the file using the `cfhttpparam type="File"` attribute, but the receiving page gets it as a Form variable, not a File variable. This `cffile` tag creates File variables, as follows.
`<cfoutput>`	Output information. The results are not displayed by this page. They are passed back to the posting page in its `cfhttp.filecontent` variable.
`The URL variable is: #URL.myurl#  `	Output the value of the URL variable sent in the HTTP request.
`The Cookie variable is: #Cookie.mycookie#  `	Output the value of the Cookie variable sent in the HTTP request.
`The CGI variable is: #CGI.mycgi#  `	Output the value of the CGI variable sent in the HTTP request.
`The Form variable is: #Form.emailaddress#.  `	Output the Form variable sent in the HTTP request. You send the variable using the `type="formField"` attribute but the receiving page gets it as a Form variable.
`The file was uploaded to #File.ServerDirectory#\#File.ServerFile#. </cfoutput>`	Output the results of the `cffile` tag on this page. This time, the variables really are File variables.

To return results of a CGI program:

The following code runs a CGI program search.exe on a website and displays the results, including both the MIME type and length of the response. The search.exe program must expect a "search" parameter.

```
<cfhttp method="Post"
     url="http://www.my_favorite_site.com/search.exe"
     resolveurl="Yes">
  <cfhttpparam type="Formfield"
     name="search"
     value="Macromedia ColdFusion">
</cfhttp>
```

```
<cfoutput>
  Response Mime Type: #cfhttp.MimeType#<br>
  Response Length: #len(cfhttp.filecontent)# <br>
  Response Content: <br>
    #htmlcodeformat(cfhttp.filecontent)#<br>
</cfoutput>
```

Performing file operations with cfftp

The `cfftp` tag lets you perform tasks on remote servers using File Transfer Protocol (FTP). You can use `cfftp` to cache connections for batch file transfers when uploading or downloading files.

Note: To use `cfftp`, you must select the Enable ColdFusion Security option on the Sandbox Security page in the Security area in the ColdFusion MX Administrator. (In the Standard Edition, select Security › Basic Security.)

For server/browser operations, use the `cffile`, `cfcontent`, and `cfdirectory` tags.

Using `cfftp` involves two major types of operations: connecting, and transferring files. The FTP protocol also provides commands for listing directories and performing other operations. For a complete list of attributes that support FTP operations and additional details on using the `cfftp` tag, see *CFML Reference*.

To open an FTP connection and retrieve a file listing:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>FTP Test</title>
</head>

<body>
<h1>FTP Test</h1>
<!--- Open ftp connection --->
<cfftp connection="Myftp"
  server="MyServer"
  username="MyUserName"
  password="MyPassword"
  action="Open"
  stoponerror="Yes">

<!--- Get the current directory name. --->
<cfftp connection=Myftp
  action="GetCurrentDir"
  stoponerror="Yes">

<!--- output directory name --->
<cfoutput>
  The current directory is:  #cfftp.returnvalue#<p>
</cfoutput>

<!--- Get a listing of the directory. --->
<cfftp connection=Myftp
  action="listdir"
  directory="#cfftp.returnvalue#"
```

```
    name="dirlist"
    stoponerror="Yes">
  <!--- Close the connection.--->
  <cfftp action="close" connection="Myftp">
  <p>Did the connection close successfully?
    <cfoutput>#cfftp.succeeded#</cfoutput></p>

  <!--- output dirlist results --->
  <hr>
  <p>FTP Directory Listing:</p>

  <cftable query="dirlist" colheaders="yes" htmltable>
    <cfcol header="<B>Name</b>" TEXT="#name#">
    <cfcol header="<B>Path</b>" TEXT="#path#">
    <cfcol header="<B>URL</b>" TEXT="#url#">
    <cfcol header="<B>Length</b>" TEXT="#length#">
    <cfcol header="<B>LastModified</b>"
    TEXT="#DateFormat(lastmodified)#">
    <cfcol header="<B>IsDirectory</b>"
      TEXT="#isdirectory#">
  </cftable>
```

2. Change `MyServer` to the name of a server for which you have FTP permission.
3. Change `MyUserName` and `MyPassword` to a valid username and password.

   To establish an anonymous connection, enter "anonymous" as the username and an e-mail address (by convention) for the password.
4. Save the file as ftp_connect.cfm in the myapps directory under your *web_root* and view it in the web browser.

Reviewing the code

The following table describes the code and its function:

Code	Description

`<cfftp connection="Myftp"`
`  server="MyServer"`
`  username="MyUserName"`
`  password="MyPassword"`
`  action="Open"`
`  stoponerror="Yes">`
	Open an FTP connection to the MyServer server and log on as MyUserName. If an error occurs, stop processing and display an error. You can use this connection in other `cfftp` tags by specifying the Myftp connection.

`<cfftp connection=Myftp`
`  action="GetCurrentDir"`
`  stoponerror="Yes">`
`<cfoutput>`
`  The current directory is:`
`  #cfftp.returnvalue#<p>`
`</cfoutput>`
	Use the Myftp connection to get the name of the current directory; stop processing if an error occurs.
	Display the current directory.

Code	Description

`<cfftp connection=Myftp`
`  action="ListDir"`
`  directory="#cfftp.returnvalue#"`
`  name="dirlist"`
`  stoponerror="Yes">`
	Use the Myftp connection to get a directory listing. Use the value returned by the last `cfftp` call (the current directory of the connection) to specify the directory to list. Save the results in a variable named dirlist (a query object). Stop processing if there is an error.

`<cfftp action="close" connection="Myftp">`
`<p>Did the connection close successfully?`
`  <cfoutput>#cfftp.succeeded#</`
`  cfoutput></p>`
	Close the connection, and do not stop processing if the operation fails (because you can still use the results). Instead, display the value of the `cfftp.succeeded` variable, which is `Yes` if the connection is closed, and `No` if the operation failed.

`<cftable query="dirlist"`
`  colheaders="yes" htmltable>`
`  <cfcol header="<B>Name</b>"`
`  TEXT="#name#">`
`  <cfcol header="<B>Path</b>"`
`  TEXT="#path#">`
`  <cfcol header="<B>URL</b>"`
`  TEXT="#url#">`
`  <cfcol header="<B>Length</b>"`
`  TEXT="#length#">`
`  <cfcol header="<B>LastModified</b>"`
`  TEXT="#DateFormat(lastmodified)#">`
`  <cfcol header="<B>IsDirectory</b>"`
`  TEXT="#isdirectory#">`
`</cftable>`
	Display a table with the results of the ListDir FTP command.

After you establish a connection with `cfftp`, you can reuse the connection to perform additional FTP operations until either you or the server closes the connection. When you access an already-active FTP connection, you do not need to re-specify the username, password, or server. In this case, make sure that when you use frames, only one frame uses the connection object.

Note: For a single simple FTP operation, such as GetFile or PutFile, you do not need to establish a connection. Specify all the necessary login information, including the server and any login and password, in the single `cfftp` request.

Caching connections across multiple pages

The FTP connection established by `cfftp` is maintained only in the current page unless you explicitly assign the connection to a variable with Application or Session scope.

Assigning a `cfftp` connection to an application variable could cause problems, since multiple users could access the same connection object at the same time. Creating a session variable for a `cfftp` connection makes more sense, because the connection is available to only one client and does not last past the end of the session.

Example: caching a connection

```
<cflock scope="Session" timeout=10>
<cfftp action="Open"
  username="anonymous"
  password="me@home.com"
  server="ftp.eclipse.com"
  connection="Session.myconnection">
</cflock>
```

In this example, the connection cache remains available to other pages within the current session. You must enable session variables in your application for this approach to work, and you must lock code that uses session variables. For more information on locking, see Chapter 15, "Using Persistent Data and Locking," on page 339.

Note: Changing a connection's characteristics, such the `retrycount` or `timeout` values, might require you to re-establish the connection.

Connection actions and attributes

The following table shows the available `cfftp` actions and the attributes they require when you use a named (that is, cached) connection. If you do not specify an existing connection name, you must specify the `username`, `password`, and `server` attributes.

Action	Attributes	Action	Attributes
Open	none	Rename	existing new
Close	none	Remove	server item
ChangeDir	directory	GetCurrentDir	none
CreateDir	directory	GetCurrentURL	none
ListDir	name directory	ExistsDir	directory
RemoveDir	directory	ExistsFile	remotefile
GetFile	localfile remotefile	Exists	item
PutFile	localfile remotefile		

CHAPTER 41
Managing Files on the Server

The `cffile`, `cfdirectory`, and `cfcontent` tags handle browser and server file management tasks, such as uploading files from a client to the web server, viewing directory information, and changing the content type that is sent to the web browser. To perform server-to-server operations, use the `cfftp` tag, described in "Performing file operations with cfftp" on page 999.

Contents

About file management 1003
Using cffile. 1003
Using cfdirectory 1012
Using cfcontent 1014

About file management

Macromedia ColdFusion MX lets you access and manage the files and directories on your ColdFusion server. The `cffile` tag has several attributes for moving, copying, deleting, and renaming files. You use the `cfdirectory` tag to list, create, delete, and rename directories. The `cfcontent` tag lets you define the MIME (Multipurpose Internet Mail Extensions) content type that returns to the web browser.

Using cffile

You can use the `cffile` tag to work with files on the server in several ways:

- Upload files from a client to the web server using an HTML form
- Move, rename, copy, or delete files on the server
- Read, write, or append to text files on the server

You use the `action` attribute to specify any of the following file actions: `upload`, `move`, `rename`, `copy`, `delete`, `read`, `readBinary`, `write`, and `append`. The required attributes depend on the `action` specified. For example, if `action="write"`, ColdFusion expects the attributes associated with writing a text file.

Note: Consider the security and logical structure of directories on the server before allowing users access to them. You can disable the `cffile` tag in the ColdFusion MX Administrator. Also, to access files that are not located on the local ColdFusion MX system, ColdFusion services must run using an account with permission to access the remote files and directories.

Uploading files

File uploading requires that you create two files:

- An HTML form to specify file upload information
- An action page containing the file upload code

The following procedures describe how to create these files.

To create an HTML file to specify file upload information:

1. Create a ColdFusion page with the following content:

```
<head><title>Specify File to Upload</title></head>
<body>
<h2>Specify File to Upload</h2>
<!--- the action attribute is the name of the action page --->
<form action="uploadfileaction.cfm"
     enctype="multipart/form-data"
     method="post">
  <p>Enter the complete path and filename of the file to upload:
  <input type="file"
       name="FiletoUpload"
       size="45">
  </p>
  <input type="submit"
       value="Upload">
</form>
</body>
```

2. Save the file as uploadfileform.cfm in the myapps directory under your *web_root* and view it in the browser.

Note: The form will not work until you write an action page for it (see the next procedure).

Reviewing the code

The following table describes the code and its function:

Code	Description

`<form action="uploadfileaction.cfm"`
`enctype="multipart/form-data"`
`method="post">`
	Create a form that contains file selection fields for upload by the user. The `action` attribute value specifies the ColdFusion template that will process the submitted form. The `enctype` attribute value tells the server that the form submission contains an uploaded file. The `method` attribute is set to `post` to submit a ColdFusion form.

`<input type="file"`
`name="FiletoUpload"`
`size="45">`
	Allow the user to specify the file to upload. The `file` type instructs the browser to prepare to read and transmit a file from the user's system to your server. It automatically includes a Browse button to let the user look for the file instead of manually entering the entire path and filename.

The user can enter a file path or browse the system and select a file to send.

1. Create a ColdFusion page with the following content:

```
<html>
<head> <title>Upload File</title> </head>
<body>
<h2>Upload File</h2>

<cffile action="upload"
     destination="c:\temp\"
     nameConflict="overwrite"
     fileField="Form.FiletoUpload">

<cfoutput>
You uploaded #cffile.ClientFileName#.#cffile.ClientFileExt#
       successfully to #cffile.ServerDirectory#.
</cfoutput>

</body>
</html>
```

2. Change the following line to point to an appropriate location on your server:

   ```
   destination="c:\temp\"
   ```

   Note: This directory must exist on the server.

3. Save the file as uploadfileaction.cfm in the myapps directory under your *web_root*.
4. View uploadfileform.cfm in the browser, enter a file to upload, and submit the form.

   The file you specified uploads.

Reviewing the code

The following table describes the code and its function:

Code	Description
`<cffile action="upload"`	Output the name and location of the uploaded file on the client machine.
`destination="c:\temp\"`	Specify the destination of the file.
`nameConflict="overwrite"`	If the file already exists, overwrite it.
`fileField="Form.FiletoUpload">`	Specify the name of the file to upload. Do not enclose the variable in number signs.
`You uploaded #cffile.ClientFileName#.#cffile.ClientFileExt# successfully to #cffile.ServerDirectory#.`	Inform the user of the file that was uploaded and its destination. For information on `cffile` scope variables, see "Evaluating the results of a file upload" on page 1008.

Note: This example performs no error checking and does not incorporate any security measures. Before deploying an application that performs file uploads, ensure that you incorporate both error handling and security. For more information, see Chapter 16, "Securing Applications," on page 373 and Chapter 14, "Handling Errors," on page 307.

Resolving conflicting filenames

When you save a file to the server, there is a risk that a file with the same name might already exist. To resolve this problem, assign one of these values to the `nameConflict` attribute of the `cffile` tag:

Error (default) ColdFusion stops processing the page and returns an error. The file is not saved.

Skip Allows custom behavior based on file properties. Neither saves the file nor returns an error.

Overwrite Overwrites a file that has the same name as the uploaded file.

MakeUnique Generates a unique filename for the uploaded file. The name is stored in the file object variables serverFile and serverFileName. You can use this variable to record the name used when the file was saved. The unique name might not resemble the attempted name. For more information on file upload status variables, see "Evaluating the results of a file upload" on page 1008.

Controlling the type of file uploaded

For some applications, you might want to restrict the type of file that is uploaded. For example, you might not want to accept graphic files in a document library.

You use the `accept` attribute to restrict the type of file that you allow in an upload. When an `accept` qualifier is present, the uploaded file's MIME content type must match the criteria specified or an error occurs. The `accept` attribute takes a comma-separated list of MIME data names, optionally with wildcards.

A file's MIME type is determined by the browser. Common types, such as image/gif and text/plain, are registered in the browser.

Note: Current versions of Microsoft Internet Explorer and Netscape support MIME type associations. Other browsers and earlier versions might ignore these associations.

ColdFusion saves any uploaded file if you omit the `accept` attribute or specify "*/*". You can restrict the file types, as demonstrated in the following examples.

The following `cffile` tag saves an image file only if it is in the GIF format:

```
<cffile action="Upload"
  fileField="Form.FiletoUpload"
  destination="c:\uploads\"
  nameConflict="Overwrite"
  accept="image/gif">
```

The following `cffile` tag saves an image file only if it is in GIF or JPEG format:

```
<cffile action="Upload"
  fileField="Form.FiletoUpload"
  destination="c:\uploads\"
  nameConflict="Overwrite"
  accept="image/gif, image/jpeg">
```

Note: If you receive an error similar to "The MIME type of the uploaded file (image/jpeg) was not accepted by the server", enter `accept="image/jpeg"` to accept JPEG files.

This `cffile` tag saves any image file, regardless of the format:

```
<cffile action="Upload"
  fileField="Form.FiletoUpload"
  destination="c:\uploads\"
  nameConflict="Overwrite"
  accept="image/*">
```

Setting file and directory attributes

In Windows, you specify file attributes using the `cffile attributes` attribute. In UNIX, you specify file or directory permissions using the `mode` attribute of the `cffile` or `cfdirectory` tag.

Windows

In Windows, you can set the following file attributes:

- Hidden
- Normal
- ReadOnly

To specify several attributes in CFML, use a comma-separated list for the `attributes` attribute; for example, `attributes="ReadOnly,Hidden"`. If you do not use the `attributes` attribute, the file's existing attributes are maintained. If you specify any other attributes in addition to Normal, the additional attribute overrides the Normal setting.

UNIX

In UNIX, you can individually set permissions on files and directories for each of three types of users—owner, group, and other. You use a number for each user type. This number is the sum of the numbers for the individual permissions allowed. Values for the `mode` attribute correspond to octal values for the UNIX `chmod` command:

- 4 = read
- 2 = write
- 1 = execute

You enter permissions values in the `mode` attribute for each type of user: owner, group, and other in that order. For example, use the following code to assign read permissions for everyone:

```
mode=444
```

To give a file or directory owner read/write/execute permissions and read only permissions for everyone else:

```
mode=744
```

Evaluating the results of a file upload

After a file upload is completed, you can retrieve status information using file upload status variables. This status information includes data about the file, such as its name and the directory where it was saved.

You can access file upload status variables using dot notation, using either file.varname or cffile.varname. Although you can use either the File or cffile prefix for file upload status variables, cffile is preferred; for example, cffile.ClientDirectory. The File prefix is retained for backward compatibility.

Note: File status variables are read-only. They are set to the results of the most recent `cffile` operation. If two `cffile` tags execute, the results of the first are overwritten by the subsequent `cffile` operation.

The following table describes the file upload status variables that are available after an upload:

Variable	Description
attemptedServerFile	Initial name that ColdFusion uses when attempting to save a file; for example, myfile.txt. (see "Resolving conflicting filenames" on page 1006).
clientDirectory	Directory on the client's system from which the file was uploaded.
clientFile	Full name of the source file on the client's system with the file extension; for example, myfile.txt.
clientFileExt	Extension of the source file on the client's system without a period; for example, txt (not .txt).
clientFileName	Name of the source file on the client's system without an extension; for example, myfile.
contentType	MIME content type of the saved file; for example, `image` for `image/gif`.

Variable	Description
contentSubType	MIME content subtype of the saved file; for example, `gif` for `image/gif`.
dateLastAccessed	Date that the uploaded file was last accessed.
fileExisted	Indicates (Yes or No) whether the file already existed with the same path.
fileSize	Size of the uploaded file.
fileWasAppended	Indicates (Yes or No) whether ColdFusion appended the uploaded file to an existing file.
fileWasOverwritten	Indicates (Yes or No) whether ColdFusion overwrote a file.
fileWasRenamed	Indicates (Yes or No) whether the uploaded file was renamed to avoid a name conflict.
fileWasSaved	Indicates (Yes or No) whether ColdFusion saved the uploaded file.
oldFileSize	Size of the file that was overwritten in the file upload operation. Empty if no file was overwritten.
serverDirectory	Directory where the file was saved on the server.
serverFile	Full name of the file saved on the server with the file extension; for example, myfile.txt.
serverFileExt	Extension of the file saved on the server without a period; for example, txt (not .txt).
serverFileName	Name of the file saved on the server without an extension; for example, myfile.
timeCreated	Date and time the uploaded file was created.
timeLastModified	Date and time of the last modification to the uploaded file.

Moving, renaming, copying, and deleting server files

With `cffile`, you can create application pages to manage files on your web server. You can use the tag to move files from one directory to another, rename files, copy a file, or delete a file.

The examples in the following table show static values for many of the attributes. However, the value of all or part of any attribute in a `cffile` tag can be a dynamic parameter.

Action	Example code

Move a file	`<cffile action="move"`
	`source="c:\files\upload\KeyMemo.doc"`
	`destination="c:\files\memo\">`

Rename a file	`<cffile action="rename"`
	`source="c:\files\memo\KeyMemo.doc"`
	`destination="c:\files\memo\OldMemo.doc">`

Copy a file	`<cffile action="copy"`
	`source="c:\files\upload\KeyMemo.doc"`
	`destination="c:\files\backup\">`

Delete a file	`<cffile action="delete"` `file="c:\files\upload\oldfile.txt">`

This example sets the ReadOnly flag bit for the uploaded file:

```
<cffile action="Copy"
  source="c:\files\upload\keymemo.doc"
  destination="c:\files\backup\"
  attributes="ReadOnly">
```

Note: Ensure you include the trailing slash (\) when you specify the destination directory. Otherwise, ColdFusion treats the last element in the pathname as a filename. This only applies to copy actions.

Reading, writing, and appending to a text file

In addition to managing files on the server, you can use `cffile` to read, create, and modify text files. As a result, you can do the following things:

- Create log files. (You can also use `cflog` to create and write to log files.)
- Generate static HTML documents.
- Use text files to store information that can be incorporated into web pages.

Reading a text file

You can use `cffile` to read an existing text file. The file is read into a local variable that you can use anywhere in the application page. For example, you could read a text file and then insert its contents into a database, or you could read a text file and then use one of the string replacement functions to modify the contents.

To read a text file:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Read a Text File</title>
</head>

<body>
Ready to read the file:<br>
<cffile action="read"
  file="C:\inetpub\wwwroot\mine\message.txt"
  variable="Message">

<cfoutput>
  #Message#
</cfoutput>
</body>
</html>
```

2. Replace C:\inetpub\wwwroot\mine\message.txt with the location and name of a text file on the server.
3. Save the file as readtext.cfm in the myapps directory under your *web_root* and view it in the browser.

Writing a text file on the server

You can use `cffile` to write a text file based on dynamic content. For example, you could create static HTML files or log actions in a text file.

To create a form in to capture data for a text file:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Put Information into a Text File</title>
</head>

<body>
<h2>Put Information into a Text File</h2>

<form action="writetextfileaction.cfm" method="Post">
  <p>Enter your name: <input type="text" name="Name" size="25"></p>
  <p>Enter the name of the file: <input type="text" name="FileName"
  size="25">.txt</p>
  <p>Enter your message:
  <textarea name="message"cols=45 rows=6></textarea>
  </p>
  <input type="submit" name="submit" value="Submit">
</form>
</body>
</html>
```

2. Save the file as writetextfileform.cfm in the myapps directory under your *web_root*.

Note: The form will not work until you write an action page for it (see the next procedure).

To write a text file:

1. Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>Write a Text File</title>
</head>
<body>
<cffile action="write"
  file="C:\inetpub\wwwroot\mine\#Form.FileName#.txt"
  output="Created By: #Form.Name#
#Form.Message# ">
</body>
</html>
```

2. Modify the path C:\inetpub\wwwroot\mine\ to point to a path on your server.
3. Save the file as writetextfileaction.cfm in the myapps directory under your *web_root*.
4. View the file writetextfileform.cfm in the browser, enter values, and submit the form.

   The text file is written to the location you specified. If the file already exists, it is replaced.

Appending a text file

You can use `cffile` to append additional text to the end of a text file; for example, when you create log files.

To append a text file:

1. Open the writetextfileaction.cfm file.
2. Change the value for the `action` attribute from `write` to `append` so that the file appears as follows:

```
<html>
<head>
  <title>Append a Text File</title>
</head>
<body>
<cffile action="append"
  file="C:\inetpub\wwwroot\mine\message.txt"
  output="Appended By: #Form.Name#">
</body>
</html>
```

3. Save the file as writetextfileaction.cfm in the myapps directory under your *web_root*.
4. View the file in the browser, enter values, and submit the form.

   The appended information displays at the end of the text file.

Using cfdirectory

Use the `cfdirectory` tag to return file information from a specified directory and to create, delete, and rename directories. When listing directory contents or deleting a directory, you can optionally use the `recurse` attribute to access or delete all subdirectories.

As with `cffile`, you can disable `cfdirectory` processing in the ColdFusion MX Administrator. For details on the syntax of this tag, see *CFML Reference*.

Returning file information

When you use the `action="list"` attribute setting, `cfdirectory` returns a query object as specified in the `name` attribute. The `name` attribute is required when you use the `action="list"` attribute setting. This query object contains result columns that you can reference in a `cfoutput` tag, using the value specified in the `name` attribute:

name Directory entry name.

directory Directory containing the entry.

size Directory entry size.

type File type: File or Dir.

dateLastModified Date an entry was last modified.

attributes (Windows only) File attributes, if applicable.

mode (UNIX only) The octal value representing the permissions setting for the specified directory.

Note: ColdFusion supports the ReadOnly and Hidden values for the `attributes` attribute for `cfdirectory` sorting.

Depending on whether your server is on a UNIX system or a Windows system, either the Attributes column or the Mode column is empty. Also, you can specify a filename in the `filter` attribute to get information on a single file.

The following procedure describes how to create a ColdFusion page in which to view directory information.

To view directory information:

1. Create a ColdFusion page with the following content:

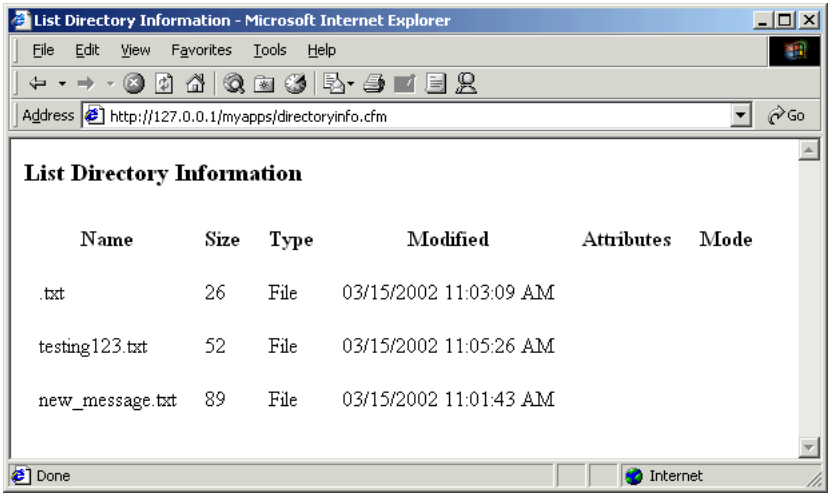
```
<html>
<head>
  <title>List Directory Information</title>
</head>

<body>
<h3>List Directory Information</h3>
<cfdirectory
  directory="c:\inetpub\wwwroot\mine"
  name="mydirectory"
  sort="size ASC, name DESC, datelastmodified">

<table cellspacing=1 cellpadding=10>
<tr>
  <th>Name</th>
  <th>Size</th>
  <th>Type</th>
  <th>Modified</th>
  <th>Attributes</th>
  <th>Mode</th>
</tr>
<cfoutput query="mydirectory">
<tr>
  <td>#mydirectory.name#</td>
  <td>#mydirectory.size#</td>
  <td>#mydirectory.type#</td>
  <td>#mydirectory.dateLastModified#</td>
  <td>#mydirectory.attributes#</td>
  <td>#mydirectory.mode#</td>
</tr>
</cfoutput>
</table>

</body>
</html>
```

2. Modify the path C:\inetpub\wwwroot\mine so that it points to a directory on your server.
3. Save the file as directoryinfo.cfm in the myapps directory under your *web_root* and view it in the browser:



Using cfcontent

The `cfcontent` tag downloads files from the server to the client. You can use this tag to set the MIME type of the content returned by a ColdFusion page and, optionally, define the filename of a file to be downloaded by the current page. By default, ColdFusion returns a MIME content type of text/html so that a web browser renders your template text as a web page.

As with `cffile` and `cfdirectory`, you can disable `cfcontent` processing in the ColdFusion MX Administrator.

About MIME types

A *MIME type* is a label that identifies the contents of a file. the browser uses the MIME type specification to determine how to interact with the file. For example, the browser could open a spreadsheet program when it encounters a file identified by its MIME content type as a spreadsheet file.

A MIME content type consists of `"type/subtype"` format. The following are common MIME content types:

- text/html
- image/gif
- application/pdf

Changing the MIME content type with cfcontent

You use the `cfcontent` tag to change the MIME content type that returns to the browser along with the content generated from your ColdFusion page.

The `cfcontent` tag has one required attribute, `type`, which defines the MIME content type returned by the current page.

To change the MIME content type with cfcontent:

1. Create an HTML page with the following content:

```
<h1>cfcontent_message.htm</h1>

<p>This is a <em>test message</em> written in HTML.</p>
<p>This is the <em>second paragraph</em> of the test message.
As you might expect, it is also written in HTML.</p>
```

2. Save the file as cfcontent_message.htm in the myapps directory under your *web_root*.

   This HTML file will be called by the ColdFusion file that you write in steps 3 through 7.

3. Create a ColdFusion page with the following content:

```
<html>
<head>
<title>cfcontent Example</title>
</head>

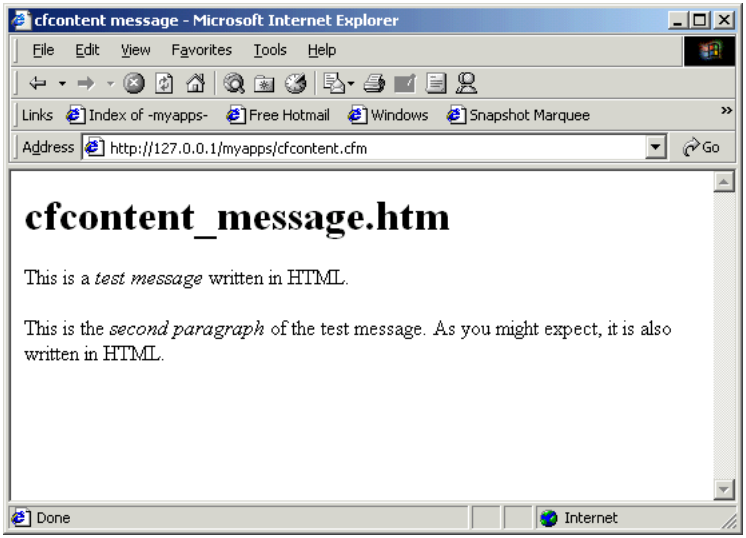
<body>
<h3>cfcontent Example</h3>

<cfcontent
  type = "text/html"
  file = "C:\CFusionMX7\wwwroot\myapps\cfcontent_message.htm"
  deleteFile = "No">
</body>
</html>
```

4. If necessary, edit the `file =` line to point to your myapps directory.

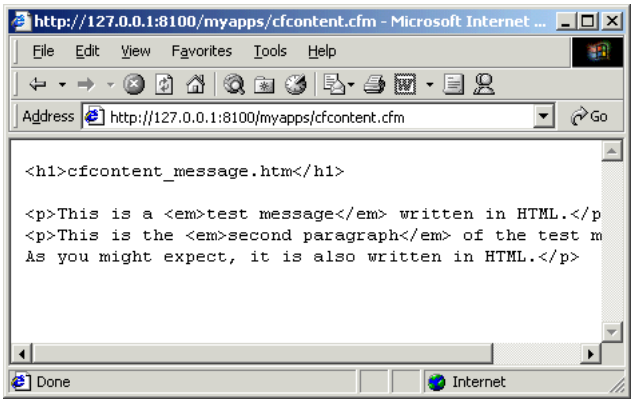
5. Save the file as cfcontent.cfm in the myapps directory under your *web_root* and view it in the browser.

   The text of the called file (cfcontent_message.htm) displays as normal HTML, as shown in the following figure:



6. In cfcontent.cfm, change `type = "text/html"` to `type = "text/plain"`.
7. Save the file and view it in the browser (refresh it if necessary).

   The text displays as unformatted text, in which HTML tags are treated as text:



The following example shows how the cfcontent tag can create an Excel spreadsheet that contains your data.

To create an Excel spreadsheet with cfcontent:

1. Create a ColdFusion page with the following content:

```
<!--- Use cfsetting to block output of HTML
outside of cfoutput tags. --->
<cfsetting enablecfoutputonly="Yes">

<!--- Get employee info. --->
<cfquery name="GetEmps" datasource="cfdocexamples">
  SELECT * FROM Employee
</cfquery>

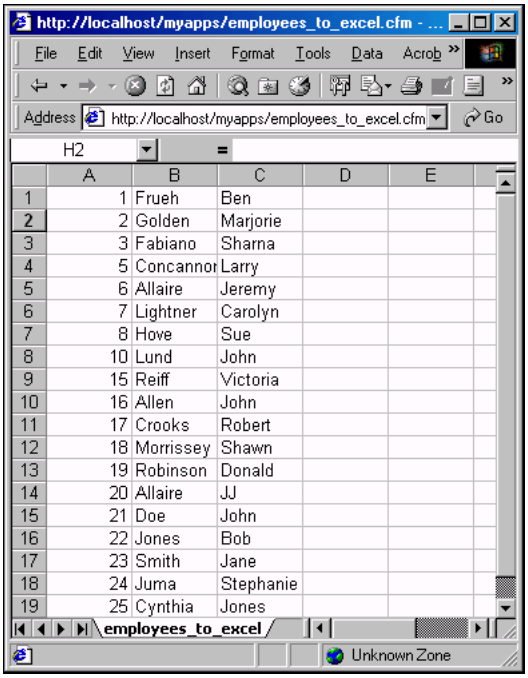
<!--- Set content type. --->
<cfcontent type="application/msexcel">

<!--- Suggest default name for XLS file. --->
<!--- "Content-Disposition" in cfheader also ensures
relatively correct Internet Explorer behavior. --->
<cfheader name="Content-Disposition" value="filename=Employees.xls">

<!--- Format data using cfoutput and a table.
     Excel converts the table to a spreadsheet.
     The cfoutput tags around the table tags force output of the HTML when
     using cfsetting enablecfoutputonly="Yes" --->
<cfoutput>
  <table cols="4">
    <cfloop query="GetEmps">
      <tr>
        <td>#Emp_ID#</td>
        <td>#FirstName#</td>
        <td>#LastName#</td>
      </tr>
    </cfloop>
  </table>
</cfoutput>
```

2. Save the file as employees_to_excel.cfm in the myapps directory under your *web_root* and view it in the browser.

   The data appears in an Excel spreadsheet, as in the following figure:



	A	B	C	D	E
1	1	Frueh	Ben		
2	2	Golden	Marjorie		
3	3	Fabiano	Sharna		
4	5	Concannor	Larry		
5	6	Allaire	Jeremy		
6	7	Lightner	Carolyn		
7	8	Hove	Sue		
8	10	Lund	John		
9	15	Reiff	Victoria		
10	16	Allen	John		
11	17	Crooks	Robert		
12	18	Morrissey	Shawn		
13	19	Robinson	Donald		
14	20	Allaire	JJ		
15	21	Doe	John		
16	22	Jones	Bob		
17	23	Smith	Jane		
18	24	Juma	Stephanie		
19	25	Cynthia	Jones		

CHAPTER 42
Using Event Gateways

This chapter provides an introduction to Macromedia ColdFusion MX event gateways, and provides information on how to write applications that use these gateways. It also describes how to configure an event gateway for an application and how to deploy the application. The final sections of the chapter describe some of the event gateways and example applications that are provided with ColdFusion MX, and how to use them. Subsequent chapters describe how to write applications that use the instant messaging and SMS event gateway types that are provided with ColdFusion MX, and how to develop your own event gateways.

This chapter assumes that developers who write applications that use event gateways have a thorough knowledge of ColdFusion development concepts and practices, including ColdFusion components (CFCs). Developers who are writing applications for custom gateways that are not provided in ColdFusion MX must also know the details of the event gateway they are using and its requirements.

Note: ColdFusion MX Standard Edition does not support event gateways. In ColdFusion MX Developer Edition, some gateway types supplied with ColdFusion MX impose connection limitations.

Contents

About event gateways 1020
Event gateway facilities and tools 1024
Structure of an event gateway application. 1027
Configuring an event gateway instance 1028
Developing an event gateway application 1029
Deploying event gateways and applications 1037
Using the CFML event gateway for asynchronous CFCs 1038
Using the example event gateways and gateway applications 1040

About event gateways

ColdFusion event gateways are ColdFusion MX elements that let ColdFusion MX react to or generate external events or messages in an asynchronous manner. Event gateways let a ColdFusion application handle information that does not come through an HTTP request. For example, you can use event gateways to handle instant messages, short messages from mobile devices, or messages sent to a TCP/IP port.

The event gateway mechanism has the following major features:

- ColdFusion MX event gateways do not require HTTP requests. ColdFusion developers can write ColdFusion MX gateway applications without using any CFM pages (just CFCs).
- ColdFusion CFCs can use event gateways to listen for and respond directly to external events.
- Event gateways operate asynchronously. A gateway typically gets a message and dispatches it for processing, without requiring or waiting for a response.
- ColdFusion developers can create event gateways to handle any kind of event that a Java application can receive.

ColdFusion MX includes several product-level event gateways, such as a gateway for the XMPP (Extensible Messaging and Presence Protocol) instant messaging protocol. Macromedia also provides the source for several example gateways, such as a generalized socket gateway, that you can extend to handle your specific needs. You can also write your own gateways in Java to handle other event or messaging technologies supported by the Java runtime or by third-party providers, such as gateways for additional instant messaging protocols, gateways for specific ERP systems, or other protocols, such as NNTP.

Using event gateways

Because event gateways provide a generalized asynchronous messaging mechanism, you can use them with many kinds of event or messaging resources. For example, ColdFusion MX includes gateways (either product quality, or lighter weight example gateways) for communicating between ColdFusion applications and the following types of resources:

- Mobile phones and other devices that support short messaging services (SMS)
- XMPP or IBM Sametime Instant message clients
- Java Sockets (which let your ColdFusion application communicate with TCP/IP-based devices and programs, such as Telnet terminal clients).
- Java Messaging Service (JMS) resources, such as storefront sales order handling systems.

Event gateways are not limited to sending or receiving information using communications protocols. For example, ColdFusion MX includes an example event gateway that monitors changes to a directory and invokes a CFC method whenever the directory changes. ColdFusion MX also includes an event gateway that lets a CFML application “call” a CFC asynchronously and continue processing without getting a response from the CFC.

Just as you can create event gateways that serve many different event or messaging based technologies, you can write many kinds of applications that use them. Just a few examples of possible gateway uses include the following.

Server to client push examples

- An application that sends an instant message (IM) or SMS text message to a person who can approve a purchase order, get a response, and mark the purchase order as approved or denied.
- A bot that notifies users through their preferred messaging method (cell phone, instant messaging, or even e-mail) when watch list stock goes up, and offers to buy or sell the stock immediately.
- An application that authenticates web users by sending them an SMS message that includes code that they must to enter into the browser in order to proceed.

Client to server examples

- A menu-based SMS application that lets users get information from any of several web service data providers. ColdFusion MX includes a SMS menuing example int the gateways/cfc directory.
- A instant messaging application that takes messages from users to technical support and assigns and directs the messages to the most available support staff member. The application could also log the user ID and session, and you could use ColdFusion MX to generate usage reports.
- A directory lookup robot IM "buddy" that responds to messages chat contain an employee's name with the employee's phone number or buddy ID.

Server to serve examples

- A JMS subsystem that publishes status updates that are consumed by business intelligence systems.
- A system that monitors and publishes download events from a website.

Event gateway terms and concepts

This document uses the following terms when referring to event gateways:

Event A trigger that ColdFusion can receive from an external source. ColdFusion event gateways receive events.

Message The information provided by an event. In ColdFusion MX, a message is the data structure that the event gateway receives when an event is triggered.

Event gateway Java code that receives events and sends them to and from ColdFusion MX application code. This document uses the term *event gateway*, without the word type or instance, to refer to the general concept of a ColdFusion event gateway. Where the context makes the meaning obvious, the term can also refer to event gateway type or event gateway instance.

Event gateway type A specific *event gateway* implementation, represented by a Java class. Each event gateway type handles messages belonging to a particular a communications method or protocol, such as short message service (SMS), an instant messaging protocol, or Sockets. You generally have one event gateway type per communication protocol. You configure each event gateway type on the Gateway Types page in the Event Gateways area in the ColdFusion MX Administrator.

Event gateway instance A specific instance of an *event gateway type* class. You configure each event gateway instance on the ColdFusion MX Gateways page by specifying the event gateway type, an ID, the path to the *event gateway application* CFC that uses this instance, and a configuration file (if needed for the selected event gateway type). You can have multiple event gateway instances per event gateway type, for example, for different event gateway applications.

Event gateway application One or more CFCs and any supporting CFM pages that handle events from an *event gateway instance* and send messages using the event gateway instance. The event gateway application is not part of an event gateway instance, but the code that is responsible for processing event messages to and from the instance.

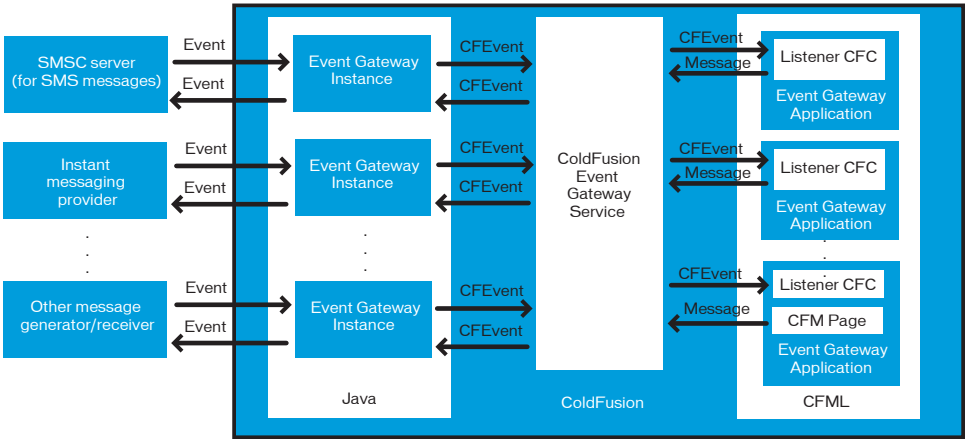
Event gateway listener Code in an *event gateway* that receives *events* from an event source and passes them to the *ColdFusion MX gateway service* for delivery to a CFML *listener CFC*.

Listener CFC A CFC that contains one or more methods that respond to incoming messages from one or more event *gateway instances*. Part of an *event gateway application.*

ColdFusion MX gateway service The part of ColdFusion that provides underlying support for *event gateways*, including a path between an *event gateway instance* and *listener CFCs*.

How event gateway applications work

The following diagram shows the architecture of ColdFusion event gateway applications:



How event gateways interact

Typically, a ColdFusion event gateway instance, a Java object, listens for events coming from an external provider. For example, a general socket event gateway listens for messages on an IP socket, and an SMS event gateway receives messages from an SMSC server.

Each event gateway instance communicates with one or more listener CFCs through the ColdFusion event gateway service. The listener CFCs receive CFEvent object instances that contain the messages, process them, and can send responses back to the event gateway, which can send the messages to the external resources.

Alternatively, a ColdFusion application can initiate a message by calling a ColdFusion function that sends the message to the event gateway. The event gateway then forwards the message to an external resource, such as an instant messaging server. A CFC in the application listens for any responses to the sent message.

Some event gateways can be one-way: they listen for a specific event and send it to a CFC, or they get messages from a ColdFusion function and dispatch it, but they do not do both. The example DirectoryWatcherGateway listens for events only, and the asynchronous CFML event gateway receives messages from CFML only. (You could even say that the directory watcher gateway doesn't listen for events; it creates its own events internally by periodically checking the directory state.)

Event gateway structure

Java programmers develop ColdFusion event gateways by writing Java classes that implement the coldfusion.eventgateway.Gateway interface. ColdFusion event gateways normally consist of one or more threads that listen for events from an event provider, such as a Socket, an SMSC server, or some other source. The event gateway sends event messages to the ColdFusion event gateway service message queue, and provides a method that gets called when an event gateway application CFC or CFM page sends an outgoing message.

The event gateway class can also do the following:

- Provide the ColdFusion application with access to a helper class that provides event gateway-specific services, such as buddy-list management or connection management.
- Use a file that specifies configuration information, such as IP addresses and ports, passwords, and other ID information, internal time-out values, and so on.

About developing event gateway applications

ColdFusion application developers write applications that use event gateways. The person or company that provides the event gateway supplies gateway-specific information to the ColdFusion developer. This information must include the structure and contents of the messages that the ColdFusion application receives and sends to the event gateway, plus any information about configuration files or helper methods that the ColdFusion application might use.

The ColdFusion developer writes a CFC that listens for messages. Many event gateway types send messages to a listener CFC method named `onIncomingMessage`. A minimal event gateway application might implement only this single method. More complex event gateway types can require multiple CFC listener methods. For example, the ColdFusion XMPP IM event gateway sends user messages to the `onIncomingMessage` CFC method, but sends requests to add buddies to the `onAddBuddyRequest` CFC method.

Depending on the event gateway and application types, the event gateway application might include CFM pages or CFC methods to initiate outgoing messages. The application also might use an event gateway-specific GatewayHelper object to do tasks such as getting buddy lists in IM applications or getting a messaging server's status.

The ColdFusion application developer also configures an event gateway instance in the ColdFusion MX Administrator, and possibly in a configuration file. The ColdFusion MX Administrator configuration information specifies the listener CFC that handles the messages from the event gateway and other standard event gateway configuration details. The configuration file, if required, contains event gateway type-specific configuration information.

Event gateway facilities and tools

ColdFusion MX provides a number of features and tools for developing and deploying event-handling applications, these including the following:

- Standard event gateways.
- Development tools and example code.
- A gateway directory structure configured for use by custom event gateways and event gateway applications. This directory also contains the example code.
- An event gateway-specific log file
- Three pages in the ColdFusion MX Administrator for managing event gateways.

Standard event gateways

Macromedia provides several event gateways as part of ColdFusion MX. These event gateways support the following messaging protocols:

- **SMS (Short Message Service)** A system designed for exchanging short, often text, messages with wireless devices, such as mobile phones or pagers. For detailed information on using the SMS event gateway, see Chapter 44, "Using the SMS Event Gateway," on page 1065.
- **XMPP (Extensible Messaging and Presence Protocol)** An open, XML-based protocol for instant messaging. For detailed information on using the XMPP event gateway, see Chapter 43, "Using the Instant Messaging Event Gateways," on page 1047.
- **IBM Lotus Instant Messaging** (commonly referred to as Lotus Sametime) The IBM product for real-time collaboration. For detailed information on using the Lotus Sametime event gateway, see Chapter 43, "Using the Instant Messaging Event Gateways," on page 1047.

ColdFusion MX also provides an event gateway, the CFML asynchronous event gateway, that lets a CFML application invoke a CFC method asynchronously. This event gateway does not follow the model of providing a mechanism for exchanging messages with resources outside of ColdFusion MX. Instead, it provides a one-way path for invoking CFCs when an application does not require (indeed, cannot receive) a return value from the CFC. For detailed information on using the CFML asynchronous event gateway, see "Using the CFML event gateway for asynchronous CFCs" on page 1038.

Development tools and example code

ColdFusion MX provides the following tools and example code for developing your own event gateways and event gateway applications:

- An SMS client (phone simulator) and a short message service center (SMSC) server simulator, for developing SMS applications without requiring an external SMS provider.
- Four sample event gateways with source code:
  - A template for an empty event gateway that contains a skeleton on which you can build your own event gateways
  - A TCP/IP socket event gateway that listens on a TCP/IP port
  - A directory watcher event gateway that monitors changes to the contents of a directory
  - A Java Messaging Service (JMS) gateway that acts as a JMS consumer or producer.
- Several sample applications, including the following:
  - A menu application that uses an inquiry-response drill-down menu to provide services such as weather reports and stock quotes.
  - A simple echo application that sends back the messages that it receives.
  - A temperature converter, an asynchronous logging application.
  - An application that returns employee phone number and other information.

  The chapters in this manual use these example applications.
- Javadoc documentation for the Java interfaces and classes that you use to create gateways.

For more information on these examples, see "Using the example event gateways and gateway applications" on page 1040.

The ColdFusion gateway directory

The ColdFusion MX installation includes a *cf_root*\WEB-INF\cfusion\gateway directory on J2EE configurations, or *cf_root*\gateway directory on server configurations. This directory contains all the code for ColdFusion example event gateways and example event gateway applications, and example configuration files for use by standard ColdFusion event gateways. You do not have to put your event gateways, event gateway application CFCs, or event gateway configuration files in this directory, but ColdFusion is configured to find event gateways and CFCs that you put there.

The following table lists the event gateway directory subdirectories, their purpose, and their initial contents. For more information on using the example event gateways and applications, see "Using the example event gateways and gateway applications" on page 1040.

Directory	Purpose
cfc	Event gateway application CFCs. ColdFusion is installed with an Administrator Mapping between /gateway and this cfc directory.
cfc/examples	Code for the ColdFusion sample applications.

Directory	Purpose
config	Configuration files for all ColdFusion event gateways, including standard ColdFusion MX event gateways, such as SMS, and example event gateways, such as the directory watcher event gateway.
doc/api	Javadoc for the Gateway, and GatewayHelper interfaces, and the CFEvent, GatewayServices, and GenericGateway classes that gateway developer use when writing gateways. This documentation is a subset of the information in the "Gateway development interfaces and classes" section in *CFML Reference*.
lib	Executable code for example and user-developed event gateway classes. The ColdFusion MX class loader includes this directory on its classpath and includes any JAR files that are in that directory on the class path. The examples.jar file in this directory contains the class files for the DirectoryWatcherGateway, EmptyGateway, and SocketGateway classes.
src/examples	Source code for the example event gateway classes that Macromedia provides. Includes the EmptyGateway.java file and the following subdirectories: • socket: Socket gateway source files • watcher: directory watcher gateway source files • JMS: JMS gateway source files

The eventgateway.log file

Event gateways provided with ColdFusion MX log event gateway errors and events to the *cf_root*\WEB-INF\cfusion\logs\eventgateway.log file on J2EE configurations, or the *cf_root*\logs\eventgateway.log file on server configurations. ColdFusion MX includes methods that let any event gateway use this file. This log file can be very useful in debugging event gateways and event gateway applications.

ColdFusion MX Administrator event gateway pages

The ColdFusion MX Administrator includes a Gateways section with three pages for managing event gateways:

- Settings
- Gateway types
- Gateways

The Settings page lets you enable and disable support for event gateways, specify the number of threads that ColdFusion can devote to processing events, specify the maximum number events that ColdFusion can hold in its event queue (which holds events that are waiting to be processed) and start the SMS test server.

The Gateway Types page lets you add, remove, and configure event gateway types by specifying a name, a Java class, and startup timeout behavior.

Note: The gateway type name in the ColdFusion MX Administrator does not have to be the same as the gateway type that is used in the gateway Java code and the `CFEvent` data structure; however, you should use the same name in both places for consistency.

The Gateways page lets you add, remove, configure, start, and stop individual event gateway instances. You configure an event gateway instance by specifying a unique ID, the gateway type, one or more listener CFC paths, a configuration file (not required for all gateway types), and a startup mode (manual, automatic, or disabled).

Structure of an event gateway application

To develop an event gateway application, you create and use some or all of the following elements:

- One or more listener CFCs that handle any incoming messages and send any necessary responses.
- In some applications, ColdFusion pages that generate outgoing messages directly.
- An event gateway instance configuration in the ColdFusion MX Administrator. This configuration might require a separate event gateway configuration file.
- In some applications, a GatewayHelper object to provide access to additional features of the protocol or technology; for example, to manage instant messaging buddy lists.

The role of the listener CFC

All incoming event messages must be handled by one or more listener CFCs. You specify the listener CFCs when you configure an event gateway in the ColdFusion MX Administrator. You must specify at least one CFC in the administrator. Some gateway types can use more than one CFC. By default, the ColdFusion event gateway service delivers events by calling the CFC's `onIncomingMessage` method.

The event gateway developer must inform the event gateway application developer of methods that the listener CFC must implement (may be only the `onIncomingMessage` method) and of the structure and contents of the event message data, contained in the CFEvent instance, that the listener CFC must handle. Outgoing messages have the same event message data structure as incoming messages.

Many gateways let the listener CFCs send a response by calling the `cfreturn` function, but ColdFusion MX does not require a return value. Listener CFCs can also use the `SendGatewayMessage` function, which provides more flexibility than the `cfreturn` tag.

The role of ColdFusion pages

ColdFusion MX CFM pages cannot receive event messages. However, they can send messages using an event gateway. Therefore, an event gateway application that initiates outgoing messages might use one or more `SendGatewayMessage` functions to send the messages. An application that sends an SMS message to notify users when a package ships, for example, could use the `SendGatewayMessage` function to send the notification.

The role of the ColdFusion MX Administrator

The Gateways page in the ColdFusion MX Administrator associates a specific event gateway instance with one or more listener CFCs that processes messages from the event gateway. It tells the ColdFusion event gateway service to send messages received by the event gateway to the listener CFC. It also lets you specify a configuration file for the event gateway instance and whether to start the event gateway instance (and therefore any responder application) when ColdFusion starts. For more information on using the Administrator, see "ColdFusion MX Administrator event gateway pages" on page 1026 and the ColdFusion MX Administrator online Help.

The role of the GatewayHelper object

A ColdFusion MX event gateway provides an information conduit: at its most basic, it receives and dispatches event messages. In some cases, however, an event gateway must provide additional functionality. An instant messaging event gateway, for example, needs to provide such services as managing buddies and providing status information. To support such use, an event gateway can enable access to a GatewayHelper object. The event gateway developer writes a Java class that provides the necessary utility routines as Java methods, and ColdFusion MX application developers can get an instance of the class by calling the CFML `GetGatewayHelper` method. The application calls the GatewayHelper object's methods using normal ColdFusion MX object access techniques. The ColdFusion MX instant messaging event gateways and the example socket event gateway provide GatewayHelper objects.

Configuring an event gateway instance

Before you develop or deploy an event gateway application, you must use the ColdFusion MX Administrator to configure an event gateway instance that will handle the event messages. You specify the following information:

- An event gateway ID to identify the specific event gateway instance. You use this value in the CFML `GetGatewayHelper` and `SendGatewayMessage` functions.
- The event gateway type, which you select from the available event gateway types, such as SMS or Socket.
- The absolute path to the listener CFC or CFCs that will handle incoming messages. If you have multiple listener CFCs, enter the paths separated by commas. You must specify absolute file paths, even if you put the CFCs in the ColdFusion MX gateway\cfc directory.
- A configuration file, if required for this event gateway type or instance.
- The event gateway start-up status; one of the following:

  Automatic Start the event gateway when ColdFusion starts.

  Manual Do not start the event gateway with ColdFusion, but allow starting it from the ColdFusion MX Administrator Event Gateways list.

  Disabled Do not allow the event gateway to start.

Developing an event gateway application

All event gateway applications handle information. They exchange event messages, and possibly other types of information, with other resources. Event gateway applications require a listener CFC to handle events that are sent to the event gateway. Event gateway applications can also use the following code elements:

- `SendGatewayMessage` CFML functions to send messages from outside the listener CFC (or, optionally, from the CFC)
- GatewayHelper objects
- The eventgateway log file

The following sections provide an overview of how an application can send information:

- Event gateway application models
- Sending information to the event gateway
- Developing event gateway listener CFCs
- Sending a message using the SendGatewayMessage function
- Using the GatewayHelper object
- Using the event gateway error log file

Event gateway application models

Event gateway applications follow one or both of the following models:

- **Responder applications** Where event messages from external sources initiate a response from a ColdFusion listener CFC
- **Initiator applications** Where a ColdFusion application generates event messages to send out using the event gateway

Unlike other ColdFusion applications, responder applications are request-free. They do not have CFM pages, just CFCs, and they do not respond to HTTP requests. Instead, ColdFusion the event gateway service deliver the event messages directly to the listener CFC, and the CFC listener method returns any response directly to the event gateway service. Applications that allow mobile phone owners to get a news feed, check for text messages, or request other forms of information follow this model.

Initiator applications are similar to most ColdFusion applications. At some point, ColdFusion executes a CFM page in response to a request. (The request could be initiated by the ColdFusion MX Administrator Scheduled Tasks page.) ColdFusion sends a message to the event gateway when the application calls a CFML `SendGatewayMessage` function. An application that uses SMS to notify customers when orders have been shipped follows this model.

Sending information to the event gateway

A ColdFusion application can send an outgoing message to the event gateway in either of the following ways:

- In a `cfreturn` tag in the listener CFC's listener method
- By calling the ColdFusion `SendGatewayMessage` function

The first method is useful to automatically respond to incoming messages. Some complex applications that respond to incoming messages might use the `SendGatewayMessage` function either in place or in addition to the return value.

Some event gateway types also use a GatewayHelper object to send information to external resources. For example, the ColdFusion MX XMPP and Lotus Sametime instant messaging event gateways provide a GatewayHelper object that can manage buddy lists, and set configuration and status information on the instant messaging server. For more information on the GatewayHelper object, see "Using the GatewayHelper object" on page 1036. For more information on the instant messaging GatewayHelper object, see "Sample IM message handling application" on page 1054.

The example code in "Example event gateway CFC" on page 1034 shows the use of a listener return value, and indicates how event gateways can require different data in the return structure to send equivalent messages.

Developing event gateway listener CFCs

The listener CFC responds to event gateway messages. The listener CFC uses, at a minimum, the following basic software elements:

- One or more listener methods
- CFEvent structures that contain the messages

Listener CFCs can use ColdFusion persistent scopes to store data that needs to be preserved over multiple CFC invocations or shared with other CFML elements.

Listener methods

The ColdFusion event gateway service calls one or more listener methods in the CFC to process incoming messages. The number of listener methods that you must write and their names depends on the event gateway. For example, the ColdFusion SMS event gateway requires a single listener method, which is typically named `onIncomingMessage`. (You can change the SMS event gateway listener method name in the event gateway configuration file.) The ColdFusion XMPP IM event gateway expects the listener CFC to have five methods: `onIncomingMessage`, `onAddBuddyRequest`, `onAddBuddyResponse`, `onBuddyStatus`, and `onIMServerMessage`. By default, if the event gateway does not specify the method name, ColdFusion calls the listener CFC's `onIncomingMessage` method. For the sake of consistency, Macromedia recommends that any event gateway with a single listener method use the `onIncomingMessage` method.

The listener method does the following:

1. Takes a single parameter, a CFEvent structure, described in the following section.
2. Processes the contents of the instance as required by the application.
3. Optionally, returns an outgoing message to the event gateway in a `cfreturn` tag. It can also send a message back to the event gateway by calling the ColdFusion `SendGatewayMessage` function.

The following code shows a listener CFC with an `onIncomingMessage` method that echoes a message back to the Socket event gateway that sent it. It contains the minimum code required to process an incoming message and respond to the sender using the socket gateway.

```
<cfcomponent displayname="echo" hint="echo messages from the event gateway">
  <cffunction name="onIncomingMessage" output="no">
    <cfargument name="CFEvent" type="struct" required="yes">
    <!--- Create a return structure that contains the message. --->
    <cfset retValue = structNew()>
    <cfset retValue.DestinationID = arguments.CFEvent.OriginatorID>
    <cfset retValue.MESSAGE = "Echo: " & arguments.CFEvent.Data.MESSAGE>
    <!--- Send the return message back. --->
    <cfreturn retValue>
  </cffunction>
</cfcomponent>
```

Other event gateways require different fields in the return structure. For example, to echo a message using the SMS event gateway, you use the following lines to specify the return value:

```
<cfset retValue.command = "submit">
<cfset retValue.sourceAddress = arguments.CFEVENT.gatewayid>
<cfset retValue.destAddress = arguments.CFEVENT.originatorid>
<cfset retValue.ShortMessage = "Echo: " & arguments.CFEvent.Data.MESSAGE>
```

The CFEvent structure

The ColdFusion event gateway service passes a CFEvent structure with information about the message event to the listener method. The following table describes the structure's fields:

Field	Description
GatewayID	The event gateway that sent the event; the value is the ID of an event gateway instance configured on the ColdFusion MX Administrator Gateways page. If the application calls the `SendGatewayMessage` function to respond to the event gateway, it uses this ID as the function's first parameter.
Data	A structure containing the event data, including the message. The `Data` structure contents depend on the event gateway type.
OriginatorID	The originator of the message. The value depends on the protocol or event gateway type. Many event gateways require this value in response messages to identify the destination of the response. Identifies the sender of the message.

Field	Description
GatewayType	The type of event gateway, such as SMS. An application that can process messages from multiple event gateway types can use this field. This value is the gateway type name that is specified by the event Gateway class. It is not necessarily the same as the gateway type name in the ColdFusion MX Administrator.
CFCPath	The location of the listener CFC. The listener CFC does not need to use this field.
CFCMethod	The listener method that ColdFusion invokes to process the event. The listener CFC does not need to use this field.
CFCTimeout	The time-out, in seconds, for the listener CFC to process the event request. The listener CFC does not need to use this field.

When a ColdFusion application responds to an event gateway message, or sends a message independently, it does not use a CFEvent structure. However, the ColdFusion event gateway service creates a Java CFEvent instance with the message data before calling the event gateway's `outgoingMessage` method.

Using persistent scopes in listener CFCs

ColdFusion listener CFCs can use the Application, Client, and Session persistent scopes.

Because incoming event gateway messages are not associated with HTTP requests, ColdFusion uses different session and client IDs for interactions initiated by these events than for CFM Page requests, as follows:

Identifier	Structure
Session ID	*gatewayType_gatewayID_originatorID*
cfid	*originatorID*
cftoken	*gatewayType_gatewayID*

The *gatewayID* value is the event gateway ID that you set in the ColdFusion MX Administrator, and *gatewayType* and *originatorID* are the values that the event gateway sets in the CFEvent instance for an incoming message.

Application scope

The Application scope lets the CFC share data with any ColdFusion page or CFC that uses the same application name. This way, a listener CFC can use the same Application scope as CFML pages that might be used to send messages. Also, you can put multiple listener CFCs in a single directory and have them share an Application.cfc or Application.cfm file and application name.

As with all ColdFusion code, use the Application.cfc This.name variable or the `cfapplication` tag to set the application name. The listener CFC can use an Application.cfc or Application.cfm file if the CFC is in a directory that is in or under one of the following places:

- the ColdFusion web root
- a directory that is in the ColdFusion MX Administrator Mappings list.

The ColdFusion MX installer creates a mapping in the ColdFusion MX Administrator for the gateway\cfc directory.

Client scope

The Client scope can store long-term information associated with a message sender's ID. For example, it can store information about an IM buddy.

To use Client variables across multiple connections, your gateway type must use the same client ID for all interactions with a particular client. For many technologies and gateways, such as the IM and SMS gateways, this is not an issue.

Note: To use Client scope variables with gateways, you must store the Client scope variables in a data source or the registry. You cannot store the variables in cookies, because gateways do not use cookies.

Session scope

The Session scope can store information required across multiple interactions. For example, an interactive IM or SMS application that uses a drill-down menu to select a service can store the information about the menu selections in the Session scope.

Event gateway sessions terminate when they time out. Because the identifiers for event sessions and clients differ from those used for request-driven sessions and clients, you cannot use the same Session or Client scope on a standard CFM page that sends an outgoing message and in a listener CFC that might handle an incoming response to that message.

For an example of using the Session scope, see the example Menu application in the gateway\cfc\examples\menu directory.

Note: ColdFusion cannot create a session if an initiator application uses a `SendGatewayMessage` method to start an interaction with a client, such as an SMS user. In this case, the sending code must keep track (for example, in a database) of the messages it sends and their destinations. When a response event arrives, it can look up the origniatorID to determine whether it was in response to an outgoing message.

Debugging event gateway CFCs

When an event gateway CFC responds to an event, it cannot display debugging information in the response page, as CFM pages do. As a result, many of the normal ColdFusion debugging techniques, including the `cfdump` tag, are not available. When you develop event gateway CFCs, you should consider the following debugging techniques:

- Put trace variables in the Application scope. These variables persist, and you can specify an application name for your CFC, as described in "Application scope" on page 1032. You can inspect the Application scope contents, including your trace variables, in any CFML page that has the same application name as your CFC.
- Use `cflog` tags to help you trace any errors by logging significant events to a file. Also, carefully inspect the eventgateway.log and exceptions.log files that ColdFusion MX maintains. For more information on using the eventgateway.log file, see "The eventgateway.log file" on page 1026.

- You can simulate responses from CFCs to the event gateway by using the `SendGatewayMessage` function in a CFM page. The function's *message* parameter should contain the information that the CFC would put in its return variable.
- If you run ColdFusion MX from the command line, you can use the Java `System.out.println` method to write messages to the console window, as the following code shows:

```
<cfscript>
  sys = createObject("java", "java.lang.System");
  sys.out.println("Debugging message goes here");
</cfscript>
```

Note: You do not have to restart the event gateway instance when you make changes to a CFC. ColdFusion MX automatically uses the updated CFC when the next event occurs.

Example event gateway CFC

The following code shows a temperature scale converter tool that can work with any of several event gateways: SMS, XMPP, Lotus Sametime, or the example Socket event gateway. Users enter a string that consists of the temperature scale (F, Fahrenheit, C, or Celsius), a comma, and a temperature on their device. The CFC converts Celsius to Fahrenheit or Fahrenheit to Celsius, and returns the result.

This example shows how a responder event gateway application can work, and illustrates how different event gateway types require different outgoing message formats:

```
<cfcomponent displayname="tempconverter" hint="Convert temperatures between
  Celsius and Fahrenheit">

<cffunction name="onIncomingMessage" output="no">
  <cfargument name="CFEvent" type="struct" required="yes">
  <!--- Standard error message giving the correct input format. --->
  <cfset var errormsg = "Please enter scale, integer where scale is F or C,
    for example:  F, 32">

  <!--- Get the message. --->
  <cfset data=cfevent.DATA>
  <cfset message="#data.message#">
  <!--- Where did it come from? --->
  <cfset orig="#CFEvent.originatorID#">

  <!--- Process the input, generate a message with the new temperature. --->
  <!--- Input format is: degrees, temperature. --->
  <cfif listlen(message) eq 2>
    <cfif (listgetat(message,1) IS "F") OR
        (listgetat(message,1) IS "Fahrenheit") OR
        (listgetat(message,1) IS "C") OR
        (listgetat(message,1) IS "Celsius")>
      <cfset scale=listgetat(message,1)>
      <cfif isNumeric(listgetat(message,2))>
        <cfset temperature=listgetat(message,2)>
        <cfswitch expression="#scale#">
          <cfcase value="F, Fahrenheit">
            <cfset retmsg = temperature & " degrees Fahrenheit is "
```

```
                & (temperature-32.0) * (5.0/9.0) & " degrees Celsius">
            </cfcase>
            <cfcase value="C, Celsius">
              <cfset retmsg = temperature & " degrees Celsius is "
                &  (temperature * 9.0/5.0) + 32 & " degrees Fahrenheit">
            </cfcase>
          </cfswitch>
        <cfelse>
          <cfset retmsg=errormsg>
        </cfif>
      <cfelse>
        <cfset retmsg=errormsg>
      </cfif>
    <cfelse>
      <cfset retmsg=errormsg>
    </cfif>

    <!--- Fill the return value as required for the event gateway type. --->
    <cfif arguments.CFEVENT.GatewayType is "Socket">
      <cfset retValue = structNew()>
      <cfset retValue.MESSAGE = retmsg>
      <cfset retValue.originatorID = orig>
    <cfelseif (arguments.CFEVENT.GatewayType is "Sametime") OR
        (arguments.CFEVENT.GatewayType is "XMPP")>
      <cfset retValue = structNew()>
      <cfset retValue.MESSAGE = retmsg>
      <cfset retValue.BuddyID = arguments.CFEVENT.DATA.SENDER>
      <cfset retValue.originatorID = orig>
    <cfelseif arguments.CFEVENT.GatewayType is "SMS">
      <cfset retValue = structNew()>
      <cfset retValue.command = "submit">
      <cfset retValue.sourceAddress = arguments.CFEVENT.gatewayid>
      <cfset retValue.destAddress = arguments.CFEVENT.originatorid>
      <cfset retValue.shortMessage = retmsg>
    </cfif>

    <!--- Send the return message back. --->
    <cfreturn retValue>

</cffunction>
</cfcomponent>
```

Sending a message using the SendGatewayMessage function

The `SendGatewayMessage` function has the following format:

`SendGatewayMessage(`*`gatewayID`*`, `*`messageStruct`*`)`

- The *gatewayID* parameter must be the gateway ID specified in the ColdFusion MX Administrator for the event gateway instance that will send the message.
- The *messageStruct* parameter is a structure whose contents depends on the requirements of the event gateway's `outgoingMessage` method, and possibly the recipient application. For example, in addition to any message, the structure might include a destination identifier.

The CFEvent instance passed to the event gateway contains these two parameters in the GatewayID and Data fields; the remaining fields are empty.

The following example sends a message to a logging CFC, which logs information to a file. If the `SendGatewayMessage` function returns "OK", the example code displays a message. The code uses an instance of the asynchronous CFML event gateway named Asynch Logger. The props variable used in the *messageStruct* parameter has two entries, the destination file and the message to log.

```
<cfscript>
  status = "No";
  props = structNew();
  props.Message = "Replace me with a variable with data to log";
  status = SendGatewayMessage("Asynch Logger", props);
  if (status IS "OK") WriteOutput("Event Message ""#props.Message#"" has been
  sent.");
</cfscript>
```

Note: To see the code for the CFC that logs the information, see "Using the CFML event gateway for asynchronous CFCs" on page 1038.

Using the GatewayHelper object

The ColdFusion `GetGatewayHelper` function tells ColdFusion to create and initialize a Java GatewayHelper object that provides event gateway-specific helper methods and properties. To use this function, the event gateway must implement a GatewayHelper class. For example, an instant messaging event gateway might make buddy list management methods available in a GatewayHelper object.

The ColdFusion `GetGatewayHelper` function takes a single parameter, the ID of the event gateway instance that provides the helper, and returns a GatewayHelper Java object. The parameter value must be the gateway ID for the instance that is specified in the ColdFusion MX Administrator. If you do not want to hard-code an ID value in the application (for example, if your listener CFC can respond to multiple event gateway instances), get the gateway ID from the CFEvent structure of the first incoming message.

The CFML code accesses the GatewayHelper object's methods and properties using standard ColdFusion Java object access techniques, as described in Chapter 37, "Integrating J2EE and Java Elements in CFML Applications," on page 917. For example, if an event gateway's GatewayHelper class includes an `addBuddy` method that takes a single String parameter, you could use the following code to get the ColdFusion MX XMPP or Sametime gateway GatewayHelper object and add a buddy to the buddies list:

```
<cfscript>
  myHelper = GetGatewayHelper(myGatewayID);
  status = myHelper.addBuddy("jsmith23", "Jim Smith", "support");
</cfscript>
```

Using the event gateway error log file

When a standard ColdFusion MX event gateway encounters an error that does not prevent the event gateway from continuing to process, it logs it to the eventgateway.log file in the ColdFusion MX logs directory. Other event gateways can also to log information in this file, or to other application-specific files in the logs directory.

The standard ColdFusion MX event gateways log errors in interaction with any messaging server, errors in messages sent by the ColdFusion application, and recoverable errors in event gateway operation. The event gateways also log informational status messages for significant normal events, including event gateway initialization and restarts.

ColdFusion MX event gateway messages in the eventgateway.log file normally have the following format:

```
gatewayType (gatewayID) message body
```

When you are developing an event gateway application, you can use the ColdFusion Log viewer to inspect the eventgateway.log file and filter the display by using the gateway type and possibly the gateway ID as keywords. By selecting different severity levels, you can get a good understanding of errors and possible inefficiencies in your application and event gateway operation.

Deploying event gateways and applications

To deploy an event gateway application in a ColdFusion server, you must install your listener CFC and configure an gateway instance that uses the CFC.

To deploy an event gateway application

1. Ensure that the ColdFusion MX Administrator is configured with the required event gateway type. If it is not, deploy the event gateway type as described in "Deploying an event gateway" on page 1100.
2. If the event gateway type requires a configuration file, ensure that there is a valid file in the gateway\config directory. Some event gateways might be designed to let multiple event gateway instances share a configuration file. Others might require a separate file for each event gateway instance.
3. Install the event gateway application listener CFC and any other application components. ColdFusion MX provides a *cf_root*\gateways\cfc directory as a convenient location for these CFCs, and includes a mapping in the ColdFusion MX Administrator page for that directory. However, ColdFusion does not require you to install the listener CFC in this directory.
4. Configure an event gateway instance on the Gateways page of the Event Gateways section in the ColdFusion MX Administrator, as described in "Configuring an event gateway instance" on page 1028.

Using the CFML event gateway for asynchronous CFCs

The ColdFusion MX CFML event gateway lets CFML code send a message to CFC methods asynchronously. This event gateway lets you initiate processing by a CFC method without waiting for it to complete or return a value. Possible uses for asynchronous CFCs that you access using this event gateway include the following:

- Reindexing a Verity collection with new information without delaying an application, for example, when a user uploads a new file
- Logging information, particularly if there is significant amount of data to log
- Running batch processes that might take a substantial amount of time to complete

Because asynchronous CFCs run independently of a request, they do not provide feedback to the user. You must save any results or error information to a file, data source, or other external resource.

By default, ColdFusion delivers the message to a CFC method named onIncomingMessage. You can specify any method name, however, in the `SendGatewayMessage` method's *data* parameter.

CFML event gateway data structure

The structure that you use in the CFML `SendGatewayMessage` function can include two types of fields:

- The structure can include any number of fields with arbitrary contents for use in by the CFC.
- Several optional fields can configure how the gateway delivers the information to the CFC.

The CFML gateway looks for the following optional fields, and, if they exist, uses them to determine how it delivers the message. Do not use these field names for data that you send to your CFC method.

Field	Use
cfcpath	Overrides the CFC path specified in the ColdFusion MX Administrator. This field lets you use a single gateway configuration in the ColdFusion MX Administrator multiple CFCs.
method	Sets the name of the method to invoke in the CFC. The default method is `onIncomingMessage`. This field lets you use a single gateway configuration in the ColdFusion MXAdministrator for a CFC that has several methods.
originatorID	Sets the originatorID field of the CFEvent object that ColdFusion MX delivers to the CFC. The default value is CFMLGateway.
timeout	Sets the time-out, in seconds, during which the listener CFC must process the event request and return before ColdFusion gateway services terminates the request. The default value is the Timeout Request value set on the Server Settings page in the ColdFusion MX Administrator. Set this value if a request might validly take longer to process than the default timeout; for example, if the request involves a very long processing time.

Using the CFML gateway

The following procedure describes how to use an asynchronous CFC that has a single, `onIncomingMessage` method.

To use an asynchronous CFC:

1. Create a CFC with an `onIncomingMessage` method. Put the CFC in an appropriate directory for your application. For example, you can put it in the *cf_root*\WEB-INF\cfusion\gateway\cfc directory on J2EE configurations, in the *cf_root*\gateway\cfc directory on server configurations, or in a subdirectory of these directories. ColdFusion MX is installed with mappings to these cfc gateway directories.

   The `onIncomingMessage` method must take a CFEvent structure that contains input information in its Data field, and processes the contents of the Data field as needed.
2. Use the Gateways page in the ColdFusion MX Administrator to add an instance of the CFML event gateway type. Specify the following:
   - A unique Gateway ID.
   - The path to the CFC that you created in step 1.
   - The startup mode. Select Automatic startup mode to start the event gateway when ColdFusion MX starts up.
   - Do *not* specify a configuration file.
3. Start the event gateway instance.
4. Write CFML code that uses `SendGatewayMessage` functions to send messages in structures to the event gateway instance ID that you specified in step 2. The `SendGatewayMessage` function returns true if the gateway successfully queues the message in the ColdFusion MX Gateway Service; false, otherwise. It does not ensure that the CFC receives or processes the message.
5. Run your CFML application.

Example: logging messages

The following asynchronous CFML event gateway CFC uses the `cflog` tag to log a message to a file in the ColdFusion MX logs directory. The CFC takes a message with the following fields:

- file The name of the file in which to put the message. The default value is defaultEventLog.
- type The `cflog` type attribute to use. The default value is info.
- message The message text.

```
<cfcomponent>
  <cffunction name="onIncomingMessage" output="no">
    <cfargument name="CFEvent" type="struct" required="yes">
    <cfscript>
      if (NOT IsDefined("CFEvent.Data.file")) {
        CFEvent.Data.file="defaultEventLog"; }
      if (NOT IsDefined("CFEvent.Data.type")) {
        CFEvent.Data.type="info"; }
    </cfscript>
    <cflog text="#CFEvent.Data.message#"
      file="#CFEvent.Data.file#"
```

```
        type="#CFEvent.Data.type#"
        thread="yes"
        date="yes"
        time="yes"
        application="yes">
    </cffunction>
</cfcomponent>
```

The following minimal CFML page tests the event gateway:

```
Sending an event to the CFML event gateway that is registered in the
  ColdFusion MX Administrator as Asynch Logger.<br>
<cfscript>
  status = false;
  props = structNew();
  props.Message = "Replace me with a variable with data to log";
  status = SendGatewayMessage("Asynch Logger", props);
  if (status IS True) WriteOutput('Event Message "#props.Message#" has been
  sent.');
</cfscript>
```

Using the example event gateways and gateway applications

ColdFusion MX provides several example event gateways and applications in the *cf_root*\WEB-INF\cfusion\gateway directory on J2EE configurations or the *cf_root*\gateway directory on server configurations. These gateways provide example code that you can examine or use in developing your gateways. They are intended as examples only, and are **not** complete, product-quality, implementations.

Example event gateways

The gateway\src\examples directory and its subdirectories include the sources for the example event gateways. Compiled versions are located in the gateway\lib\examples.jar file. The following sections briefly describe the event gateways and their functions. For more detailed information, see the code and comments in the files.

EmptyGateway

The EmptyGateway.java file contains an event gateway template that you can use as a skeleton for creating your own event gateway. For more information on this class, and on creating new event gateways, see Chapter 45, “Creating Custom Event Gateways,” on page 1085.

SocketGateway

The SocketGateway event gateway listens on a TCP/IP port. Therefore, you can use this gateway for applications that send and respond to messages using TCP/IP-based protocols such as Telnet, or for applications that send messages between sockets. For example, a simple gateway application that might respond to messages from a Telnet terminal client without supporting the full Telnet protocol.

Note: The ColdFusion MX Administrator uses Socket as the gateway type name for the SocketGateway class.

The SocketGateway.java file defines two classes: SocketGateway, the event gateway, and SocketHelper, a GatewayHelper class. The Source file is located in the gateway\src\examples\socket directory.

SocketGateway Listens on a TCP/IP port. This event gateway is multithreaded and can handle multiple clients simultaneously. It can send outgoing messages to existing clients, but cannot establish a link itself.

By default, the SocketGateway class listens on port 4445, but you can specify the port number in a configuration file. The file should contain a single line in the following format:

```
port=portNumber
```

SocketHelper A GatewayHelper class with the following methods:

- `getSocketIDs()` returns an array containing the socket IDs of all Java sockets that are open. The event gateway opens a socket for each remote client.
- `killSocket(String socketid)` Removes the specified socket. Returns a Boolean success indicator.

DirectoryWatcherGateway

The DirectoryWatcherGateway event gateway sends events to the listener CFC when a file is created, deleted, or modified in a directory. The watcher runs in a thread that sleeps for an interval specified in the configuration file, and when the interval has passed, checks for changes since the last time it was awake. If it finds added, deleted, or changed files, it sends a message to a listener CFC. You can configure separate CFCs for add, delete, and change events, or use a single CFC for all events. The source for this event gateway is located in the gateway/src/examples/watcher directory.

Note: The ColdFusion MX Administrator uses DirectoryWatcher as the gateway type name for the DirectoryWatcherGateway class.

Configuration file

This event gateway requires a configuration file, consisting of lines in the following format:

```
directory=C:/temp
```

Note: If you use backward slash characters (\) as directory separators in Windows the file paths, you must escape them by using double slashes, as in C:\\temp. You can use forward slashes (/) as the directory separator on all operating systems, including Windows.

The configuration file can have comment lines, preceded by a number sign (#). If you omit a property or comment it out, ColdFusion uses the default value. If you specify a property with no value, ColdFusion sets an empty property. The configuration file can define the following values:

Property	Req/Opt	Description
directory	Required	Path to the directory to watch.
recurse	Optional	Whether to check subdirectories. The default value is `No`.

Property	Req/Opt	Description
extensions	Optional	Comma-delimited list of extensions to watch. The event gateway logs only changed files with these extensions. An asterisk (*) indicates all files. The default is all files.
interval	Optional	Number of milliseconds between the times that the event gateway checks the directory. The default value is 60 seconds.
addFunction	Optional	Name of the function to call when a file is added. The default value is `onAdd`.
changeFunction	Optional	Name of the function to call when a file is changed. The default value is `onChange`.
deleteFunction	Optional	Name of the function to call when a file is deleted. The default value is `onDelete`.

An example configuration file is located in the gateway\config\directory-watcher.cfg file.

CFC methods

When the directory contents change, the event gateway calls one of the following CFC listener methods, unless you change the names in the configuration file:

- `onAdd`
- `onChange`
- `onDelete`

The CFEvent.Data field sent to the listener methods includes the following fields:

Field	Description
TYPE	Event type, one of ADD, CHANGE, DELETE.
FILENAME	Name of the file that was added, deleted, or changed.
LASTMODIFIED	The date and time that the file was created or modified. This field is not included if the file was deleted.

The event gateway supports multiple listener CFCs and sends the event messages to all listeners. The event gateway is one-way; it watches for events and dispatches event information to a CFC, but it does not accept return values from the CFC or input from `SendGatewayMessage` functions.

The directory watcher logs errors to the watcher.log file in the ColdFusion logs directory.

Example DirectoryWatcher application

The following code shows a simple directory watcher application. It enters a line in a log file each time a file is added, deleted, or changed in the directory specified in the configuration file. ColdFusion MX includes the date and time that a log entry is made. However, if the directory watcher monitors changes infrequently, for example once every minute or more, the time in the log entry might differ from the time a file was added or changed, so the information includes the time (but not date) for these actions.

```
<cfcomponent>
<cffunction name="onAdd" output="no">
  <cfargument name="CFEvent" type="struct" required="yes">
  <cfset data=CFEvent.data>
  <cflog file="MydirWatcher" application="No"
    text="ACTION: #data.type#;  FILE: #data.filename#;
    TIME: #timeFormat(data.lastmodified)#">
</cffunction>

<cffunction name="onDelete" output="no">
  <cfargument name="CFEvent" type="struct" required="yes">
  <cfset data=CFEvent.data>
  <cflog file="MydirWatcher" application="No"
  text=" ACTION: #data.type#;  FILE: #data.filename#">
</cffunction>

<cffunction name="onChange" output="no">
  <cfargument name="CFEvent" type="struct" required="yes">
  <cfset data=CFEvent.data>
  <cflog file="MydirWatcher" application="No"
    text=" ACTION: #data.type#;  FILE: #data.filename#;
    TIME: #timeFormat(data.lastmodified)#">
</cffunction>
</cfcomponent>
```

JMSGateway

The JMSGateway class acts as a Java Messaging Service consumer or producer. The source for this event gateway is located in gateway/src/examples/JMS. The gateway requires a configuration file, which is in gateway/config/jmsgateway.cfg. For full documentation of the configuration options, See the configuration file. The ColdFusion MX Administrator lists the compiled gateway (which is included in the gateway\lib\examples.jar file) on the Gateway Types page.

Note: The ColdFusion MX Administrator uses JMS as the gateway type name for the JMSGateway class.

Using the JMS Gateway as a consumer

The JMSGateway class creates a subscriber to the topic specified in the configuration file. The gateway consumes the following types of messages:

- TextMessage
- BytesMessage containing raw UTF-8 text

The gateway passes the contents of the message to the configured CFC in the event structure, as follows:

Field	Contents
data.id	Message correlation ID
data.msg	Text of the message
gatewayType	Gateway type: JMS
originatorID	Topic name from which the message was consumed

The listener CFC method must be named `onIncomingMessage`. If the CFC method does not send a message in response, it should return a structure containing a status field with a value of OK or EXCEPTION. (In this case, The gateway checks the return status field, but does not process these return values further.) To send a message, the CFC method must return a structure as documented in the following section.

Using the JMS Gateway as a producer

To send a JMS message, the return value of your CFC method or the second, *`messageStruct`*, parameter to the `SendGatewayMessage` function must be a structure with the following fields:

Field	Contents
status	Must be SEND.
topic	Name of the topic to publish the message to.
id	(Optional) The JMS correlation ID to associate with the message. The default is null.
message	Text of the message to publish.
asBytes	(Optional) How to publish the message: • If omitted, no, or false, send the message as text. • If any other value, send the message as byte-encoded UTF-8.

If you send the message in a `SendGatewayMessage` function, the function returns OK if the gateway sends the message, or EXCEPTION if it fails to send the message.

Menu example application

ColdFusion MX is installed with a menu-based responder application. The menu application is written to work with any of the standard ColdFusion MX event gateways (SMS, XMPP, and Sametime) and with the Socket example event gateway, and ColdFusion MX is preconfigured with an instance of the application that uses SMS, as follows:

- The Gateways page in the ColdFusion MX Administrator includes a gateway instance for this application that uses the SMS gateway type.
- The gateway/cfc/examples/menu directory and its subdirectories include the CFML for the application

- The gateway/config/sms-test.cfg file is configured to use this application with the SMS client (phone simulator), and short message service center (SMSC) server simulator that are provided with ColdFusion MX.

The application presents users with a drill-down menu of tools that they can use, including a weather report, stock information, status and configuration information, and language tools such as a dictionary.

The code for this application is relatively complex and is distributed among 13 files. The following brief description provides an overview of how it works. To get a full understanding of how the application works, see the source code.

- The top level, menu, directory contains two files: Application.cfm and main.cfc.
- The Application.cfm file consists of a single `cfapplication` tag that enables session management and names the application. Session variables maintain the current state information of the session, such as the active menu, and so on.
- The main.cfc file contains the master CFC; the event gateway configuration in ColdFusion MX Administrator uses it as the listener CFC. The main CFC file processes CFEvent structures from the event gateway. It does the following:
  a. Inspects the gatewayType field to determine the rest of the structure contents. This is necessary because different event gateways put the message in fields with different names.
  b. If a Session.menu variable does not exist, initializes the menu system. To do so, it calls methods in two other CFCs: menu and menunode. These two CFCs contain the menu system code.
  c. Calls the session.menu.process method to process the user's input. This method can dispatch a message to an individual application for processing, if appropriate.
- The apps directory contains several CFCs. Each file contains the code for a single application, such as the weather report or dictionary lookup (definition.cfc).

To use the menu application with the Socket event gateway:

1. On the Gateway Settings page in the ColdFusion MX Administrator, click the Start SMS Test Server button.
2. On the Gateways page in the ColdFusion MX Administrator, start the SMS Menu App - 5551212 event gateway by clicking the green play button (third button from the left in the Actions column). If the Status does not say Running after a few seconds, click Refresh to check that the server started.
3. In the *cf_root*\WEB-INF\cfusion\bin directory on J2EE configurations or the *cf_root*\bin directory on server configurations, run the SMSClient.bat file (on Windows) or SMSClient.sh file (on UNIX or Linux) to start the SMS phone simulator. The simulator is preconfigured by default to "call" the default SMS event gateway configuration.
4. Enter any character by typing or by using the mouse to click the simulator keypad, and press Enter on your keyboard or click Send on the simulator.
5. The menu application responds with the top-level menu. Enter L for language tools such as a dictionary and thesaurus, S to get stock quotes or weather forecasts, or C to get information about the server. Press Enter on your keyboard or click Send on the simulator.

6. The application displays a submenu. For example, if you select S in step 5, the options are Q for a stock quote, W for weather, or B to go back to the previous menu. Enter your selection.
7. The application requests information such as a Zip code for the weather, stock symbol for a price, word for the dictionary, and so on. Enter and send the required information (or enter B to go back to the menu).
8. The application gets and displays the requested information. Depending on the application, you might also be prompted to enter M to get more. Enter M (if there is more information available), another term, or B to return to the previous menu.
9. Continue by entering menu items and detailed information requests.
10. To exit, select File > Exit from the menu bar.

CHAPTER 43
Using the Instant Messaging Event Gateways

This chapter describes how to develop an application that uses either of two instant message (IM) event gateway types provided with Macromedia ColdFusion MX: an IBM Lotus Sametime gateway, and an Extensible Messaging and Presence Protocol (XMPP) gateway. It briefly describes IM and the tools that ColdFusion MX provides for developing IM applications. It then describes how you can configure the IM event gateway and write and test applications that use the event gateway.

This chapter assumes that you are familiar with ColdFusion event gateway principles and programming techniques, as described in Chapter 42, "Using Event Gateways," on page 1019. It does not assume that you are familiar with instant messaging, but a basic knowledge is helpful.

Note: ColdFusion MX Standard Edition does not support event gateways.

Contents

About ColdFusion and instant messages 1047

Configuring an IM event gateway 1050

Handling incoming messages 1052

Sending outgoing messages 1053

Sample IM message handling application 1054

Using the GatewayHelper object 1058

About ColdFusion and instant messages

ColdFusion includes two instant messaging gateway types: one for messaging using the XMPP protocol, and one for IBM Lotus Instant Messaging (Sametime). These gateway types use identical interfaces for sending and receiving messages and for managing the IM presence information and infrastructure. This chapter, therefore, refers to IM gateways, and only describes the two types where there are differences.

The ColdFusion IM gateways act as IM clients and let you do the following:

- Send and receive instant messages.
- Send and respond to buddy or friend requests and manage buddy/friend information.
- Set and get status and other information.
- Receive and handle messages from the IM server.

About XMPP

XMPP (Extensible Messaging and Presence Protocol) is an open, XML-based protocol for instant messaging. It is the core protocol of the Jabber Instant Messaging and Presence technology that is developed by the Jabber Software Foundation. As of November 2004, XMPP was defined by four Internet Engineering Task Force (IETF) specifications (RFCs), numbers 3920-3922. RFC 3920 covers the XMPP core, and 3921, covers instant messaging and presence. Numerous XMPP servers and clients are available. ColdFusion MX supports the IETF XMPP protocol.

The following websites provide additional information about the XMPP protocol:

- Jabber Software Foundation: www.jabber.org/. This site includes information on available XMPP servers and clients.
- IETF has copies of the internet standards for XMPP: www.ietf.org/rfc/.
- The xmpp.org website was under development as of December 2004; at that time it included several useful links, including links to relevant specifications: www.xmpp.org/.

About IBM Lotus Instant Messaging (Sametime)

IBM Lotus Instant Messaging, commonly referred to as Lotus Sametime, is the IBM product for real-time collaboration. For more information about this product, see www.lotus.com/sametime.

About IM application development and deployment

The following sections introduce the ColdFusion IM application development tools and process, and discuss IM messaging providers.

ColdFusion MX IM gateway classes

ColdFusion provides the following instant messaging gateway classes:

XMPPGateway The class for the XMPP event gateway type

SAMETIMEGateway The class for the IBM Lotus Instant Messaging event gateway

You implement your IM application by configuring a gateway instance in ColdFusion MX Administrator that uses one of these gateway classes and creating a ColdFusion application that uses the gateway instance to communicate with an instant messaging server.

Application development and deployment process

The following is a typical process for developing and deploying an IM application:

1. Design your application.
2. Configure an IM event gateway instance to use an available XMPP or Lotus Sametime server.
3. Write your CFCs, CFM pages, and any other application elements.
4. Test your application using your XMPP or Lotus Sametime server and an appropriate client application.
5. Deploy the application, as described in "Deploying event gateways and applications" on page 1037.

How the IM event gateway and provider interact

Each IM event gateway instance has a single instant messaging ID. You must establish the ID and its related password on the IM server using server-specific tools, such as a standard instant messaging client. In ColdFusion MX, you set the ID, password, and other gateway-specific information in a gateway configuration file, and you create a gateway instance that uses this file.

When you start the gateway, it logs onto the IM server with the ID and password, and receives and sends the messages for the ID. The gateway sends incoming messages to a CFC, which you specify when you configure the gateway instance in the ColdFusion MX Administrator. The gateway passes outgoing messages from this CFC and from other CFML code to the IM server.

The IM event gateway also provides a number of helper methods for managing the gateway and its configuration information.

Incoming message handling

You write the following ColdFusion CFC methods to handle incoming messages and requests from the IM event gateway. These CFCs receive messages from the IM server and can respond to them by setting a return value.

CFC method	Message type
`onIncomingMessage`	Standard message from IM users.
`onAddBuddyRequest`	Requests from others to add the gateway ID to their buddy list.
`onAddBuddyResponse`	Responses from others to requests from your gateway to add them to your buddy lists. Also used by buddies to ask to be removed from your list.
`onBuddyStatus`	Presence status messages from other users.
`onIMServerMessage`	Error and status messages from the IM server.

For more information on these methods, see "Handling incoming messages" on page 1052.

Outgoing message handling

Applications send outgoing instant messages using the CFML `SendGatewayMessage` method. Incoming message-handling CFC methods can also send messages, including responses to requests from others to add the ColdFusion gateway's ID to their buddy list. For more information on sending messages, see "Sending outgoing messages" on page 1053.

IMGatewayHelper methods

The ColdFusion IM gateway provides the IMGatewayHelper class, a gateway helper that you can access by calling the CFML `GetGatewayHelper` function. The IMGatewayHelper class has methods that let you do the following:

- Get and set gateway configuration information and get gateway statistics
- Get and set the gateway online presence status
- Manage the gateway's buddy list
- Manage permissions for others to get information about the gateway status.

For more information on using GatewayHelper methods, including lists of all the methods, see "Using the GatewayHelper object" on page 1058.

Configuring an IM event gateway

You provide IM-specific configuration information to the IM event gateway in a configuration file. You specify the configuration file location when you configure the IM event gateway instance in the ColdFusion MX Administrator. ColdFusion provides sample XMPP and Lotus Sametime event gateway configuration files in the *cf_root*\WEB-INF\cfusion\gateway\config directory on J2EE configurations, and *cf_root*\gateway\config directory on server configurations. The configuration file can have the following information.

Note: The default value in the table is the value the gateway uses if the configuration file omits the property, not the value in the default configuration files.

Property	Default value	Description
userID	none	(Required) The IM user ID to use to connect to the IM server.
password	none	(Required) Password for the user.
secureprotocol	none	XMPP only. Required if you set `securerequirement` to true. The protocol to use for secure communications. The following values are valid: • TSL • SSL

Property	Default value	Description
securerequirement	false	XMPP only. Specifies whether the gateway must use secure communications. The following values are valid: • true • false If this value is true, you must specify a `secureprotocol` value, and connections succeed only if a secure connection is established.
serverip	XMPP: jabber.org Sametime: stdemo3.dfw.ibm.com	Address of XMPP or Lotus Sametime server to which to send messages. Can be a server name or IP address.
serverport	XMPP: 5222 Sametime:1533	Port on the server to which to send the messages. If the XMPP `secureprotocol` parameter is set to SSL, speechify 5223.
retries	-1	Integer number of times to retry connecting to the IM server on gateway startup or if the Gateway gets disconnected. 0 = do not to retry -1 = try forever
retryinterval	5	Real number of seconds to wait between connection attempts. The minimum is 1 second.
onIncomingMessage Function	onIncomingMessage	Name of CFC method to call to handle an incoming message. If you specify the property without a value, such as "onIncomingMessageFunction=", the gateway does not send this event to a CFC.
onAddBuddy RequestFunction	onAddBuddy Request	Name of CFC method to call to handle an incoming buddy request. If you specify the property without a value, the gateway does not send this event to a CFC.
onAddBuddy ResponseFunction	onAddBuddy Response	Name of CFC method to call to handle an incoming response to a buddy request sent by ColdFusion. If you specify the property without a value, the gateway does not send this event to a CFC.
onBuddyStatus Function	onBuddyStatus	Name of CFC method to call to handle an incoming buddy status message, such as If you specify the property without a value, the gateway does not send this event to a CFC.
onIMServerMessage Function	onIMServerMessage	Name of CFC method to call to handle an incoming message method. If you specify the property without a value, the gateway does not send this event to a CFC.

Note: If you do not have a CFC method to handle any of the event types, you must specify the corresponding property without a value. Use the following entry in the configuration file, for example, if you do not have a method to handle IMServerMessage events:

```
onIMServerMessageFunction=
```

Handling incoming messages

The IM event gateway handles five types of messages, and your CFC must implement a listener method for each message type. The following table describes the message-handling CFC methods and the messages they handle. It lists the default CFC method names; however, you can change the names in the gateway configuration file.

CFC method	Description
`onIncomingMessage`	Standard message from an IM user. The application processes the message body appropriately; for example, it could display the message in an interface window. This method can return a response message to the sender.
`onAddBuddyRequest`	Request from another IM user to add your application's IM ID to their buddy list. The CFC must determine whether to accept or reject the request, or to take no action. No action might be appropriate in cases where the request must be reviewed offline for approval and responses are sent at a later time. The CFC returns a message with the decision as a command value and optionally a text message specifying the reason. If you accept the request, the requestor automatically gets added to the list of IDs that can get status information for the gateway. If you specify no action, ColdFusion does not respond.
`onAddBuddyResponse`	Response from another IM user to a request from the gateway to be added to their buddy list. The response message is accept or decline. Your application can handle this response as appropriate; for example, to add or remove the ID from a list of message recipients. This method does not return a value.
`onBuddyStatus`	Message indicating a gateway buddy's status. Received when a buddy's status changes; for example, from OFFLINE to ONLINE. This method does not return a value.
`onIMServerMessage`	Status messages from the IM server, such as warning or error messages. The messages you might receive depend on the IM server that sends them. For information on the server messages, see the documentation for the IM server that your gateway instance uses. This method does not return a value.

For detailed information on each method, including examples of their use, see "IM Gateway CFC incoming message methods" in *CFML Reference*. For an example that uses these functions, see "Sample IM message handling application" on page 1054.

Sending outgoing messages

You use the SendGatewayMessage CFML function or the return value of a CFC listener method to send outgoing messages. The ColdFusion MX 7 IM gateway accepts the following outgoing message commands:

Command	Description
submit	(Default) Sends a normal message to another IM user.
accept	Accepts an add buddy request. Adds the buddy to the list of IDs that get your presence information and sends an acceptance message to the buddy ID.
decline	Declines an add buddy request and sends a rejection message to the buddy ID.
noact	Tells the gateway to take no action. The gateway logs a message that indicates that it took no action, and contains the gateway type, gateway ID, and buddy ID.

The message structure that you return in the gateway listener CFC function or use as the second parameter in the CFML SendGatewayMessage function can have the following fields. The table lists the fields and the commands in which they are used, and describes each field's use.

Field	Commands	Description
buddyID	All	The destination user ID.
command	All	The command; if omitted, ColdFusion treats the message as a submit command.
message	submit	A text message to send to the destination user.
reason	accept, decline	A text description of the reason for the action or other message to send to the add buddy requestor.

In typical use, a ColdFusion application uses the accept, decline, and noact commands in the return value of the onAddBuddyRequest method, and uses the submit command (or no command, because submit is the default command) in SendGatewayMessage CFML functions and the return value of the onIncomingMessage CFC method.

The SendGatewayMessage CFML function can send any command, and might be used to send an accept or decline message. One possible use is in an application where someone must review all buddy requests before they are added. In this case, the onAddBuddyRequest CFC method could initially send a noact command in its return value, and save the request information in a database. Administrators could use a separate ColdFusion application to review the request information. This application could use the SendGatewayMessage function with an accept or decline command to act on the request and inform the requestor.

The following example onIncomingMessage method of a listener CFC echoes incoming IM messages to the message originator:

```
<cffunction name="onIncomingMessage" output="no">
  <cfargument name="CFEvent" type="struct" required="yes">
  <cfset retValue.MESSAGE = "echoing: " & CFEvent.DATA.message>
  <cfset retValue.BuddyID = arguments.CFEVENT.DATA.SENDER>
  <cfreturn retValue>
</cffunction>
```

Sample IM message handling application

The application described in this section consists of two CFCs: an employee phone directory lookup CFC that responds to an onIncomingMessage event, and a Gateway management CFC that responds to all other events. This example shows how an application can respond to events and send outgoing messages.

You can configure a gateway to use both CFCs by entering the paths to the two CFCs, separated by a comma, in the CFC Path field of the Add/Edit ColdFusion Event Gateways form on the Gateways page in the ColdFusion MX Administrator.

Phone directory lookup CFC

The following CFC implements a simple employee phone directory lookup application. The user sends an instant message containing some part of the name to be looked up (a space requests all names). The `onIncomingMessage` response depends on the number matches.

- If there is no match, the `onIncomingMessage` function returns a message indicating that there are no matches.
- If there is one match, the function returns the name, department, and phone number.
- If there are up to ten matches, the function returns a list of the names preceded by a number that the user can enter to get the detailed information.
- If there are over ten matches, the function returns a list of only the first ten names. A more complex application might let the user get multiple lists of messages to provide access to all names.
- If the user enters a number, and previously got a multiple-match list, the application returns the information for the name that corresponds to the number.

The following listing shows the CFC code:

```
<cfcomponent>
  <cffunction name="onIncomingMessage">
    <cfargument name="CFEvent" type="struct" required="YES">
    <!--- Remove any extra white space from the message. --->
    <cfset message =  Trim(arguments.CFEvent.data.MESSAGE)>
    <!--- If the message is numeric, a previous search probably returned a
      list of names. Get the name to search for from the name list stored in
      the Session scope. --->
    <cfif isNumeric(message)>
      <cfscript>
        if (structKeyExists(session.users, val(message))) {
          message = session.users[val(message)];
        }
      </cfscript>
    </cfif>

    <!--- Search the database for the requested name. --->
    <cfquery name="employees" datasource="cfdocexamples">
      select FirstName, LastName, Department, Phone
      from Employees
      where 0 = 0
      <!--- A space indicates the user entered a first and last name. --->
      <cfif listlen(message, " ") eq 2>
```

```
          and FirstName like '#listFirst(message, " ")#%'
          and LastName like '#listlast(message, " ")#%'
        <!--- No space: the user entered a first or a last name. --->
        <cfelse>
          and (FirstName like '#listFirst(message, " ")#%'
          or LastName like '#listFirst(message, " ")#%')
        </cfif>
      </cfquery>

      <!--- Generate and  return the message.--->
      <cfscript>
        retrunVal = structNew();
        retrunVal.command = "submit";
        retrunVal.buddyID = arguments.CFEvent.data.SENDER;

        //No records were found.
        if (employees.recordCount eq 0) {
          retrunVal.message = "No records found for '#message#'";
        }
        //One record was found.
        else if (employees.recordCount eq 1) {
        // Whitespace in the message text results in bad formatting,
        // so the source cannot be indented.
          retrunVal.message = "Requested information:
#employees.firstName# #employees.lastName#
#employees.Department#
#employees.Phone#";
        }
        //Multiple possibilities were found.
        else if (employees.recordCount gt 1) {
          //If more than ten were found, return only the first ten.
          if (employees.recordCount gt 10)
          {
            retrunVal.message = "First 10 of #employees.recordCount# records";
          }else{
            retrunVal.message = "Records found: #employees.recordCount#";
          }
          // The session.users structure contains the found names.
          // The record key is a number that is also returned in front of the
          // name in the message.
          session.users = structNew();
          for(i=1; i lte min(10, employees.recordCount); i=i+1)
          {
            // These two lines are formatted to prevent extra white space.
            retrunVal.message = retrunVal.message & "
#i# - #employees.firstName[i]# #employees.lastName[i]#";
            // The following two lines must be a single line in the source
            session.users[i]="#employees.firstName[i]#
              #employees.lastName[i]#";
          }
        }
        return retrunVal;
      </cfscript>
    </cffunction>
  </cfcomponent>
```

Status and request-handling CFC

The following CFC handles all IM events, except onIncomingMessage. It maintains an Application scope buddyStatus structure that contains information on the gateway buddies. This structure limits the interactions that are needed with the IM server to get buddy and status information. The application also logs significant events, such as requests to add buddies and error messages from the IM server. In particular, it does the following:

- The `onBuddyStatus` function updates the Application scope buddy status structure when the gateway gets an event message indicating that a buddy's status has changed.
- The `onAddBuddyRequest` function searches for the requested buddy's name in a data source. If it finds a single instance of the name, it adds the buddy and updates the status in the Application scope buddyStatus structure. If it doesn't find name, it declines the buddy request. If it finds multiple instances of the name, it tells the gateway to take no action. It also logs all actions.
- The `onAddBuddyResponse` function adds the buddy to the Application scope buddy status structure if the buddy request is accepted, and sets the current status. It logs all responses.
- The `onIMServerMessage` function logs all messages that it receives.

This example uses the IM_ID column of the Employees database of the cfdocexamples database that is included with ColdFusion MX 7. The entries in this column assume that you use an XMPP server "company." To run this example you must configure an XMPP server with this name and with clients with names in this database, or you must change the database entries to match IM server clients. You must also configure a gateway instance in the ColdFusion MX Administrator that uses this server.

The following listing shows the CFC code:

```
<cfcomponent>

<cffunction name="onBuddyStatus">
  <cfargument name="CFEvent" type="struct" required="YES">
  <cflock scope="APPLICATION" timeout="10" type="EXCLUSIVE">
    <cfscript>
    // Create the status structures if they don't exist.
      if (NOT StructKeyExists(Application, "buddyStatus")) {
        Application.buddyStatus=StructNew();
      }
      if (NOT StructKeyExists(Application.buddyStatus,
  CFEvent.Data.BUDDYNAME)) {
        Application.buddyStatus[#CFEvent.Data.BUDDYNAME#]=StructNew();
      }
      // Save the buddy status and timestamp.

  Application.buddyStatus[#CFEvent.Data.BUDDYNAME#].status=CFEvent.Data.BUDDY
  STATUS;

  Application.buddyStatus[#CFEvent.Data.BUDDYNAME#].timeStamp=CFEvent.Data.TI
  MESTAMP;
    </cfscript>
  </cflock>
</cffunction>
```

```
<cffunction name="onAddBuddyRequest">
  <cfargument name="CFEvent" type="struct" required="YES">
  <cfquery name="buddysearch" datasource="cfdocexamples">
    select IM_ID
    from Employees
    where IM_ID = '#CFEvent.Data.SENDER#'
  </cfquery>
  <cflock scope="APPLICATION" timeout="10" type="EXCLUSIVE">
    <cfscript>
      // If the name is in the DB once, accept; if it is missing, decline.
      // If it is in the DB multiple times, take no action.
      if (buddysearch.RecordCount IS 0) {
        action="decline";
        reason="Invalid ID";
      }
      else if (buddysearch.RecordCount IS 1) {
        action="accept";
        reason="Valid ID";
        //Add the buddy to the buddy status structure only if accepted.
        if (NOT StructKeyExists(Application,
             "buddyStatus")) {
          Application.buddyStatus=StructNew();
        }
        if (NOT StructKeyExists(Application.buddyStatus,
             CFEvent.Data.SENDER)) {
          Application.buddyStatus[#CFEvent.Data.SENDER#]=StructNew();
        }
        Application.buddyStatus[#CFEvent.Data.SENDER#].status=
          "Accepted Buddy Request";
        Application.buddyStatus[#CFEvent.Data.SENDER#].timeStamp=
          CFEvent.Data.TIMESTAMP;
        Application.buddyStatus[#CFEvent.Data.SENDER#].message=
          CFEvent.Data.MESSAGE;
      }
      else {
        action="noact";
        reason="Duplicate ID";
      }
    </cfscript>
  </cflock>
  <!--- Log the request and decision information. --->
  <cflog file="#CFEvent.GatewayID#Status"
      text="onAddBuddyRequest; SENDER: #CFEvent.Data.SENDER# MESSAGE:
  #CFEvent.Data.MESSAGE# TIMESTAMP: #CFEvent.Data.TIMESTAMP# ACTION:
  #action#">
  <!--- Return the action decision. --->
  <cfset retValue = structNew()>
  <cfset retValue.command = action>
  <cfset retValue.BuddyID = CFEvent.DATA.SENDER>
  <cfset retValue.Reason = reason>
  <cfreturn retValue>
</cffunction>
```

```
<cffunction name="onAddBuddyResponse">
  <cfargument name="CFEvent" type="struct" required="YES">
  <cflock scope="APPLICATION" timeout="10" type="EXCLUSIVE">
    <cfscript>
      //Do the following only if the buddy accepted the request.
      if (NOT StructKeyExists(Application, "buddyStatus")) {
        Application.buddyStatus=StructNew();
        }
      if (#CFEVENT.Data.MESSAGE# IS "accept") {
        //Create a new entry in the buddyStatus record for the buddy.
        if (NOT StructKeyExists(Application.buddyStatus,
            CFEvent.Data.SENDER)) {
          Application.buddyStatus[#CFEvent.Data.SENDER#]=StructNew();
        }
        //Set the buddy status information to indicate buddy was added.
        Application.buddyStatus[#CFEvent.Data.SENDER#].status=
          "Buddy accepted us";
        Application.buddyStatus[#CFEvent.Data.SENDER#].timeStamp=
          CFEvent.Data.TIMESTAMP;
        Application.buddyStatus[#CFEvent.Data.SENDER#].message=
          CFEvent.Data.MESSAGE;
      }
    </cfscript>
  </cflock>
  <!--- Log the information for all responses. --->
  <cflog file="#CFEvent.GatewayID#Status"
    text="onAddBuddyResponse; BUDDY: #CFEvent.Data.SENDER# RESPONSE:
  #CFEvent.Data.MESSAGE# TIMESTAMP: #CFEvent.Data.TIMESTAMP#">
</cffunction>


<cffunction name="onIMServerMessage">
  <!--- This function just logs the message. --->
  <cfargument name="CFEvent" type="struct" required="YES">
  <cflog file="#CFEvent.GatewayID#Status"
    text="onIMServerMEssage; SENDER: #CFEvent.OriginatorID# MESSAGE:
  #CFEvent.Data.MESSAGE# TIMESTAMP: #CFEvent.Data.TIMESTAMP#">
</cffunction>
</cfcomponent>
```

Using the GatewayHelper object

The CFML `GetGatewayHelper` function returns a GatewayHelper object with several methods that manage your gateway and buddy list. The GatewayHelper methods let you do the following:

- Get and set gateway configuration information and get gateway statistics.
- Get and set the gateway online status.
- Manage the gateway's buddy list
- Manage permissions for others to get information about the gateway status.

The following sections briefly describe the class methods. For detailed information about each method, see "IM Gateway GatewayHelper class methods" in *CFML Reference*.

Gateway configuration information and statistics methods

The following table describes the methods that you can use to get and set configuration information and get gateway statistics:

Method	Description
`getName`	Returns the gateway's user name.
`getNickName`	Returns the gateway's display name (nickname).
`getProtocolName`	Returns the name of the instant messaging protocol (JABBER for XMPP, or SAMETIME).
`numberOfMessagesReceived`	Returns the number of messages received by the gateway since it was started.
`numberOfMessagesSent`	Returns the number of messages sent by the gateway since it was started.
`setNickName`	Sets the gateway's display name (nickname).

Gateway online status methods

The following table describes the methods that you can use to get and set the gateway's online availability status (presence information):

Method	Description
`getCustomAwayMessage`	Returns the gateway's custom away message if it has been set by the `setStatus` method.
`getStatusAsString`	Returns the online status of the gateway.
`getStatusTimeStamp`	Returns the date/time that the gateway changed its online status.
`isOnline`	Returns True if the gateway is connected to the IM server; otherwise, returns false.
`setStatus`	Changes the gateway's online status; for example, to away or idle.

Gateway buddy management methods

The following table describes the methods that you can use to manage the gateway's buddy list:

Method	Description
`addBuddy`	Adds a buddy to the gateway's buddy list and tells the IM server to send the gateway messages with the buddy's online state.
`getBuddyInfo`	Gets information about the specified user from the buddy list, deny list, and permit list.
`getBuddyList`	Returns the gateway's buddy list.
`removeBuddy`	Removes the specified user name from the gateway's buddy list and tells the IM server to stop sending the gateway messages with the user's online state.

Gateway permission management methods

The IM gateways can manage the information that other users can get about the gateway's online status.

Note: XMPP permission management is included in the XMPP 1.0 specification, but several XMPP servers that were available at the time of the ColdFusion MX 7 release do not support permission management.

The following table describes the gateway permission management methods:

Method	Description
`addDeny`	Tells the IM server to add the specified user to the gateway's deny list. If the `permitMode` is DENY_SOME, these users cannot receive messages on the gateway's state.
`addPermit`	Tells the IM server to add the specified user to the server's permit list. If the `permitMode` is PERMIT_SOME, these users receive messages on the gateway's state.
`getDenyList`	Returns the list of users that the server has been told not to send state information to.
`getPermitList`	Returns the list of users that the server has been told to send state information to.
`getPermitMode`	Gets the gateway's permit mode from the IM server. The permit mode determines whether all users can get the gateway's online state information, or whether the server uses a permit list or a deny list to control which users get state information.
`removeDeny`	Removes the user from the gateway's deny list.
`removePermit`	Removes the user from the gateway's permit list.
`setPermitMode`	Sets the gateway's permit mode on the IM server.

GatewayHelper example

This example lets you use the XMPP or SameTime GatewayHelper class to get and set status and other information, including managing buddy lists and view permissions lists.

```
<cfapplication name="gateway_tool" sessionmanagement="yes">

<!--- Set the gateway buddy name to default values.--->
<cfparam name="session.gwid" default="XMPP Buddy Manager">
<cfparam name="session.buddyid" default="hlichtin2@mousemail">

<!--- Reset gateway and buddy ID if form was submitted. --->
<cfif isdefined("form.submitbuddy")>
  <cfset session.buddyid=form.buddyid>
  <cfset session.gwid=form.gwid>
</cfif>

<!--- Display the current gateway and buddy ID. --->
<h3>Using the GatewayHelper</h3>
<!--- Form to display and reset gateway and Buddy ID. --->
```

```
<cfform action="#cgi.script_name#" method="post" name="changeIDs">
  Current buddy ID: <cfinput type="text" name="buddyid"
  value="#session.buddyid#"><br>
  Current gateway ID: <cfinput type="text" name="gwid"
  value="#session.gwid#"><br>
  <cfinput name="submitbuddy" value="Change gateway/buddy" type="submit">
</cfform>

<!--- When a buddy is set, display the links and forms to get and set
     information etc. Where form input is required, the form uses a GET method
     so all selections are represented by a url.cmd variable. --->

<cfoutput>
<h3>Select one of the following to get or set.</h3>
<ul>
  <li><a href="#cgi.script_name#?cmd=buddyinfo">buddyinfo</a>
  <li>LIST: <a href="#cgi.script_name#?cmd=buddylist">buddylist</a> |
    <a href="#cgi.script_name#?cmd=permitlist">permitlist</a> |
    <a href="#cgi.script_name#?cmd=denylist">denylist</a>
  <li>ADD: <a href="#cgi.script_name#?cmd=addbuddy">addbuddy</a> |
    <a href="#cgi.script_name#?cmd=addpermit">addpermit</a> |
    <a href="#cgi.script_name#?cmd=adddeny">adddeny</a>
  <li>REMOVE: <a href="#cgi.script_name#?cmd=removebuddy">removebuddy</a> |
    <a href="#cgi.script_name#?cmd=removepermit">removepermit</a> |
      <a href="#cgi.script_name#?cmd=removedeny">removedeny</a>
  <!--- NOTE: This list does not include OFFLINE because the gateway resets
  itself to online. --->
  <li>setStatus (XMPP):
    <cfloop list="ONLINE,AWAY,DND,NA,FREE_TO_CHAT" index="e">
      <a href="#cgi.script_name#?cmd=setstatus&status=#e#">#e#</a> |
    </cfloop>
  <li>setStatus (Sametime):
    <cfloop list="ONLINE,AWAY,DND,IDLE" index="e">
      <a href="#cgi.script_name#?cmd=setstatus&status=#e#">#e#</a> |
    </cfloop>
  <li>
    <form action="#cgi.script_name#" method="get">
      setStatus with CustomAwayMessage:
      <input type="hidden" name="cmd" value="setstatus2">
      <select name="status">
        <cfloop
  list="ONLINE,OFFLINE,AWAY,DND,IDLE,INVISIBLE,NA,OCCUPIED,FREE_TO_CHAT,ONPHO
  NE,ATLUNCH,BUSY,NOT_AT_HOME,NOT_AT_DESK,NOT_IN_OFFICE,ON_VACATION,STEPPED_O
  UT,CUSTOM_AWAY" index="e">
          <option value="#e#">#e#</option>
        </cfloop>
      </select>
      <input type="text" name="custommsg" value="(custom away massage)"
  size="30"/>
      <input type="submit"/>
    </form>
  <li>
    <form action="#cgi.script_name#" method="get">
      setNickName:
      <input type="hidden" name="cmd" value="setnickname">
```

```
        <input type="text" name="nickname" value="(enter nickname)">
        <input type="submit">
      </form>
---->
  <li>INFO: <a href="#cgi.script_name#?cmd=getname">getname</a> |
    <a href="#cgi.script_name#?cmd=getnickname">getnickname</a> |
    <a
  href="#cgi.script_name#?cmd=getcustomawaymessage">getcustomawaymessage</a>
  |
    <a href="#cgi.script_name#?cmd=getprotocolname">getprotocolname</a> |
    <a href="#cgi.script_name#?cmd=getstatusasstring">getstatusasstring</a> |
    <a href="#cgi.script_name#?cmd=isonline">isonline</a>
  <li>MESSAGE COUNT:
    <a
  href="#cgi.script_name#?cmd=numberofmessagesreceived">numberofmessagesrecei
  ved</a> |
    <a
  href="#cgi.script_name#?cmd=numberofmessagessent">numberofmessagessent</a>
  <li>RUNNING TIME: <a
  href="#cgi.script_name#?cmd=getsignontimestamp">getsignontimestamp</a> |
    <a href="#cgi.script_name#?cmd=getstatustimestamp">getstatustimestamp</a>
  <li>setPermitMode:
    <cfloop
  list="PERMIT_ALL,DENY_ALL,PERMIT_SOME,DENY_SOME,IGNORE_IN_LIST,IGNORE_NOT_I
  N_LIST"
        index="e"><a
  href="#cgi.script_name#?cmd=setpermitmode&mode=#e#">#e#</a> |
    </cfloop> <span class="note">doesn't work for XMPP</span>
  <li><a href="#cgi.script_name#?cmd=getpermitmode">getpermitmode</a>
  <li>setPlainTextMode:
    <cfloop list="PLAIN_TEXT,RICH_TEXT" index="e">
      <a href="#cgi.script_name#?cmd=setplaintextmode&mode=#e#">#e#</a> |
    </cfloop>
  <li><a href="#cgi.script_name#?cmd=getplaintextmode">getplaintextmode</a>
</ul>
</cfoutput>

<!--- The url.cmd value exists if one of the previous links or forms has been
  submitted, and identifies the type of request. --->
<cfoutput>
<cfif isdefined("url.cmd")>
  <!--- Get the GatewayHelper for the gateway. --->
  <cfset helper = getGatewayHelper(session.gwid)>
  <!--- Need to get the buddy list if requested the list or full buddy
  information. --->
  <cfswitch expression="#LCase(url.cmd)#">
    <cfcase value="buddylist,buddyinfo">
      <cfset ret=helper.getBuddyList()>
    </cfcase>
    <cfcase value="denylist">
      <cfset ret=helper.getDenyList()>
    </cfcase>
    <cfcase value="permitlist">
      <cfset ret=helper.getPermitList()>
    </cfcase>
```

```
<cfcase value="addbuddy">
  <cfset ret=helper.addBuddy("#session.buddyid#",
    "#session.buddyid#", "")>
</cfcase>
<cfcase value="addpermit">
  <cfset ret=helper.addPermit("#session.buddyid#",
    "#session.buddyid#", "")>
</cfcase>
<cfcase value="adddeny">
  <cfset ret=helper.addDeny("#session.buddyid#",
    "#session.buddyid#", "")>
</cfcase>
<cfcase value="removebuddy">
  <cfset ret=helper.removeBuddy("#session.buddyid#", "")>
</cfcase>
<cfcase value="removepermit">
  <cfset ret=helper.removePermit("#session.buddyid#", "")>
</cfcase>
<cfcase value="removedeny">
  <cfset ret=helper.removeDeny("#session.buddyid#", "")>
</cfcase>
<cfcase value="setstatus">
  <cfset ret=helper.setStatus(url.status, "")>
</cfcase>
<cfcase value="setstatus2">
  <cfset ret=helper.setStatus(url.status, url.custommsg)>
</cfcase>
<cfcase value="getcustomawaymessage">
  <cfset ret=helper.getCustomAwayMessage()>
</cfcase>
<cfcase value="getname">
  <cfset ret=helper.getName()>
</cfcase>
<cfcase value="getnickname">
  <cfset ret=helper.getNickname()>
</cfcase>
<cfcase value="getprotocolname">
  <cfset ret=helper.getProtocolName()>
</cfcase>
<cfcase value="getsignontimestamp">
  <cfset ret=helper.getSignOnTimeStamp()>
</cfcase>
<cfcase value="getstatusasstring">
  <cfset ret=helper.getStatusAsString()>
</cfcase>
<cfcase value="getstatustimestamp">
  <cfset ret=helper.getStatusTimeStamp()>
</cfcase>
<cfcase value="isonline">
  <cfset ret=helper.isOnline()>
</cfcase>
<cfcase value="numberofmessagesreceived">
  <cfset ret=helper.numberOfMessagesReceived()>
</cfcase>
<cfcase value="numberofmessagessent">
```

```
        <cfset ret=helper.numberOfMessagesSent()>
      </cfcase>
      <cfcase value="setnickname">
        <cfset ret=helper.setNickName(url.nickname)>
      </cfcase>
      <cfcase value="setpermitmode">
        <cfset ret=helper.setPermitMode(url.mode)>
      </cfcase>
      <cfcase value="getpermitmode">
        <cfset ret=helper.getPermitMode()>
      </cfcase>
      <cfcase value="setplaintextmode">
        <cfset ret=helper.setPlainTextMode(url.mode)>
      </cfcase>
      <cfcase value="getplaintextmode">
        <cfset ret=helper.getPlainTextMode()>
      </cfcase>
      <cfdefaultcase>
        <cfset ret[1]="Error; Invalid command. You shouldn't get this.">
      </cfdefaultcase>
    </cfswitch>
    <br>
    <!--- Display the results returned by the called GatewayHelper method. --->
    <strong>#url.cmd#</strong><br>
    <cfdump var="#ret#">
    <br>
    <!--- If buddy information was requested, loop through buddy list to get
        information for each buddy and display it. --->
    <cfif comparenocase(url.cmd, "buddyinfo") is 0 and arraylen(ret) gt 0>
    <b>Buddy info for all buddies</b><br>
      <cfloop index="i" from="1" to="#arraylen(ret)#">
        <cfdump var="#helper.getBuddyInfo(ret[i])#" label="#ret[i]#"></cfloop>
    </cfif>
  </cfif>
  </cfoutput>
```

CHAPTER 44
Using the SMS Event Gateway

This chapter describes how to develop an application that uses the short message service (SMS) event gateway type provided with Macromedia ColdFusion MX 7. It briefly describes SMS and the tools that ColdFusion MX 7 provides for developing SMS applications. It then describes how you can configure the SMS event gateway and write and test an application that uses the event gateway.

This chapter assumes that you are familiar with ColdFusion event gateway principals and programming techniques, as described in Chapter 42, "Using Event Gateways," on page 1019. It does not assume that you are familiar with SMS, but a basic knowledge would be helpful.

Note: ColdFusion MX Standard Edition does not support event gateways. ColdFusion MX Developer Edition limits connections so that each gateway can send messages to only a single SMS address.

Contents

About SMS and ColdFusion 1065
Configuring an SMS event gateway 1071
Handling incoming messages 1073
Sending outgoing messages 1075
ColdFusion MX SMS development tools 1080

About SMS and ColdFusion

Short Message Service (SMS) is a system designed for sending short, often text, messages to and from wireless devices, such as mobile phones or pagers. SMS is widely used in Europe and Asia and is becoming increasingly popular in the United States and elsewhere. Some uses for SMS include the following:

- Performing banking transactions
- Sending authentication codes, for example, to be used to access web resources
- Voting, such as popularity voting for reality television shows
- Initiating an action (such as a server reboot) and getting a response

- Notifying users of events such as package shipments or restaurant table availability, or providing stock or weather alerts
- Sending person-to-person text messages
- Presenting interactive text-based menus on a cell phone
- Providing cellular phone updates, such as direct download of logos
- Providing telematics and mobile or remote wireless device applications, such as soda machines, vehicle tracking, smart gas pumps, and so on

SMS protocol features include, but are not limited to, the following:

- Authentication verification is built in.
- Communications can be secure.
- Store and forward communication is performed in near real time.
- Communications can be two-way and session-aware.
- Mobile devices such as cell phones already include support; there is nothing to install on the client.

The following sections provide a very brief technical introduction to SMS and the procedures required to develop and implement an SMS ColdFusion application:

- About SMS
- About SMS application development and deployment
- How the SMS event gateway and provider SMSC interact

About SMS

The following discussion simplifies SMS technology and describes only a typical use with a ColdFusion application. For a more complete discussion of SMS, see the publicly available literature, including the several books that discuss SMS.

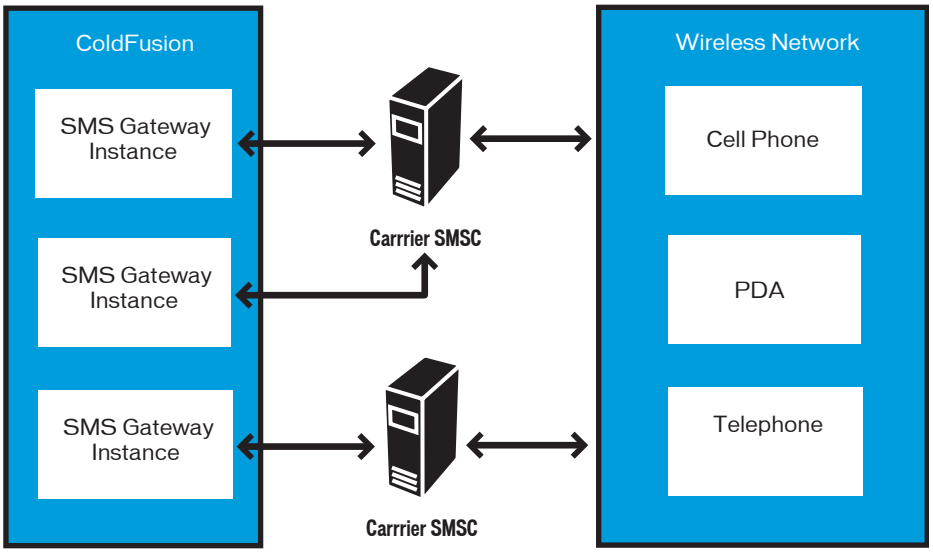
In a ColdFusion SMS application, a mobile device such as a mobile phone communicates (via intermediate steps) with a message center, such as a short message service center (SMSC). For example, a mobile phone user calls a telephone number that the SMS provider has associated with your account; the SMSC gets the messages that are sent to this number. The SMSC can store and forward messages. A ColdFusion application can initiate messages to wireless devices, or it can respond to incoming messages from the devices.

The SMSC communicates with a ColdFusion SMS event gateway using short message peer-to-peer protocol (SMPP) over TCP/IP. Information is transferred by exchanging Protocol Data Units (PDUs) with structures that depend on the type of transaction, such as a normal message submission, a binary data submission, or a message intended for multiple recipients.

Because the SMSC is a store-and-forward server, it can hold messages that cannot be immediately delivered and try to deliver them when the receiving device is available. The time that a message is held on the server for delivery is configured by the SMSC provider. For example, AT&T Wireless saves messages for 72 hours; after that time, any undelivered messages are deleted. Your messages can request a different timeout (by specifying a `ValidityPeriod` field). The message can also use a `registeredDelivery` field to tell the SMSC to inform the you about whether and when the message is delivered.

SMS communication can be secure. Voice and data communications, including SMS message traffic between the SMSC and the mobile device is encrypted as part of the GSM standard. The mobile user's identity is also authenticated by the SMSC before the encrypted communication session begins. You must secure the communications between ColdFusion and the SMSC. Typically, you do this by using a secure hardware or software VPN connection around the SMPP connection.

The following figure shows the SMS path between mobile devices and ColdFusion gateways:



Using the SMS event gateway, ColdFusion establishes a two-way (transceiver) connection to the SMSC of the telecommunications carrier or other SMPP account provider. The SMPP provider assigns an address for your account, and you associate an event gateway instance with the address. Addresses are normally telephone numbers, but carriers often support “short code” addresses that do not require a full 10-digit number. You also configure the gateway instance to communicate with the provider’s specified TCP/IP address using a system ID and a password.

Note: The ColdFusion MX SMS event gateway conforms to the SMPP 3.4 specification, which you can download from the SMS Forum at www.smsforum.net/.

A ColdFusion application can initiate and send messages to SMS-enabled devices by specifying the destination mobile device telephone number, or mobile devices can send messages to a ColdFusion listener CFC by using the gateway instance’s telephone number. Incoming messages include the sender’s number, so listener CFCs can respond to messages sent by mobile devices.

About SMS application development and deployment

The following sections introduce the ColdFusion SMS application development tools and process, and discuss SMS messaging providers:

- ColdFusion MX SMS application tools
- Application development and deployment process
- About SMS providers

ColdFusion MX SMS application tools

ColdFusion provides the following tools for developing SMS applications:

SMSGateway The class for the SMS event gateway type

SMS test server A lightweight SMSC simulator

SMS client simulator A graphical interface for sending and receiving SMS messages with the SMS test server

You implement your SMS application by creating a ColdFusion application that uses an instance of the SMSGateway class to communicate with one or more SMSCs. You can use the SMS testing server and client simulator to test your application without requiring an outside SMS service provider.

Application development and deployment process

The following is a typical process for developing and deploying an SMS application::

1. Design your application.
2. Configure an SMS event gateway instance to use the ColdFusion MX SMS test server.
3. Write your ColdFusion CFCs, CFM pages, and any other application elements.
4. Test your application using the test server and client simulator.
5. Establish an SMPP account with a telecommunications provider.
6. Reconfigure your event gateway, or create a new event gateway instance, to use your telecommunications provider's SMSC. Configure the gateway using the information supplied by your provider.
7. Test your application using the telecommunications provider's SMSC and target mobile devices.
8. Make the application publicly available.

About SMS providers

Before you can deploy an SMS application, you must establish an account with a provider that supports SMPP 3.4 over TCP/IP. There are generally two kinds of providers:

- Telecommunications carriers such as nation-wide cellular phone providers
- Third-party SMPP aggregators

The type of provider and specific provider you use should depend on your needs and provider capabilities and price structures. Less expensive providers may have slower response times. Telecommunications carriers might be more expensive but might provide more throughput and faster SMPP response times.

How the SMS event gateway and provider SMSC interact

This section provides a brief overview of the interactions between the ColdFusion MX SMS event gateway and the SMPP provider's SMSC. It is designed to help you to understand the basics of SMPP interactions, and defines the terms necessary to for you to understand gateway configuration and message handling. For more details, see the SMPP specification, which is available at www.smsforum.net/.

A typical interaction between an SMSC and a ColdFusion SMS event gateway instance consists of messages, or PDUs sent between the two entities, such as a mobile device and a ColdFusion MX event gateway instance (and therefore, and event gateway application). The following sections describe these interactions and how you handle them in ColdFusion MX.

Gateway binding

The event gateway must *bind* to the SMSC before they can communicate. The SMS event gateway instance initiates a binding by sending a bind_transceiver PDU to the SMSC, which includes the gateway's ID and password. If the initial bind request fails, the gateway retries the bind at the rate specified by the gateway configuration file retry-interval value until either the bind is successful or the gateway reaches the maximum number of retries, specified by the retries configuration value. If the bind operation fails, ColdFusion logs an error to the eventgateway.log file, and you must restart the gateway instance in the ColdFusion MX Administrator to establish the connection.

Note: Some SMSCs can send a prohibited status in response to a bind request. If the gateway receives such a status response, it sets the retry interval to one minute and the maximum number of retries to 15. The SMS gateway detects SMPP 5.0-compliant and AT&T prohibited status responses.

When the SMSC accepts the bind request, it returns a bind_transceiver_resp PDU. The binding remains in effect until the gateway instance shuts down and sends an unbind PDU to the SMSC. Because the gateway binds as a transceiver, it can initiate messages to the SMSC, and the SMSC can send messages to it.

Incoming PDU handling

If the ColdFusion SMS event gateway gets an Unbind PDU from the SMSC, it sends an unbind_resp PDU to the SMSC, does a restart, and attempts to rebind to the SMSC.

When the event gateway receives an EnquireLink or any other request PDU from the SMSC, it sends a default response to the SMSC.

The gateway receives incoming messages from the SMSC in deliver _sm PDUs; it does not handle data_sm PDUs. Deliver_sm PDUs can contain user- or application-generated messages, or disposition responses for messages that the gateway has sent. The gateway extracts the short message field and source and destination addresses from the PDU, puts them in a CFEvent object, and sends the object to ColdFusion event gateway services for delivery to the listener CFC. For information on how the CFML application must handle these incoming messages, see "Handling incoming messages" on page 1073.

Outgoing message handling

The gateway supports three types of outgoing messages from ColdFusion applications. The CFML `sendGatewayMessage` function or a listener CFC method `cfreturn` tag can specify the following commands:

submit Sends a submit_sm PDU with the message contents to the SMSC. This PDU sends a message to a single destination.

submitMulti Sends a submit_multi PDU with the message contents to the SMSC. This PDU sends a message to multiple destinations.

data Sends a data_sm PDU with the message contents to the SMSC. This is an alternative to the submit command, and is typically used by interactive applications such as those provided via a wireless application protocol (WAP) framework.

The SMS gateway lets you control the contents of all of the fields of these PDUs. For more information, see the documentation for the individual commands in "Sending outgoing messages" on page 1075.

When you send a message, if the SMSC responds with a status that indicates that the message was rejected or not sent, ColdFusion MX logs information about the failure in the eventgateway.log file. If the SMSC indicates that the service type is not available (SMPP v5 ESME_RSERTYPUNAVAIL status or AT&T Serviced denied status), and the gateway configuration file transient-retry value is set to yes, the gateway also tries to resend the message.

Outgoing message synchronization and notification

The gateway and SMSC communicate asynchronously: the gateway does not wait for a response from the SMSC for one message before it sends another message. However, you can configure your gateway instance so that the CFML `sendGatewayMessage` function behaves asynchronously or synchronously.

- In *asynchronous mode*, the function returns when the message is queued in ColdFusion gateway services.
- In *synchronous mode*, the function waits until the SMSC receives the message and returns a message ID, or an error occurs.

For more information on configuring message synchronization and sending messages synchronously, see "Controlling SMS message sending and response" on page 1078.

Configuring an SMS event gateway

You provide SMS-specific configuration information to the SMS event gateway in a configuration file. You specify the configuration file location when you configure the SMS event gateway instance in the ColdFusion MX Administrator. ColdFusion provides a sample SMS event gateway configuration file in *cf_root*\WEB-INF\cfusion\gateway\config\sms-test.cfg on J2EE configurations, and *cf_root*\gateway\config\sms-test.cfg on server configurations. The configuration file contents is describe in the following table.

Note: The following configuration information describes the configuration fields, but does not include detailed explanations of SMPP-specific terminology, listings of all valid values of properties that are defined in the SMPP specification, or explanations of how to select appropriate SMPP-specific values for your application. For further information, see documentation on the SMPP 3.4 protocol at www.smsforum.net/ and other publicly available documentation.

Note: Your SMS service provider might specify requirements for several of these configuration values. Consult the provider documentation.

Property	Default	Description
ip-address		IP address of the SMSC, as specified by the SMPP provider. For the ColdFusion SMS test server, you normally use 127.0.0.1.
port	0	Port number to bind to on the SMSC. The ColdFusion SMS test server uses port 7901.
system-id		Name that identifies the event gateway to the SMSC, as established with the SMPP provider. To connect to the ColdFusion SMS test server, the system-id must be cf.
password		Password for authenticating the event gateway to the SMSC. To connect to the ColdFusion SMS test server, the password must be cf.
source-ton	1	Type of Number (TON) of the source address, that is, of the address that the event gateway uses for outgoing messages, as specified in the SMPP specification. Values include 0, unknown; 1, international number; 2, national number.
source-npi	1	Numeric Plan Indicator (NPI) of the source address as specified in the SMPP specification. Values include 0, unknown; 1, ISDN.
source-address	empty string	Address (normally, a phone number) of the event gateway. Identifies the sender of outgoing messages to the SMSC.
addr-ton	1	TON for the incoming addresses that this event gateway serves.
addr-npi	1	NPI for the incoming addresses that this event gateway serves.

Property	Default	Description
address-range		The range of incoming addresses (phone numbers) that remote devices can use to send messages to the event gateway instance; often, the same as the `source-address`.
message-rate	100	Integer or decimal value that specifies the number of messages the gateway is allowed to send to your service provider per second. 0 is unlimited.
mode	synchronous	Message transmission mode: • synchronous The gateway waits for the response from the server when sending a message. In this mode, the `SendGatewayMessage` CFML function returns the SMS `messageID` of the message, or an empty string if there is an error. • asynchronous The gateway does not wait for a response. In this mode, the `SendGatewayMessage` CFML function always returns an empty string.
network-retry	no	Gateway behavior when a network error occurs while trying to deliver a message: • yes The gateway queues the message for delivery when the gateway is able to rebind to the SMSC. Retrying is useful if the gateway is in asynchronous mode, where the CFML `SendGatewayMessage` function does not return an error. • no The gateway does not retry sending the message.
transient-retry	no	Gateway behavior when the SMSC returns an error that indicates a transient error, where it may be able to accept the message in the future: • yes The gateway attempts to resend the message. Retrying is useful if the gateway is in asynchronous mode, where the CFML `SendGatewayMessage` function does not return an error. • no The gateway does not retry sending the message.
cfc-method	onIncomingMessage	Listener CFC method for ColdFusion to invoke when the gateway gets incoming messages.
destination-ton	1	Default TON of addresses for outgoing messages.
destination-npi	1	Default NPI of addresses for outgoing messages.
service-type	empty string	Type of messaging service; can be empty or one of the following values: CMT, CPT, VMN, VMA, WAP, or USSD.
system-type	empty string	Type of system (ESME, External Short Message Entity); used when binding to the SMSC. Some SMSCs might be able to send responses that are specific to a given type of ESME. Normally, should be set to SMPP.
receive-timeout	-1 (do not time out)	The time-out, in seconds, for trying to receive a message from the SMSC after it establishes a connection. To wait indefinitely until a message is received, set the `receive-timeout` to -1.

Property	Default	Description
ping-interval	60	Number of seconds between EnquireLink messages that the event gateway sends to the server to verify the health of the connection.
retries	-1 (try forever)	Number of times to retry connecting to the SMSC to send a message before the gateway goes into a failed state. If the gateway is in a failed state, the `getStatus` method returns FAILED, and theColdFusion MX Administrator shows the gateway status as Failed. The gateway must be restarted before it can be used.
retry-interval	10	Number of seconds between connection retries.

You can also set the following values in each outgoing message: source-ton, source-npi, source-address, destination-ton, destination-npi, and service-type. The message field names differ from the configuration file property names.

Handling incoming messages

The SMS event gateway handles messages that are contained in deliver_sm PDUs. These PDUs request the gateway to deliver one of the following types of message:

- A user- or application-generated text message
- A message disposition response

Note: The SMS event gateway does not handle messages that are contained in data_sm PDUs.

The event gateway sends the object to event gateway services, which delivers it to the listener CFC. The CFEvent object that the listener CFC receives contains the following fields:

Field	Value
CfcMethod	Listener CFC method name
Data.dataCoding	Character set or the noncharacter data type of the message contents
Data.destAddress	Address to which the message was sent
Data.esmClass	Message type
Data.MESSAGE	Message contents
Data.messageLength	Length of the MESSAGE field
Data.priority	Message priority level, in the range 0-3
Data.protocol	GSM protocol; not used for other networks
Data.registeredDelivery	Requested type of delivery receipt or acknowledgement, if any
Data.sourceAddress	Address of the device that sent this message
GatewayType	Always SMS
OriginatorID	Address of the device that sent the message

For a detailed description of each field, see "SMS Gateway incoming message CFEvent structure" in *CFML Reference*.

The CFC's listener method extracts the message from the Arguments.CFEvent.Data.MESSAGE field and acts on it as appropriate for the application. If necessary, the listener can use two fields to determine the required action:

- CFEvent.Data.esmClass indicates the type of information in the MESSAGE field.
- CFEvent.Data.registeredDelivery indicates whether the sender requested any type of delivery receipt or acknowledgement.

CFEvent.Data.esmClass field

The CFEvent.Data.esmClass field identifies whether the CFEvent.Data.Message field contains a message, or any of the following types of message disposition responses. For these responses, the CFEvent object Data.MESSAGE field contains the acknowledgment or receipt information.

SMSC Delivery Receipt An indication of the message's final status, sent by the SMSC. The short message text includes the message ID of the original message, the number of messages sent and delivered (for messages sent to a distribution list), the date and time that the message was sent and delivered or otherwise disposed of, the message disposition, a network-specific error code (if available), and the first 20 bytes of the message. For details of the SMSC delivery receipt message structure, see Appendix B of the SMS 3.4 specification.

SME Delivery Acknowledgement An indication from the recipient device that the user has read the short message. Supported by TDMA and CDMA wireless networks only.

SME Manual/User Acknowledgement An application-generated reply message sent in response to an application request message. Supported by TDMA and CDMA wireless networks only.

Intermediate Delivery Notification A provider-specific notification on the status of a message that has not yet been delivered, sent during the SMSC retry lifetime for the message. Intermediate Notification support depends on the SMSC implementation and SMSC service provider. For more information, see your provider documentation.

When you send a message, you can request any combination of message disposition responses in the outgoing message's `registered_delivery` parameter. If your application requests responses, the listener CFC must be prepared to handle these messages, as appropriate.

CFEvent.Data.registeredDelivery field

The CFEvent.Data.registeredDelivery field indicates whether the message sender has requested a receipt or acknowledgment. Your application can respond to a request for an SME Delivery Acknowledgement or an SME Manual/User Acknowledgement. (The other notification types are sent by the SMSC only.) For more information on these notification types, see the SMS 3.4 specification. Appendix B contains detailed information on the information that you must put in the shortMessage field of the returned acknowledgment message.

Incoming message handling example

The following example code is an SMS-only version of the echo.cfc example that is included in the ColdFusion MX gateway/cfc/examples directory. This example shows the minimal code needed to receive and respond to an SMS message.

```
<cfcomponent displayname="echo" hint="Process events from the test gateway and
  return echo">
<cffunction name="onIncomingMessage" output="no">
  <cfargument name="CFEvent" type="struct" required="yes">
  <!--- Get the message --->
  <cfset data=cfevent.DATA>
  <cfset message="#data.message#">
  <!--- where did it come from? --->
  <cfset orig="#CFEvent.originatorID#">
  <cfset retValue = structNew()>
  <cfset retValue.command = "submit">
  <cfset retValue.sourceAddress = arguments.CFEVENT.gatewayid>
  <cfset retValue.destAddress = arguments.CFEVENT.originatorid>
  <cfset retValue.shortMessage = "echo: " & message>
  <!--- send the return message back --->
  <cfreturn retValue>
</cffunction>
</cfcomponent>
```

Sending outgoing messages

Your ColdFusion application can send `submit`, `submitMulti`, and `data` commands to the event gateway in an outgoing message. The following sections describe these commands in detail, followed by information on some of the more common options for sending messages.

The submit command

To send a message to a single destination address in an SMPP SUBMIT_SM PDU, the structure used in the *Data* parameter of a `SendGatewayMessage` function or the return variable of the CFC listener method normally has the following fields:

Field	Contents
command	If present, the value must be `"submit"`. If you omit this field, the event gateway sends a submit message.
shortMessage *or* messagePayload	The Message contents. You must specify one of these fields, but not both. The SMPP specification imposes a maximum size of 254 bytes on the `shortMessage` field, and some carriers might limit its size further. The `messagePayload` field can contain up to 64K bytes; it must start with 0x0424, followed by 2 bytes specifying the payload length, followed by the message contents.
destAddress	The address to which to send the message (required).
sourceAddress	The address of this application. You can omit this field if it is specified in the configuration file.

You can also set optional fields in the structure, such as a field that requests a delivery receipt. For a complete list of fields, see "submit command" in *CFML Reference*. For detailed descriptions of these fields, see the documentation for the SUBMIT_MULTI PDU in the SMPP3.4 specification, which you can download from the SMS Forum at www.smsforum.net/.

Note: To send long messages, you can separate the message into multiple chunks and use a submit command to send each chunk separately. In this case, a CFC would use multiple `SendGatewayMessage` functions, instead of the `cfreturn` function.

Example: Using the submit command in sendGatewayMessage function

The following example from a CFM page uses a `sendGatewyMessage` CFML function with a `submit` command to send an SMS messages that you enter in the form. This example uses the SMS gateway that is configured in the ColdFusion MX installation, and will send the message to the SMS client simulator.

```
<h3>Sending SMS From a Web Page Example</h3>
<cfif IsDefined("form.oncethrough") is "Yes">
    <cfif IsDefined("form.SMSMessage") is True AND form.SMSMessage is not "">
      <h3>Sending Text Message: </h3>
     <cfoutput>#form.SMSMessage#</cfoutput><br>
     <cfscript>
       /* Create a structure that contains the message. */
       msg = structNew();
       msg.command = "submit";
       msg.destAddress = "5551234";
       msg.shortMessage = form.SMSMessage;
       ret = sendGatewayMessage("SMS Menu App - 5551212", msg);
     </cfscript>
    </cfif>
    <hr noshade>
</cfif>
<!--- begin by calling the cfform tag --->
<cfform action="command.cfm" method="POST">
  SMS Text Message: <cfinput type="Text" name="SMSMessage" value="Sample text
  Message" required="No" maxlength="160">
<p><input type = "submit" name = "submit" value = "Submit">
<input type = "hidden" name = "oncethrough" value = "Yes">
</cfform>
</body>
</html>
```

For a simple example of a listener CFC uses the `submit` command to echo incoming SMS messages to the message originator, see "Incoming message handling example" on page 1075.

The submitMulti command

To send a single text message to multiple recipients using an SMPP SUBMIT_MULTI PDU, the *Data* parameter of a `SendGatewayMessage` function or the return variable of the CFC listener method normally has the following fields:

Field	Contents
command	Must be `"submitMulti"`.
shortMessage *or* messagePayload	The message contents. You must specify one of these fields, but not both. The SMPP specification imposes a maximum size of 254 bytes on the `shortMessage` field, and some carriers might limit its size further. The `messagePayload` field can contain up to 64K bytes; it must start with 0x0424, followed by 2 bytes specifying the payload length, followed by the message contents.
destAddress	A ColdFusion array of destination addresses (required). You cannot specify individual TON and NPI values for these addresses; all must conform to a single setting.
sourceAddress	The address of this application; you can omit this field if it is specified in the configuration file.

You can also set optional fields in the structure, such as a field that requests delivery receipts. For a complete list of fields, see "submitMulti command" in *CFML Reference*. For detailed descriptions of these fields, see the documentation for the SUBMIT_MULTI PDU in the SMPP 3.4 specification, which you can download from the SMS Forum at www.smsforum.net/.

Example: Using the submitMulti command in an onIncomingMessage method

The following example `onIncomingMessage` method sends a response that echoes an incoming message to the originator address, and sends a copy of the response to a second address. To test the example, run two instances of the ColdFusion SMS client application. Use the default phone number of 5551212 for the first, and set the second one to have a phone number of 555-1235. (Notice that the second phone number requires a hyphen (-).) Send a message from the first simulator, and the response will appear in both windows.

```
<cffunction name="onIncomingMessage" output="no">
  <cfargument name="CFEvent" type="struct" required="yes">
  <!--- Get the message. --->
  <cfset data=CFEvent.DATA>
  <cfset message="#data.message#">
  <!--- Create the return structure. --->
  <cfset retValue = structNew()>
  <cfset retValue.command = "submitmulti">
  <cfset retValue.sourceAddress = arguments.CFEVENT.gatewayid>
  <cfset retValue.destAddresses=arraynew(1)>
  <!--- One destination is incoming message originator;
          get the address from CFEvent originator ID. --->
  <cfset retValue.destAddresses[1] = arguments.CFEvent.originatorid>
  <cfset retValue.destAddresses[2] = "555-1235">
  <cfset retValue.shortMessage = "echo: " & message>
<cfreturn retValue>
</cffunction>
</cffunction>
```

The data command

To send binary data to a single destination address in an SMPP DATA_SM PDU, the *Data* parameter of a `SendGatewayMessage` function or the return variable of the CFC listener method must have the following fields:

Field	Contents
command	Must be `"data"`.
messagePayload	Message data. To convert data to binary format, use the ColdFusion MX `toBinary` function.
destAddress	Address to which to send the message.
sourceAddress	Address of this application; can be omitted if specified in the configuration file.

You can also set optional fields in the structure, such as a field that requests a delivery receipt. For a complete list of fields, see "data command" in *CFML Reference*. For detailed descriptions of these fields, see the documentation for the SUBMIT_MULTI PDU in the SMPP3.4 specification, which you can download from the SMS Forum at www.smsforum.net/.

Example: Using the data command

The following example `onIncomingMessage` method converts an incoming message to binary data, and sends the binary version of the message back to the originator address:

```
<cffunction name="onIncomingMessage" output="no">
  <cfargument name="CFEvent" type="struct" required="yes">
  <!--- Get the message. --->
  <cfset data=CFEvent.DATA>
  <cfset message="#data.message#">
  <!--- Create the return structure. --->
  <cfset retValue = structNew()>
  <cfset retValue.command = "data">
  <!--- Sending to incoming message originator; get value from CFEvent. --->
  <cfset retValue.destAddress = arguments.CFEvent.originatorid>
  <cfset retValue.messagePayload = tobinary(tobase64("echo: " & message))>
  <cfreturn retValue>
</cffunction>
```

Controlling SMS message sending and response

This section describes some of the more common options for sending messages, and how they affect your application. For information on other ways to configure outgoing message, see the SMPP specification.

This section describes the following topics:

- Synchronization mode
- Message disposition notification
- Validity period

Synchronization mode

You can specify asynchronous or synchronous message mode in the gateway configuration file.

- If you specify *asynchronous* mode, the `sendGatewayMessage` function returns an empty string when the message is submitted by the gateway to service code for sending to the SMSC. ColdFusion logs errors that might occur after this point, such as if a message sent by the gateway to the SMSC times out or if the gateway gets an error response; the application does not get notified of any errors.
- If you specify *synchronous* mode (the default), the `sendGatewayMessage` function does not return until the gateway gets a response from the SMSC or the attempt to communicate times out. If the message is sent successfully, the function returns the SMPP message ID string. If an error occurs, the function returns an error string.

Use synchronous mode if your application must determine whether its messages reach the SMSC. Also use synchronous mode if the application requests return receipts.

Note: If you use synchronous mode and the SMSC returns the messgeID as a hexadecimal string, ColdFusion MX converts it automatically to its decimal value.

The following example is an expansion of "Example: Using the submit command in sendGatewayMessage function" on page 1076. It checks for a nonempty return value and displays the message number returned by the SMS. This example uses the SMS gateway that is configured when ColdFusion MX is installed. If you change the gateway specified in the `SendGatewayMessage` function, make sure that you gateway's configuration file specifies synchronous mode.

```
<h3>Sending SMS From a Web Page Example</h3>

<cfif IsDefined("form.oncethrough") is "Yes">
  <cfif IsDefined("form.SMSMessage") is True AND form.SMSMessage is not "">
     <h3>Sending a Text Message: </h3>
      <cfoutput>#form.SMSMessage#</cfoutput><br><br>

    <cfscript>
      /* Create a structure that contains the message. */
      msg = structNew();
      msg.command = "submit";
      msg.destAddress = "5551234";
      msg.shortMessage = form.SMSMessage;
      ret = sendGatewayMessage("SMS Menu App - 5551212", msg);
    </cfscript>
  </cfif>
  <cfif isDefined("ret") AND ret NEQ "">
    <h3>Text message sent</h3>
    <cfoutput>The Message Id is: #ret#</cfoutput>
    <br><br>
  </cfif>
  <hr noshade>
</cfif>

<!--- begin by calling the cfform tag --->
<cfform>
```

```
  SMS Text Message: <cfinput type="Text" name="SMSMessage"
    value="Sample text Message" required="No" maxlength="160">
  <p><input type = "submit" name = "submit" value = "Submit">
  <input type = "hidden" name = "oncethrough" value = "Yes">
</cfform>
```

Message disposition notification

You can request the SMSC to return a message disposition response to indicate the fate of your message. To request a delivery receipt, include a RegisteredDelivery field in the *Data* parameter of a `SendGatewayMessage` function or the return variable of the CFC listener method. This field can have the following values:

Value	Meaning
0	(Default) Do not return delivery information.
1	Return a receipt if the message is not delivered before the time-out.
2	Return a receipt if the message is delivered or fails.

Some providers also support intermediate delivery notifications. For more information, see your provider's documentation.

To use delivery notification, you must send your message using synchronous mode, so you get a message ID. Your incoming message routine must be able to handle the receipts as described in "Handling incoming messages" on page 1073

Validity period

You can change the length of time that the SMSC keeps a message and tries to deliver it. (Often the default value is 72 hours.) For a message sent to an emergency worker, for example, you might want to specify a very short validity period (such as 15 minutes). To change this value, include a validityPeriod field in the *Data* parameter of a `SendGatewayMessage` function or the return variable of the CFC listener method. To specify a time period, use the following pattern: *YYMMDDhhmmsst*00R. In this pattern, *t* indicates tenths of seconds, and 00R specifies that this is a relative time period, not a date-time value. The time format 00001063000000R, for example, specifies a validity period of 0 years, 0 months, 1 day, 6 hours, 30 minutes.

ColdFusion MX SMS development tools

ColdFusion MX provides the following tools for developing SMS applications:

- SMS test server
- SMS client simulator

SMS test server

The CColdFusion MX SMS test server is a lightweight SMSC simulator that listens on TCP/IP port 7901 for SMPP connection requests from other SMS resources, such as ColdFusion MX SMS gateways or the SMS client simulator. The resource supplies a user name, password, and telephone number (address). The user name and password must correspond to a name and password in the simulator's configuration file (described later in this section).

After the SMS test server establishes a connection, it listens for incoming messages and forwards them to the specified destination address, if the destination address also corresponds to an existing SMPP connection.

The SMS test server lets you develop SMS applications without having to use an external SMSC supplier such as a telecommunications provider. The server supports the ColdFusion SMS gateway `submit` and `submitMulti` commands. It also accepts, but does not deliver messages sent using the SMS gateway `data` command. It does not include any store and forward capabilities.

Start the SMS test server by clicking the Start SMS Test Server button on the Settings page in the Event Gateways area in the ColdFusion MX Administrator.

Note: The SMS test server does not automatically restart when you restart ColdFusion MX. You must manually restart the server if you restart ColdFusion MX.

The SMS test server reads the *cf_root*\WEB-INF\cfusion\lib\sms-test-users.txt file on J2EE configurations or *cf_root*\lib\sms-test-users.txt file on server configurations to get valid user names and passwords. ColdFusion MX includes a version of this file configured with several names and passwords. One valid combination is user name cf and password cf. You can edit this file to add or delete entries. The file must include a name and password entry for each user that will connect to the test server, and user entries must be separated by blank lines, as the following example shows:

```
name=cf
password=cf

name=user1
password=user1
```

SMS client simulator

The ColdFusion SMS client simulator is a simple External Short Message Entity (ESME) that simulates a (limited-function) mobile phone. It can connect to the SMS test server and exchange messages with it.

Note: On UNIX and Linux systems, the client simulator requires X-Windows.

Use the following procedure to use the simulator.

To use the SMS simulator:

1. Ensure that you have started the SMS test server and configured and started an SMS event gateway instance in ColdFusion MX Administrator.
2. Run SMSClient.bat in Windows or SMSClient.sh on UNIX or Linux. These files are located in the *cf_root*\WEB-INF\cfusion\bin directory on J2EE configurations and the *cf_root*\bin directory on server configurations.

   If you installed a pure Java version of ColdFusion MX, for example, on Apple OS X systems, enter the following command to start the simulator:

   ```
   java -jar cf_root/WEB-INF/cfusion/lib/smpp.jar
   ```
3. A dialog box appears, requesting the server, port, username, password, and the phone number to use for this device. The simulator sends this phone number as the source address, and accepts SMS messages sent by the SMSC server to it using this number as the destination address.

   To connect to the SMS test server, accept the default values and specify an arbitrary phone number; you can also specify any username-password pair that is configured in the *cf_root*\WEB-INF\cfusion\lib\sms-test-users.cfg file or *cf_root*\lib\sms-test-users.cfg or file.
4. Click Connect.
5. The SMS device simulator client appears. In the Send SMS To field, enter a phone number in the address-range property specified in the configuration file of the SMS event gateway that you want to send messages to.
6. Type a message directly into the message field (to the left of the Send button), or use the simulator keypad to enter the message.
7. Click the Send button.

The client simulator has a Connection menu with options to connect and disconnect from the SMSC server, and to check the connection. The connection information appears in a status line at the bottom of the client.

Sample SMS application

The following CFC implements a simple employee phone directory lookup application. The user sends an message containing some part of the name to be looked up (a space requests all names). The `onIncomingMessage` response depends on the number matches.

- If there is no match, the `onIncomingMessage` function returns a message indicating that there are no matches.
- If there is one match, the function returns the name, department, and phone number.
- If there are up to ten matches, the function returns a list of the names preceded by a number that the user can enter to get the detailed information.
- If there are over ten matches, the function returns a list of only the first ten names. A more complex application might let the user get multiple lists of messages to provide access to all names.
- If the user enters a number, and previously got a multiple-match list, the application returns the information for the name that corresponds to the number.

The following listing shows the CFC code:

```
<cfcomponent>
  <cffunction name="onIncomingMessage">
    <cfargument name="CFEvent" type="struct" required="YES">
    <!--- Remove any extra white space from the message. --->
    <cfset message =  Trim(arguments.CFEvent.data.MESSAGE)>
    <!--- If the message is numeric, a previous search probably returned a
      list of names. Get the name to search for from the name list stored in
      the Session scope. --->
    <cfif isNumeric(message)>
      <cfscript>
        if (structKeyExists(session.users, val(message))) {
          message = session.users[val(message)];
        }
      </cfscript>
    </cfif>

    <!--- Search the database for the requested name. --->
    <cfquery name="employees" datasource="cfdocexamples">
      select FirstName, LastName, Department, Phone
      from Employees
      where 0 = 0
      <!--- A space indicates the user entered a first and last name. --->
      <cfif listlen(message, " ") eq 2>
        and FirstName like '#listFirst(message, " ")#%'
        and LastName like '#listlast(message, " ")#%'
      <!--- No space: the user entered a first or a last name. --->
      <cfelse>
        and (FirstName like '#listFirst(message, " ")#%'
        or LastName like '#listFirst(message, " ")#%')
      </cfif>
    </cfquery>

    <!--- Generate and  return the message.--->
    <cfscript>
      returnVal = structNew();
      returnVal.command = "submit";
      returnVal.sourceAddress = arguments.CFEVENT.gatewayid;
      returnVal.destAddress = arguments.CFEVENT.originatorid;

      //No records were found.
      if (employees.recordCount eq 0) {
        returnVal.shortMessage = "No records found for '#message#'";
      }
      //One record was found.
      else if (employees.recordCount eq 1) {
      // Whitespace in the message text results in bad formatting,
      // so the source cannot be indented.
        returnVal.shortMessage = "Requested information:
#employees.firstName# #employees.lastName#
#employees.Department#
#employees.Phone#";
      }
      //Multiple possibilities were found.
```

```
        else if (employees.recordCount gt 1) {
          //If more than ten were found, return only the first ten.
          if (employees.recordCount gt 10)
          {
            returnVal.shortMessage = "First 10 of #employees.recordCount#
    records";
          }else{
            returnVal.shortMessage = "Records found: #employees.recordCount#";
          }
          // The session.users structure contains the found names.
          // The record key is a number that is also returned in front of the
          // name in the message.
          session.users = structNew();
          for(i=1; i lte min(10, employees.recordCount); i=i+1)
          {
            // These two lines are formatted to prevent extra white space.
            returnVal.shortMessage = returnVal.shortMessage & "
#i# - #employees.firstName[i]# #employees.lastName[i]#";
            // The following two lines must be a single line in the source
            session.users[i]="#employees.firstName[i]#
    #employees.lastName[i]#";
          }
        }
        return returnVal;
      </cfscript>
    </cffunction>
</cfcomponent>
```

CHAPTER 45
Creating Custom Event Gateways

This chapter describes how to create event gateways. It assumes you have a thorough knowledge of Java programming, including Java event-handling and thread-handling concepts, and of the technology to which you are providing the gateway, including the types of messages that you will handle. It also assumes that you have a thorough knowledge of Macromedia ColdFusion development concepts and practices, including ColdFusion components (CFCs).

Note: ColdFusion MX Standard Edition does not support event gateways.

Contents

Event gateway architecture 1085

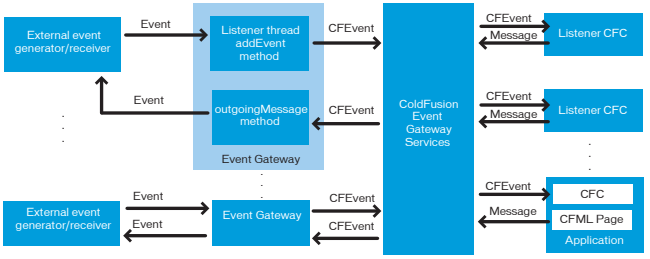
Event gateway elements 1086

Building an event gateway 1092

Deploying an event gateway 1100

Event gateway architecture

A ColdFusion event gateway listens for events and passes them to ColdFusion for handling by the application's listener CFC or CFCs. It must implement the coldfusion.eventgateway.Gateway interface, and use the ColdFusion GatewayServices class. The following figure expands on the basic event handling architecture diagram to show how a ColdFusion event gateway works:



Receiving messages The event gateway listener thread receives events from an external event source such as a socket or SMSC server, and calls the GatewayServices `addEvent` method to send a `CFEvent` instance to ColdFusion.

Sending messages The ColdFusion event gateway service calls the event gateway's `outgoingMessage` method and passes it a CFEvent instance with the destination and message information. The event gateway forwards the message as appropriate to the external receiver.

The event gateway architecture is not limited to handling messages from external sources, such as SMS devices or IM clients. It can also be used to handle events that are internal to the local system or even the ColdFusion application. Also, a gateway does not have to implement two-way communications.

The sample directory watcher gateway provided with ColdFusion MX is an example of an internal, one way, gateway. It has a single thread that periodically checks a local directory and sends a message to a CFC when the directory contents change. This gateway does not support outgoing messages. (The code for this gateway is in the gateway/src/examples/watcher directory.)

Another internal gateway, the asynchronous CFML gateway, is provided as part of the ColdFusion product. Unlike most gateways, it does not have a listener thread. Its `outgoingMessage` method gets messages from CFML `SendGatewayMessage` functions, and dispatches them to a CFC `onIncomingMessage` method for handling. This gateway lets ColdFusion support request-free asynchronous processing. For more information on using this gateway, see "Using the CFML event gateway for asynchronous CFCs" on page 1038.

Event gateway elements

The following sections describe the elements that you use to create and configure a gateway:

- Gateway interface
- GatewayServices class
- CFEvent class
- GatewayHelper class
- Gateway configuration file
- Gateway development classes

Note: The gateway interfaces and classes, with the exception of the GenericGateway class are fully documented in the "Gateway development interfaces and classes" section in *CFML Reference*. All interfaces and classes in this list, including the GenericGateway class, are documented in less detail in the Javadocs located in the ColdFusion MX gateways\docs directory. The Javadocs documentation lacks examples and does not have the detailed usage information that you find in *CFML Reference*.

Gateway interface

The ColdFusion event gateway must implement the coldfusion.eventservice.Gateway interface. The following table lists the interface method signatures:

Note: For detailed information on implementing each method, see "Building an event gateway" on page 1092. For reference pages for these methods, see "Gateway interface" in *CFML Reference*.

Signature	Description
`void setGatewayID(String id)`	Sets the gateway ID that uniquely identifies the gateway instance. ColdFusion calls this method before it starts the event gateway, even if the gateway class constructor also sets the ID.
`void setCFCListeners(String[] listeners)`	Identifies the CFCs that listen for incoming messages from the event gateway. The array contains one or more paths to the listener CFCs. ColdFusion calls this method before it starts the event gateway, and if the configuration for a running event gateway changes.
`GatewayHelper getHelper()`	Returns a coldfusion.eventgateway.GatewayHelper class instance, or null. The GatewayHelper class provides event gateway-specific utility methods to CFML applications. ColdFusion calls this method when a ColdFusion application calls the `GetGatewayHelper` function.
`String getGatewayID()`	Returns the gateway ID.
`int getStatus()`	Gets the event gateway status. The interface defines the following status constants: `STARTING`, `RUNNING`, `STOPPING`, `STOPPED`, `FAILED`.
`void start()`	Starts the event gateway. Starts at least one thread for processing incoming messages. ColdFusion calls this method when it starts an event gateway.
`void stop()`	Stops the event gateway. Stops the threads and destroys any resources. ColdFusion calls this method when it stops an event gateway.
`void restart()`	Restarts a running event gateway. ColdFusion calls this method when the ColdFusion MX Administrator restarts a running event gateway.
`String outgoingMessage (coldfusion.eventgateway.CFEvent cfmessage)`	Handles a message sent by ColdFusion and processes it as needed by the gateway type to send a message. ColdFusion calls this method when a listener CFC's listener method returns a CFEvent or when a ColdFusion application calls the `SendGatewayMessage` function. The CFML `SendGatewayMessage` function gets the returned String as its return value.

GatewayServices class

The Gateway class uses the coldfusion.eventgateway.GatewayServices class to interact with the ColdFusion event gateway services. This class has the following methods:

Signature	Description
`GatewayServices getGatewayServices()`	Static method that returns the GatewayServices object. Gateway code can call this method at any time, if required.
`boolean addEvent(CFEvent msg)`	Sends a `CFEvent` instance to ColdFusion for dispatching to a listener CFC. The event gateway uses this method to send all incoming messages to the application for processing. Returns False if the event is not added to the queue.
`int getQueueSize()`	Returns the current size of the ColdFusion event queue. This queue handles all messages for all gateways.
`int getMaxQueueSize()`	Returns the maximum size of the ColdFusion event queue, as set in the ColdFusion MX Administrator.
`Logger getLogger()` `Logger getLogger(String logfile)`	Returns a ColdFusion Logger object that the event gateway can use to log information in the eventgateway.log log file (the default) or the specified log file. The *logfile* attribute must be a file name without a file extension, such as mylogifile. ColdFusion MX puts the file in the ColdFusion logs directory and appends .log to the specified file name. For information on using the logger object, see "Logging events and using log files" on page 1099.

CFEvent class

The Gateway class sends and receives CFEvent instances to communicate with the ColdFusion listener CFC or application. The Gateway notifies ColdFusion of a message by sending a CFEvent instance in a `GatewayServices.addEvent` method. Similarly, the Gateway receives a CFEvent instance when ColdFusion calls the gateway's `outgoingMessage` method.

The CFEvent class extends the java.util.Hashtable class and has the following methods to construct the instance and set and get its fields. (In CFML, you treat CFEvent instances as structures.)

Methods	Description
`CFEvent(String gatewayID)`	CFEvent constructor. The *`gatewayID`* parameter must be the value that is passed in the gateway constructor or set using the Gateway `setGatewayID` method.
`void setGatewayType(String type)` `String getGatewayType()`	Identifies the type of event gateway, such as SMS. For the sake of consistency, use this name in the Type Name field when you add an event gateway type on the Gateway Types page in the ColdFusion MX Administrator.

Methods	Description
`void setData(Map data)` `Map getData()`	The event data; includes the message being passed to or from ColdFusion. The content of the field depends on the event gateway type. The Map keys must be strings. Because ColdFusion is not case-sensitive, it converts the Map passed in the `setData` method to a case-insensitive Map. As a result, do not create entries in the data with names that differ only in case.
`void setOriginatorID(String id)` `String getOriginatorID()`	Identifies the originator of an incoming message or the destination of an outgoing message. The value depends on the protocol or event gateway type.
`void setCFCPath(String path)` `String getCFCPath()`	An absolute path to the application listener CFC that will process the event. By default, ColdFusion uses the first path configured for the event gateway instance on the Event Gateways page in the ColdFusion MX Administrator.
`void setCFCMethod(String method)` `String getCFCMethod()`	The method in the listener CFC that ColdFusion calls to process this event. By default, ColdFusion invokes the `onIncomingMessage` method. For the sake of consistency, Macromedia recommends that any event gateway with a single listener not override this default. A gateway, such as the ColdFusion MX XMPP gateway, that uses different listener methods for different message types, uses this method to identify the destination method..
`void setCFCTimeout(String seconds)` `String getCFCTimeout()`	The time-out, in seconds, for the listener CFC to process the event request. When ColdFusion calls the listener CFC to process the event, and the CFC does not process the event in the specified time-out period, ColdFusion terminates the request and logs an error in the application.log file. By default, ColdFusion uses the Timeout Request value set on the Server Settings page in the ColdFusion MX Administrator.
`String getGatewayID()`	The event gateway instance that processes the event. Returns the gateway ID that was set in the CFEvent constructor.

GatewayHelper class

ColdFusion MX includes a coldfusion.eventgateway.GatewayHelper Java marker interface. You implement this interface to define a class that provides gateway-specific utility methods to the ColdFusion application or listener CFC. For example, an instant messaging event gateway might use a helper class to provide buddy list management methods to the application.

The Gateway class must implement a `getHelper` method that returns the helper class or null (if the gateway does not need such a class).

ColdFusion applications call the `GetGatewayHelper` CFML function, which invokes gateway's the `getHelper` method to get an instance of the helper class. The application can then call helper class methods using ColdFusion object dot notation.

The following code defines the SocketHelper class, the gateway helper for the SocketGateway class. It has an empty constructor and two public methods: one returns the socket IDs; the other closes a specified socket. These classes let an application monitor and end session connections.

```
public class SocketHelper implements GatewayHelper {
  public SocketHelper() {
  }
  public coldfusion.runtime.Array getSocketIDs () {
    coldfusion.runtime.Array a = new coldfusion.runtime.Array();
    Enumeration e = socketRegistry.elements();
    while (e.hasMoreElements()) {
      a.add(((SocketServerThread)e.nextElement()).getName());
     }
     return a;
   }
  public boolean killSocket (String socketid) {
    try
    {
      ((SocketServerThread)socketRegistry.get(socketid)).socket.close();
      ((SocketServerThread)socketRegistry.get(socketid)).socket = null;
      socketRegistry.remove(socketid);
      return true;
    }
    catch (IOException e) {
      return false;
    }
  }
}
```

Gateway configuration file

Gateways can use a configuration file to specify information that does not change frequently. For example, the ColdFusion SMS event gateway configuration file contains values that include an IP address, port number, system ID, password, and so on.

You can specify a configuration file path for each event gateway instance in the ColdFusion MX Administrator. ColdFusion passes the file path in the gateway constructor when it instantiates the event gateway. The configuration file format and content handling is up to you. It is the responsibility of the gateway class to parse the file contents and use it meaningfully.

One good way to access and get configuration data is to use the java.util.Properties class. This class takes an ISO8859-1 formatted input stream with one property setting per line. Each property name must be separated from the value by an equal sign (=) or a colon (:), as the following example shows:

ip-address=127.0.0.1
port=4445

The example SocketGateway event gateway uses this technique to get an optional port number from a configuration file. For an example of reading a properties file and using its data, see the code in “Class constructor” on page 1092.

Gateway development classes

ColdFusion provides two classes that you can use as building blocks to develop your event gateway classes. Each corresponds to a different development methodology:

- The coldfusion.eventgateway.GenericGateway class is an abstract class from which you can derive your gateway class.
- The EmptyGateway class in the gateway\src\examples directory is a template gateway that you can complete to create your gateway class.

The GenericGateway class

ColdFusion MX includes a coldfusion.eventgateway.GenericGateway abstract class that implements many of the methods of ColdFusion Gateway interface and provides some convenience methods for use in your gateway class.

You can derive your gateway class from this class, which handles the basic mechanics of implementing a gateway, such as the `getGatewayID` and `SetCFCListeners` methods. Your derived class must implement at least the following methods:

- `startGateway` (not `start`)
- `stopGateway` (not `stop`)
- `outgoingMessage`

Your derived gateway class also must implement the following:

- If you support a configuration file, a constructor that takes a configuration file, and configuration loading routines.
- If you use a gatewayHelper class, the `getHelper` method.
- If the event source status can change asynchronously from the gateway, the `getStatus` method.

The example JMS gateway is derived from the generic gateway class. The gateway class JavaDocs in the gateway\docs directory provide documentation for this class. (The *CFML Reference* does not document this class.)

The EmptyGateway class

The gateway\src\examples\EmptyGateway.java file contains an event gateway template that you can use as a skeleton for creating your own event gateway. (The gateway directory is in the *cf_root* directory in the server configuration and the *cf_root*\WEB-INF-cfusion directory on J2EE configurations.) This file contains minimal versions of all methods in the coldfusion.eventgateway.Gateway interface. It defines a skeleton listener thread and initializes commonly used Gateway properties. The EmptyGateway source code includes comments that describe the additional information that you must provide to complete an event gateway class.

Building an event gateway

This section describes how to build an event gateway. To build a Gateway class, you can start with the EmptyGateway.java file as a template. (In the server configuration, this file is located in the *cf_root*/gateway/src/examples/ directory; in the J2EE configuration, the file is in the *cf_root*/WEB-INF/cfusion/gateway/src/examples/ directory.) This file defines a nonfunctional event gateway, but has the basic skeleton code for all Gateway class methods.

Wherever possible, the following sections use code based on the sample Socket event gateway to show how to implement event gateway features. (In the server configuration, this file is *cf_root*/gateway/src/examples/socket/SocketGateway.java; in the J2EE configuration, the file is *cf_root*/WEB-INF/cfusion/gateway/src/examples/socket/SocketGateway.java.)

Class constructor

An event gateway can implement any of the following constructors:

- `MyGateway(String gatewayID, String configurationFile)`
- `MyGateway(String gatewayID)`
- `MyGateway()`

When ColdFusion MX starts, it calls the constructor for each event gateway instance that you configure in ColdFusion MX. (ColdFusion also calls the gateway `Start` method after the event gateway is instantiated.). ColdFusion first attempts to use the two-parameter constructor.

Because each event gateway instance must have a unique ID, ColdFusion provides redundant support for providing the ID. If the event gateway implements only the default constructor, ColdFusion provides the ID by calling the event gateway's `setGatewayID` method.

If the event gateway does not implement the two-parameter constructor, it does not get configuration file information from ColdFusion.

The constructor normally calls the static `GatewayServices.getGatewayServices` method to access ColdFusion event gateway services. Although you need not do this, it is a good coding practice.

A minimal constructor that takes only a gateway ID might look like the following:

```
public MyGateway(String gatewayID) {
  this.gatewayID = gatewayID;
  this.gatewayService = GatewayServices.getGatewayServices();
}
```

The gateway constructor must throw a coldfusion.server.ServiceRuntimeException exception if there is an error that otherwise cannot be handled. For example, you should throw this exception if the event gateway requires a configuration file and cannot read the file contents.

If your gateway uses a configuration file, the constructor should load the file, even if the `Start` method also loads the file. You should do this because the constructor does not run in an independent thread, and ColdFusion can display an error in the ColdFusion MX Administrator of the file fails to load. If the `Start` method, which does run in a separate thread, fails to load the file, ColdFusion logs the error, but it cannot provide immediate feedback in the administrator.

The sample Socket event gateway has a single constructor that takes two parameters. It tries to load a configuration file. If you specify a configuration file in the ColdFusion MX Administrator, or the file path is invalid, it gets an IO exception. It then uses the default port and logs a message indicating what it did. The following example shows the Gateway constructor code and the `loadProperties` method it uses:

```
public SocketGateway(String id, String configpath)
{
  gatewayID = id;
  gatewayService = GatewayServices.getGatewayServices();
  // log things to socket-gateway.log in the CF log directory
  log = gatewayService.getLogger("socket-gateway");
  propsFilePath=configpath;
  try
  {
    FileInputStream propsFile = new FileInputStream(propsFilePath);
    properties.load(propsFile);
    propsFile.close();
    this.loadProperties();
   }
   catch (IOException e)
  {
    // Use default value for port and log the status.
    log.warn("SocketGateway(" + gatewayID + ") Unable to read configuration
      file '" + propsFilePath + "': " + e.toString() + ".  Using default port
      " + port + ".", e);
  }
}

private void loadProperties() {
  String tmp = properties.getProperty("port");
  port = Integer.parseInt(tmp);
}
```

Providing Gateway class service and information routines

Several gateway methods perform event gateway configuration services and provide event gateway information. The ColdFusion event gateway services call many of these methods to configure the event gateway by using information stored by the ColdFusion MX Administrator, and to get access to resources and information that are needed by event gateway services and applications. Some of these methods can also be useful in event gateway code. The following methods provide these services and information:

- `setCFCListeners`
- `setGatewayID`
- `getHelper`
- `getGatewayID`
- `getStatus`

ColdFusion MX calls the `setCFCListeners` method with the CFC or CFCs that are specified in the ColdFusion MX Administrator when it starts a gateway. ColdFusion MX also calls the method in a running event gateway when the configuration information changes, so the method must be written to handle such changes. The `setCFCListeners` method must save the listener information so that the gateway code that dispatches incoming messages to gateway services can use the listener CFCs in `setCFCPath` methods.

ColdFusion MX calls the `setGatewayID` method when it starts a gateway. The `getGatewayID` method must return the value set by this method.

ColdFusion MX calls the `getHelper` method when an application calls the CFML `GetGatewayHelper` function.

The following code shows how the SocketGateway class defines these methods. To create a new gateway, modify the `getHelper` definition to return the correct class, or to return null if there is no gateway helper class. Most gateways do not need to change the other method definitions.

```
public void setCFCListeners(String[] listeners) {
  this.listeners = listeners;
}
public GatewayHelper getHelper() {
  // SocketHelper class implements the GatewayHelper interface.
  return new SocketHelper();
}
public void setGatewayID(String id) {
  gatewayID = id;
}
public String getGatewayID() {
  return gatewayID;
}
public int getStatus() {
  return status;
}
```

Starting, stopping, and restarting the event gateway

Because an event gateway uses at least one listener thread, it must have `start`, `stop`, and `restart` methods to control the threads. These methods must also maintain the status variable that the Gateway class `getStatus` method checks, and change its value among `STARTING`, `RUNNING`, `STOPPING`, `STOPPED`, and `FAILED`, as appropriate.

The start method

The `start` method initializes the event gateway. It starts one or more listener threads that monitor the gateway's event source and respond to any messages it receives from the source.

The `start` method should return within a time-out period that you can configure for each event gateway type in the ColdFusion MX Administrator. If it does not, the ColdFusion MX Administrator has a Kill on Startup Timeout option for each gateway type. If you select the option, and a time-out occurs, the ColdFusion starter thread calls an interrupt on the gateway thread to try to kill it, and then exits.

Note: If the `start` method is the listener (for example, in a single-threaded gateway), the method does not return until the gateway stops. Do not set the Kill on Startup Timeout option in the ColdFusion MX Administrator for such gateways.

If the gateway uses a configuration file, the `start` method should load the configuration from the file. Doing so lets users change the configuration file and restart the gateway without restarting ColdFusion. Applications should also load the configuration file in the constructor; for more information, see "Class constructor" on page 1092.

In the SocketGateway class, the `start` method starts an initial thread. (In a single-threaded Gateway, this would be the only thread.) When the thread starts, it calls a `socketServer` method, which uses the Java ServerSocket class to implement a multithreaded socket listener and message dispatcher. For more information on the listener, see "Responding to incoming messages" on page 1096.

```
public void start()
{
  status = STARTING;
  listening=true;
  // Start up event generator thread
  Runnable r = new Runnable()
  {
    public void run()
    {
      socketServer();
    }
  };
  Thread t = new Thread(r);
  t.start();
  status = RUNNING;
}
```

The stop method

The `stop` method performs the event gateway shutdown tasks, including shutting down the listener thread or threads and releasing any resources. The following example shows the SocketGateway `stop` method:

```
public void stop()
{
  // Set the status variable to indicate that the server is stopping.
  status = STOPPING;
  // The listening variable is used as a switch to stop listener activity.
  listening=false;
  // Close the listener thread sockets.
  Enumeration e = socketRegistry.elements();
  while (e.hasMoreElements()) {
    try
    {
      ((SocketServerThread)e.nextElement()).socket.close();
    }
    catch (IOException e1)
    {
      // We don't care if a close failed.
```

```
        //log.error(e1);
      }
    }
    // Close and release the serverSocket instance that gets requests from the
    // network.
    if (serverSocket != null) {
      try
      {
        serverSocket.close();
      }
      catch (IOException e1)
      {
      }
      //Release the serverSocket.
      serverSocket = null;
    }
    // Shutdown succeeded; set the status variable.
    status = STOPPED;
}
```

The restart method

In most cases, you implement the `restart` method by calling the `stop` method and the `start` method consecutively, but you might be able to optimize this process for some services. The following code shows the SocketGateway class `restart` method:

```
public void restart() {
  stop();
  start();
}
```

Responding to incoming messages

One or more listener threads respond to incoming messages (events). The threads must include code to dispatch the messages to ColdFusion event gateway services, as follows:

1. Create a CFEvent instance.
2. Create a Map instance that contains the message and any other event gateway-specific information, and pass it to the CFEvent `setData` method.
3. Call the CFEvent `setOriginator` method to specify the source of the message. (This is required if the ColdFusion application will send a response.)
4. Call the CFEvent `setGateWayType` method to specify the event gateway type.
5. Set any other CFEvent fields where the default behavior is not appropriate; for example, call the `setCFCPath` method to replace the default listener CFC. (For information on default CFEvent fields, see "CFEvent class" on page 1088.)
6. Call the `gatewayService.addEvent` method to dispatch the CFEvent instance to ColdFusion.
7. Handle cases where the event is not added to the event gateway service queue (the `addEvent` method returns False).

If your application sends any messages to multiple listener CFCs, the gateway must create and configure a CFEvent instance and call the `gatewayService.addEvent` method to send the message to each separate listener CFC. The gateway's `setCFCListeners` method must make the CFC paths available to the gateway for configuring the CFEvent instances.

If your ColdFusion server carries a heavy event gateway message load, the ColdFusion event gateway services event queue might reach the maximum value set in the ColdFusion MX Administrator. When this happens, the `gatewayService.addEvent` method returns False and fails. Your code can do any of the following:

- Return a message to the sender to indicate that their message was not received.
- Wait until the queue is available by periodically comparing the values returned by the GatewayService `getQueueSize` and `getMaxQueueSize` methods, and retry the `addEvent` method when the queue size is less than the maximum.
- Log the occurrence using the logger returned by the GatewayService `getLogger` method. (For more information, see "Logging events and using log files" on page 1099.)

The SocketGateway class implements the listener using a java.net.ServerSocket class object and SocketServerThread listener threads. (See the SocketGateway source for the SocketServerThread code.) When the listener thread gets a message from the TCP/IP socket, it calls the following `processInput` method to dispatch the message to ColdFusion. This method explicitly sets all required and optional CFEvent fields and sends the event to ColdFusion. If the `addEvent` call fails, it logs the error.

Note: Much of the `processInput` method code supports multiple listener CFCs. A gateway that uses only a single listener cfc, would require only the code in the latter part of this method.

```
private void processInput(String theInput, String theKey)
{
  // Convert listeners list to a local array
  // Protect ourselves if the list changes while we are running
  String[] listeners;
  int size = cfcListeners.size();
  if (size > 0)
  {
    // Capture the listeners list
    synchronized (cfcListeners)
    {
      listeners = new String[size];
      cfcListeners.toArray(listeners);
    }
  }
  else
  {
    // Create a dummy list
    listeners = new String[1];
    listeners[0] = null;
  }
  // Broadcast to all the CFC listeners
  // Send one message at a time with different CFC address on them
  for (int i = 0; i < listeners.length; i++)
  {
```

```
      String path = listeners[i];
      CFEvent event = new CFEvent(gatewayID);
      Hashtable mydata = new Hashtable();
      mydata.put("MESSAGE", theInput);
      event.setData(mydata);
      event.setGatewayType("SocketGateway");
      event.setOriginatorID(theKey);
      event.setCfcMethod(cfcEntryPoint);
      event.setCfcTimeOut(10);
      if (path != null)
        event.setCfcPath(path);
        boolean sent = gatewayService.addEvent(event);
      if (!sent)
        log.error("SocketGateway(" + gatewayID + ") Unable to put message on
          vent queue. Message not sent from " + gatewayID + ", thread " + theKey
          + ".  Message was " + theInput);
    }
  }
```

Responding to a ColdFusion function or listener CFC

The ColdFusion event gateway services call the event gateway's `outgoingMessage` method to handle messages generated when an event gateway application listener CFC's listener method returns a message or any CFML code calls a `SendGatewayMessage` function. This method must send the message to the appropriate external resource.

The `outgoingMessage` method's parameter is a CFEvent instance, containing the information about the message to send out. The CFEvent `getData` method returns a Map object that contains event gateway-specific information about the message, including any message text. All CFEvent instances received by the `outgoingMessage` contain information in the Data and GatewayID fields.

CFEvent instances returned from listener CFC `onIncomingMessage` methods include the incoming message's originator ID and other information. However, a gateway that might handle messages from the ColdFusion `SendGatewayMessage` function cannot rely on this information being available, so it is good practice to require that all outgoing messages include the destination ID in the data Map.

The `outgoingMessage` method returns a String value. The CFML `sendGatewayMessage` function returns this value to the ColdFusion application. The returned string should indicate the status of the message. By convention, ColdFusion event gateway `outgoingMessage` methods return "OK" if they do not encounter errors and do not have additional information (such as a message ID) to return.

Because event messages are asynchronous, a positive return normally does not indicate that the message was successful delivered, only that the `outgoingMessage` method successfully handled the message. In some cases, however, it is possible to make the `outgoingMessage` method at least partially synchronous. The SMS gateway, for example, provides two `outgoingMessage` modes:

Asynchronous mode The `outgoingMessage` method returns when the message is queued internally for delivery to the messaging provider's short message service center (SMSC)

Synchronous mode The method does not return until the message is delivered to the SMSC, or an error occurs.

This way, an SMS application can get a message ID for later use, such as to compare with a message receipt.

Example outgoingMessage method

The following `outgoingMessage` method is similar to the version in the SocketGateway class. It does the following:

1. Gets the contents of a MESSAGE field of the Data Map returned by the CFEvent class `getData` method.
2. Gets the destination from an `outDestID` field in the data Map.
3. Uses the destination's socket server thread to write the message.

```
public String outgoingMessage(coldfusion.eventgateway.CFEvent cfmsg) {
  String retcode="ok";
  // Get the table of data returned from the event handler
  Map data = cfmsg.getData();
  String message = (String) data.get("MESSAGE");
  // Find the right socket to write to from the socketRegistry hashtable
  // and call the socket thread's writeoutput method.
  // (Get the destination ID from the data map.)
  if (data.get("outDestID") != null)
    ((SocketServerThread)socketRegistry.get(data.get("outDestID"))).
        writeOutput(message);
  else {
    System.out.println("cannot send outgoing message. OriginatorID is not
      available.");
    retcode="failed";
    }
  return retcode;
}
```

Logging events and using log files

The `GatewayServices.getLogger` method returns an instance of the coldfusion.eventgateway.Logger class that you can use to log messages to a file in the ColdFusion logs directory. (You set this directory on the ColdFusion MX Administrator Logging Settings page.) The method can take no parameter, or one parameter:

- The default `GatewayServices.getLogger` method uses the eventgateway.log file.
- Optionally, you can specify a log filename, without the .log extension or directory path.

The following example tells ColdFusion to log messages from the gateway to the mygateway.log file in the ColdFusion logs directory:

```
coldfusion.eventgateway.Logger log =
  getGatewayServices().getLogger(“mygateway“);
```

The Logger class has the following methods, all of which take a message string. The method you use determines severity level that is set in the log message.

- `info`
- `warn`
- `error`
- `fatal`

You can also pass these methods an exception instance as a second parameter. When you do this, CColdFusion MX puts the exception information in the exception.log file in the ColdFusion MX logs directory.

You can use these methods to log any messages that you find appropriate. If you use the default eventgateway.log file, however, remember that it is used by all ColdFusion MX standard gateways, and might be used by other gateways. As a result, you should limit the messages that you normally log to this file to infrequent normal occurrences (such as gateway startup and shutdown) or errors for which you want to retain data.

ColdFusion MX uses the following format for the message text, and your application should follow this pattern:

```
GatewayType (Gateway ID) Message
```

The SMS event gateway, for example, includes the following exception catching code, which logs a general exception messages and the exception name in the eventgateway.log file, and also (automatically) logs the exception in the exceptions.log file:

```
catch(Exception e)
{
  logger.error("SMSGateway (" + gatewayID + ") Exception while processing
  incoming event: " + e, e);
}
```

Note: When you are developing an event gateway application, you can use the ColdFusion Log viewer to inspect your log files and the standard ColdFusion MX log files, including the eventgateway.log file and exception.log file. You can use the viewer to filter the display, for example, by selecting different severity levels, or searching for keywords.

Deploying an event gateway

To deploy an event gateway, you deploy and event gateway type and configure one or more event gateway instances.

To deploy a event gateway type in ColdFusion:

1. Compile your Gateway class and put it in a JAR file along with any other required classes.

   Note: The ColdFusion MX class loader includes the gateway \lib directory on its classpath and includes any JAR files that are in that directory on the class path.

2. Put the JAR file in the *cf_root*\WEB-INF\cfusion\gateway\lib directory on J2EE configurations or the *cf_root*\gateway\lib directory on server configurations. This directory is on the ColdFusion classpath.

3. Ensure that ColdFusion event gateway services are enabled on the ColdFusion MX Administrator Data & Services > Event Gateway > Settings page.
4. On the ColdFusion MX Administrator Data & Services > Event Gateways page, click the Manage Gateway Types button to display the Gateway Types page.
5. On the Add/Edit ColdFusion Event Gateway Types form, enter a type name (for example, SMS for the SMS event gateway), a description, and the full Java class name (for example, coldfusion.eventgateway.sms.SMSGateway for the SMS event gateway). If appropriate, change the Startup Timeout settings from the default values.
6. Click the Add Type button to deploy the event gateway type in ColdFusion.

The following procedure describes how to configure an event gateway instance that uses the gateway type.

To configure an event gateway instance

1. If you have just finished deploying the event gateway type, click the Manage Gateway Instances button; otherwise, select Event Gateways > Gateway Instances in the ColdFusion MX Administrator.
2. On the Add/Edit ColdFusion Event Gateways Instances form do the following:
   - Enter the instance name in the Gateway ID field
   - Select the event gateway type that you added from the Gateway Type drop-down list
   - Specify the paths to the listener CFC or CFCs that will handle the messages.
   - If the event gateway requires a configuration file, enter the path to the file in Gateway Configuration File field.
   - If you do not want the gateway to start up automatically when ColdFusion MX starts, change the Startup Mode selection to Manual or Disabled
3. Click the Add Gateway Instance button.
4. In the list of configured instances, click the start button (green triangle) on the gateway instance's entry to start the instance.

INDEX

A

access
  client variables 349, 356
  generated content 252
access security, component 235
accordion, Flash form cfformgroup element 693
action pages 613
ActionScript, in Flash forms 704
Active Server Pages 875
addBuddy IM GatewayHelper method 1059
addDeny IM GatewayHelper method 1060
addEvent GatewayServices method
  responding to messages 1096
  signature and description 1088
adding
  data elements to structures 112
  elements to an array 102
addPermit IM GatewayHelper method 1060
AddSOAPRequestHeader CFML function 907
AddSOAPResponseHeader CFML function 907
administrator, event gateway pages 1026
alignment palette, Report Builder 776
ancestor tags
  data access 256
  defined 254
AND operator, SQL, defined 454
application events
  about 278
  example 292
  handler for 285
  onError example 291
  order of processing 286
application framework
  about 342
  approaches to 281
  custom error pages 317
  mapping 279
application pages
  errors 317
  variables 33
Application scope 34, 66, 277, 340
  for COM objects 958
  in event gateway listener CFCs 1032
  sharing with JSP pages, servlets 923
application security. *See* security, application; authentication
application servers, data exchange across 876
application variables
  configuring 357
  description 277, 340
  listing 359
  usage tips 358
  using 357
application-defined exception 311
application-level settings 278
Application.cfc file
  about 278
  defining an application with 275
  defining applications with 282
  generated by Dreamweaver Login Wizard 389
  handling errors 290
  how ColdFusion locates 280
  managing applications with 286
  migrating from Application.cfm 294
  request management 288
  session management 288
Application.cfm file
  application-level settings 295
  creating 295
  example 296
  how ColdFusion locates 280
  migrating to Application.cfc 294
  user-defined functions in 194

applications
authentication 380
caching 299
ColdFusion and J2EE 276
default variables 287
defaults 296
defined 275
defining utility functions 286
defining with Application.cfc 282
definition pages 280
directory structure 279, 281
elements of 276
ending 276
error handling 290, 296
framework 277
globalization 403
in ColdFusion 275
internationalization 404
JSP tags 922
localization 404
login 289, 296
managing with Application.cfc 286
migrating to application.cfc 294
naming 283, 295
OnRequestEnd.cfm 280
optimizing 298
optimizing database access 303
page settings 284, 296
persistent scope variables 340
reusable elements 277
security 279, 374
servlets in 922
shared variables 277
specifying client variable storage 348
stored procedures in 303
storing variables in 358
unnamed 924
user security 388, 394
variable options, setting 295
See also application events
events
See also application events
applicationToken 382
area chart, example 753
arguments
optional 174, 184
passing 179
user-defined functions 179
using function names 196
arguments. *See* parameters
Arguments scope
as array 182
as structure 183
in components 229
user-defined functions and 181
variables 34, 66
arithmetic operators 76
ArrayAppend CFML function 102
ArrayDeleteAt CFML function 103
ArrayInsertAt CFML function 102
ArrayNew CFML function 100
ArrayPrepend CFML function 102
arrays
2-dimensional 98
3-dimensional 99
adding data to 100
adding elements to 100, 102
as variables 52
copying 104
creating 100
description 97
elements 98
elements, adding 102
elements, deleting 103
functions 108
in dynamic expressions 86
index 98
multidimensional 100
passing to functions 179
populating 105
referencing elements in 100
resizing 103
user-defined functions and 179
validating 666
variables 52
ArraySet CFML function 105
ArraySort CFML function 114
ASCII 406
assignment, CFScript statements 128
associative array notation 109
asynchronous mode
defined 1070
sending SMS messages using 1079
attachments, e-mail 988
attributecollection reserved attribute 248
attributes
for custom tags 246
passed from forms to XML 729
passing values 245, 246
Attributes scope 33, 65

authenticating, users 289
authentication
  application-based example 394
  application-based scenario 387
  cookies and 380
  defined 376
  digest 379
  HTTP, basic 379
  LDAP example 399
  logout 385
  Microsoft NTLM 379
  persistence of 380
  persistence of information 386
  Session scope and 380
  storing login information 380
  types 379
  user 378
  using a database 390
  web server 379
  web server scenario 386
  web server-based example 391
  web servers and 378
authorization
  defined 376
  described 376
  web servers and 378
AVG SQL function 739

B

backreferences
  about 150
  case conversions with 152
  in regular expressions 674
  in replacement strings 151
  omitting from 152
banded reports
  defined 770
  sample 771
base tags 254
Base64 variables 51
basic authentication
  HTTP 379
  web services and 906
basic exception types 311
best practices, Flash forms 707
BETWEEN SQL operator 454
binary data
  validating 666
binary data type 45
binary files, saving 994
binary variables 51
binding
  data, in Flash forms 699
  SMS gateway 1069
BOM 411
Boolean
  operators 76
  values, validating 665
  variables 49
break, CFScript statement 135
browser, ColdFusion component 238
browsers
  cfform considerations 634
  displaying e-mail in 984
  requesting component pages with 237
  transferring data to a server 881
buddies
  adding instant messaging 1053
  IM GatewayHelper management methods 1059
  management example 1056
  methods for managing 1052
build
  components 205
  drop-down list boxes 642
  queries 467
  reports 778
  search interfaces 619
  slider bar controls 643
  tree controls 634
built-in variables
  client 350
  custom tags 250
  server 359
  session 355
Byte Order Mark (BOM) 411

C

C++ CFX tags
  implementing 269
  LD_LIBRARY_PATH 269
  registering 270
  SHLIB_PATH 269
C++ development environment 269
cacerts file 544
caching
  applications 299
  attributes 300
  Flash data 707
  flushing pages 300
  locations of 299

page areas 300
pages 299
possible issues 300
to variables 300
caching connections 1001
calculated field
calculation options 783
defining 782
Report Builder 782
Caller scope 33, 65
calling
CFX tags 262
COM objects 947
CORBA objects 961
Java objects 928
nested objects 930, 948
object methods 929, 947
user-defined functions 175
cascading style sheets. *See* CSS
case sensitivity
assigning XML document object data 854
referencing XML document object data 852
specifying XML document object 857
case sensitivity, of CFML 39
cellular phone, simulator for SMS 1081
cfabort tag
about 38
OnRequestEnd.cfm 280
cfapplet tag
description 632
using 655, 656
cfapplication tag 344
defining an application with 275
relation to application definition pages 280
cfargument tag
and validation 662
handling invalid data 670
validation considerations 664
cfassociate tag 255
cfbreak tag 37
cfcache tag
about 299
possible issues 300
cfcache tag, location of tag 300
cfcalendar tag, masking input of 671
cfcase tag 36
cfcatch tag 323
example 327
using 323
variables 325
cfchart tag
about 734
charting queries 736
introduced 733
cfchartdata tag 734, 736
cfchartseries tag 734
cfcollection tag 555
cfcompile utility 250
cfcomponent tag
attributes for document-literal web services 905
basic usage 205
web services, publishing 898
cfcontent tag
about 412
Excel spreadsheet 1017
using 999, 1014
CFCs
See also components
calling asynchronuously 1038
cfdefaultcase tag 36
cfdirectory tag
about 1012
for file operations 999
queries and 489
cfdocument tag
about 764
cfdocument scope 767
unsupported tags 764
with cfhttp 768
cfdocumentitem tag 766
cfdocumentsection tag 766
cfdump tag
for COM objects 949
for debugging 435
for multidimensional array 101
with query of queries 495
cfelse tag 36
cfelseif tag 36
cferror page 317
cferror tag 317
CFEvent class 1088
about 1088
constructor 1088
responding to incoming messages 1096
responding to outgoing messages 1098
CFEvent object
for SMS gateway messages 1073
CFEvent structure 1031

cfexit tag
about 38
behavior of 253
OnRequestEnd.cfm and 280
cffile tag 1003
cfflush tag
HTML headers and 623
SetLocale and 409
using 306, 623
cffont.properties file 804
cfform controls, described 632
cfform tag
and XML skinnable forms 710
usage notes 633
XML generated by, example 726
cfformgroup
accordion and tabnavigator attributes 696
in Flash forms 693
repeater attribute 695
cfformgroup tag
example 694
extending in XML forms 730
in XML skinnable forms 713
XML structure for 726
cfformitem tag
extending in XML forms 730
in Flash forms 690
in XML skinnable forms 713
XML structure for 726
cfftp tag
attributes 1002
connection actions 1002
using 999
cffunction tag
attributes 175
creating user-defined functions 174
security and 382
web services, publishing 899
cfgrid tag
controlling cell contents 648
editing data in 649
handling failed validation 682
returning user edits 648
using 644
validating with JavaScript 680
XML structure for 723
cfhttp tag
creating queries 994
Get method 836, 992
Post method 838, 996
queries and 489
use in Report Builder advanced query mode 797
using 992
with cfdocument tag 768
cfhttpparam tag 996
CFID
cookie 344
server-side variable 346
cfif tag 36
cfimport tag
about 243
calling custom tags 243
cfinclude tag
about 162
recommendations for 164
using 163
cfindex tag
queries and 489
cfinput tag
bind attribute 699
handling failed validation 682
validating with JavaScript 680
XML structure for 722
cfinsert tag
creating action pages 478
form considerations 477
inserting data 475
cfinvoke tag
example 892
invoking component methods 218
passing parameters with 225
web services, consuming 891, 892
within a component definition 220
cfinvokeargument tag
basic usage 226
omit attribute 893
web services invocation 892
cfldap tag
about 520
indexing queries 572, 573
output 539
queries and 489, 539
Verity and 539
cflocation tag 351
cflock tag
controlling time-outs 365
examples 368
for file access 371
name attribute 365
nesting 366

scope attribute 364
throwOnTimeout attribute 365
time-out attribute 365
using 360, 363
cflog tag 321
cflogin tag
structure 383
using 381, 382
cfloginuser tag 381
cflogout tag
about 382
and user sessions 386
cfloop tag
about 37
emulating in custom tags 253
nested 106
cfmail tag
attributes 975
sample uses 977
sending mail as HTML 976
cfmailparam tag 981
cfmailpart tag, multipart e-mail 977
CFML
case sensitivity 39
CFScript 40
Code Compatibility Analyzer 442
code validation 442
comments 28
components 32
constants 32
converting data to JavaScript 880
data types 35
debugging 434
description 22
development tools 23
elements 28
expressions 34, 76
extending 259
extensions 23
flow control 35
functions 23, 31
in Report Builder 797
reserved words 39
special characters 39
syntax errors 443
tags 22, 29
variables 33
CFML event gateway 1038
CFML functions
ArrayAppend 102
ArrayDeleteAt 103
ArrayInsertAt 102
ArrayNew 100
ArrayPrepend 102
ArraySet 105
ArraySort 114
CreateObject 891
CreateTimeSpan 306, 354, 492
DateFormat 679
DeleteClientVariablesList 351
DollarFormat 679
dynamic evaluation 87
evaluating 88
Expression Builder 799
for arrays 108
for globalization 413
for queries 489
for security 381
for structures 121
formatting data 617
GetAuthUser 382
GetClientVariablesList 350
GetLocale 409
GetLocaleDisplayName 409
HTMLEditFormat 883, 986
IIF 90
IsCustomFunction 198
IsDefined 70, 113, 616, 678
IsStruct 113
IsUserInRole 382
IsWDDX 851
IsXML 851
IsXmlAttribute 851
IsXmlDoc 851
IsXmlElem 851
IsXmlNode 851
IsXmlRoot 851
JavaCast 936
ListQualify 629, 630
ListSort 114
MonthAsString 105
Rand 625
RandRange 625
REFind 153
REFindNoCase
153
Report Builder 799
Report Builder report functions 798

SetEncoding 418
SetLocale 409
SetVariable 90
StructClear 116
StructCount 113
StructDelete 116
StructIsEmpty 113
StructKeyArray 114
StructKeyExists 113
StructKeyList 114
StructNew 112
syntax 80
ToString 851
URLEncodedFormat 444
XmlChildPos 851
XmlElemNew 850
XmlFormat 851
XmlGetNodeType 851
XmlNew 850, 856
XMLParse 850
XmlParse 857
XmlSearch 850
XmlTransform 850
XmlValidate 851
See also individual function names
CFML syntax, Code Compatibility Analyzer 442
CFML tags, for globalization 413
cfmodule tag, calling custom tags 243
cfNTauthenticate tag 382
cfobject tag
instantiating components 218
invoking component methods 221
cfoutput tag
data-type conversions 60
populating list boxes 625
using with component objects 928, 947
cfparam tag
about 71, 349
and validation 662
handling invalid data 670
testing and setting variables 71
using for validation 683
validation considerations 664
cfpop tag
deleting e-mail 989
queries and 489
query results 579
query variables 984
retrieving attachments 988
retrieving headers 985
retrieving messages 987
using 984
using cfindex with 572, 573
cfprocessingdirective tag 411
cfquery tag
cachedWithin attribute 306
creating action pages 479, 485
debugging with 434
populating list boxes 625
syntax 466
using 466
using cfindex with 572, 573
using cfquery attributes in the Query Builder 797
CFR file
displaying in a browser 791
displaying using cfreport 791
cfreport tag
displaying reports 775
saving to a file 791
cfreportparam tag
about 778
input parameters 779
using 792
cfrethrow tag
nesting 333
syntax 333
using 325, 333
cfsavecontent tag 300
CFScript
comments 126
conditional processing 129
creating user-defined functions 172
description 23
differences from JavaScript 127
example 124, 137
exception handling 136
expressions 125
function statement 173
introduction 40
language 125
looping 131
reserved words 127
return statement 174
statements 125
syntax, for user-defined functions 173
user-defined function syntax 173
using 123
var statement 174
variables 125
web services, consuming 893

cfsearch tag
about 548
properties 561
cfselect tag
handling failed validation 682
populating list boxes 642
XML structure for 722
cfset tag
component objects and 928, 947
description 23
tag syntax 29
cfsetting tag, debugging with 435
cfslider tag
description 633
handling failed validation 682
validating with JavaScript 680
cfstat utility
enabling 425
Windows NT and 425
cfstoredproc tag 303
cfswitch tag 36
cftextarea tag, bind attribute 699
cftextinput tag
handling failed validation 682
validating with JavaScript 680
cfthrow tag
nesting 333
using 332
cftimer tag 440
CFToken
cookie 344
server-side variable 346
cftrace tag
attributes 439
using 436
cftree tag
description 633
form variables 638
handling failed validation 682
image names 640
in cfform tag 633
URLs in 641
validating with JavaScript 680
XML structure for 725
cftry tag 323
example 327
nesting 333
cfupdate tag
creating action pages 483
using 483
cfwddx tag 876
CFX tags
calling 168, 262
cfx.jar 262
compiling 269
creating in Java 261
debugging in C++ 270
debugging in Java 266
description 259
developing in C++ 269
Java 260
LD_LIBRARY_PATH 269
locking access to 360, 365, 371
recommendations for 168
registering 270
sample C++ 269
sample Java 260
scopes and 69
SHLIB_PATH 269
testing Java 262
using 167
cfx.jar 262
cfxml tag 850
CGI
cfhttp Post method and 836, 992
returning results to 998
scope 33, 65
character classes 149
character encodings
about 406
conversion issues 408
defined 405
determining page 411
files 419
forms and 417, 418
processing data from various sources 419, 420
searching and indexing 421
Unicode 407
character sets
defined 405
See also character encodings
Chart Wizard, Report Builder 800
charting
about 736
individual data points 736
Report Builder 800
charts
3D 746
administering 756
area 753

background color 744
border 745
caching 756
curve chart considerations 755
data markers 748
dimensions 744
drill-down 758
embedding URLs 758
example 751
file type 744
foreground 744
labels 745, 746
linking from 758
markers 746
multiple series 747
paint 747
referencing JavaScript 758
threads 756
tips 747
check boxes
errors 615
lists of values 627
multiple 627
child tags 254
class loading, mechanism 919
classes, debugging 267
classpath
configuring 261
Java objects and 919
client cookies 344
Client scope 277
about 34, 66, 340
in event gateway listener CFCs 1033
client state management
clustering 346
described 342
client variable storage, specifying 348
client variables
built-in 350
caching 351
cflocation tag and 351
characteristics of 277, 340
configuring 346
creating 349
deleting 351
description 343
exporting from Registry 351
listing 350
periods, using 58
setting options for 346
storage method 347
using 349, 356
clustering
and persistent variables 342
client state management 346
code
locking 360
reusing 161
structuring components 230
Code Compatibility Analyzer, using 442
ColdFusion
applications 275
CFScript 123
component browser 238
development tools 23
EJBs and 938
J2EE and 24
JSP and 918
logout 385
searching 547
servlets and 918
support for globalization 405
using for instant messages 1047
using for SMS 1065
variables 44
XML and 843
See also ColdFusion MX
ColdFusion components. *See* components
ColdFusion MX
about 22
action pages, extension for 613
CFML 22
CORBA type support 964
dynamic evaluation 87
error handling 312
error types 309
functions 23
integrating e-mail with 973
Java objects and 919
scripting environment 22
security 374
standard event gateways 1024
support for LDAP 519
tags 22
Verity Search Server 24
See also ColdFusion
ColdFusion MX Administrator
creating collections 553
debugging settings 424
event gateway pages 1026

options 24
web services, consuming 896
collections
creating 553
creating with cfcollection tag 555
defined 547
indexing 554, 559, 560
populating 554
searching 548
color format, Flash styles 702
column aliases, SQL 457
columns 447
COM
arguments 954
calling objects 947
character encodings 420
component ProgID and methods 949
connecting to objects 952
creating objects 951
description 946
displaying object with cfdump 949
error messages 955
getting started 948
requirements 949
setting properties 952
threading 954
using properties and methods 952
viewing objects 950
WDDX and 875
COM objects
Application scope, using 958
calling 947
connecting to 952
creating 951
displaying with cfdump 949
improving performance of 955, 958
Java proxies for 955
releasing 953
viewing 950
commas, in search expressions 591
comments
CFScript 126
in CFML 28
commits 451
Common Object Request Broker Architecture. *See* CORBA
compiler exceptions
about 310
errors 313
compiling, C++ CFX tags 269
complex data types
about 45
returning 911
web services and 909
complex variables 52
Component Object Model. *See* COM
component objects
about 945
cfset tag and 947
invoking 947
components
accessing remotely 223
building 205
building secure 235
defining methods 205
displaying output 213
documenting 214
elements of 204
finding ProgID and methods 949
for application utility functions 286
for web services 898, 902
function local variables 229
getting information about 236
in persistent scopes 234
inheritance 231
initializing instance data 207
instantiating 218
introductions 32
introspecting 236
invocation techniques 217
invoking directly 220
invoking from Flash 223
invoking methods dynamically 220
invoking methods transiently 219
invoking with forms 222
invoking with URLs 222
metadata 237
method parameters 209
naming 216
packages 233
programmatic security 235
recommendations for 165
requesting from the browser 237
requirements for web services 899
returning method results 213
reusing code 230
saving 215
specifying location of 224
Super keyword 230
tags and functions for 204

using multiple files for 207
variables 227
web services and 898
when to use 202
concatenation operators
& 34, 78
string (QofQ) 509
configurations 24
configuring
event gateway instances 1101
event gateways 1090
IM gateways 1050
SMS gateways 1071
connections, caching FTP 1001
constants 32
constants, for applications 287
constructors
CFC 208
using alternate 933
continue, CFScript statement 135
Cookie scope 33
about 66
catching errors 623
variables 58
cookies
authentication and 380
client 344
client state management 342
for storing client variables 347
security without using 381
sending with cfhttp 838, 996
copying, server files 1009
CORBA
calling objects 964
case considerations 963
character encodings 420
description 946
double-byte characters 968
example 969
exception handling 968
getting started 961
interface 946
interface methods 963
naming services 962
parameter passing 963
CreateObject CFML function
about 893
example 894
web services, consuming 891, 894
CreateTimeSpan CFML function 306, 354, 492
creating
action pages 614
action pages to insert data 477
action pages to update data 483
Application.cfm 295
arrays 100
basic charts 734
client variables 349
collections 553, 555
data grids 644
dynamic form elements 627
error application pages 318
forms with cfform 631
HTML insert forms 476
insert action pages 478, 479
Java CFX tags 261
multidimensional arrays 100
queries from text files 994
queries of queries 489
structures 111
update action pages 483, 485
update forms 481
updateable grids 647
credit card numbers, validating 665
criteria, multiple search 619
cross-site scripting, protecting from 664
Crystal Reports 805
CSS, styling XML forms using
currency, globalization functions 414
currentpagenumber, cfdocument scope 767
curve charts 755
custom exception types 312
custom functions. *See* user-defined functions
custom tags
ancestor 254
attributes 246
base 254
built-in variables 250
calling 166, 242, 250
calling with cfimport 243
calling with cfmodule 243
CFX 259
children 254
compiling 250
data access example 257
data accessibility 255
data exchange 255
descendants 254
downloading 244
encoding 250

example 247
execution modes 251
filename conflicts 244, 249
instance data 250
location of 242
managing 249
naming 242
nesting 253
parent 254
passing attributes 245, 246
passing data 254
path settings 242
recommendations for 167
restricting access to 244, 249
terminating execution 253
types 30
using 166, 244

D

data
accessibility with custom tags 255
binding in Flash forms 699
caching in Flash forms 707
component instance 207
converting to JavaScript object 880
exchanging across application servers 876
exchanging with WDDX 876
graphing 735
passing between nested tags 254
transferring from browser to server 881
data binding error, Report Builder 797
data command, SMS 1078
data model, XML skinnable forms 716
data sharing, JSP pages 923
data sources
configuration problems 444
storing client variables in 347
troubleshooting 444
types of 465, 830
data types
about 33
binary 35, 45
complex 35, 45
considerations 46
conversions 59
default conversion 934
in CFML 35
object 35, 45
resolving ambiguous 935
simple 35, 45
validating 73, 666
variables 45
See also complex data types
data validation
about 659
considerations for forms 667
handling invalid data 669
security considerations 664
selecting techniques 662
techniques 660
with cfparam tag 683
with IsValid function 683
data, charting data from query 735
data-type conversions
ambiguous types 62
case sensitivity 60
cfoutput tag and 60
considerations 60
date-time values 63
date-time variables 61
default Java 934
example 64
issues in 61
Java and 63
JavaCast and 63
numeric values 61
process 59
Query of Queries CAST function 504
quotation marks 64
types 59
web services and 896
database
exceptions 326
failures 311
Database Management System. See DBMS
databases
authenticating users with 390
building queries 467
case sensitivity 454
character encodings 419
columns 447
commits 451
controlling access to 360
debug output 430
deleting data 487
deleting records 487, 488
deleting rows 459
elements of 448
fields 447
forms for updating 476

insert form 478
inserting data 458, 477
inserting records 475
introduction 447
locking 360
modifying 458
multiple tables 449
optimizing access 303
permissions 451
reading 455
record delete 487
record sets 455
records 447
retrieving data from 466
rollbacks 451
rows 447
SQL 452
stored procedures 303
stored procedures, debugging 431
tables 447
transactions 451
update form 481
updating 459, 475, 480
date fields, masking input in Flash 671
date-time format 50
date-time values, conversions 63
date-time variables
about 49
conversions 61
format 50
locale specific 50
representation of 51
DateFormat CFML function 679
dates
globalization functions 414
masking input 671
Query of Queries 511
validating 665
DBCS 406
DBMS 452
DCOM
description 946
getting started 948
See also COM
deadlocks 366
debug information
for a query 434
outputting 266
debug pane 434
debugging
browser output 426
C++ CFX tags 270
cftimer tag 440
ColdFusion MX Administrator and 424
configuring 423
custom pages and tags 317
Dreamweaver MX 423
enabling 424
event gateway listener CFCs 1033
Java CFX tags 266
Java classes for 267, 268
output 425
output format 424
programmatic control of 434
SQL queries 430
stored procedures 431
debugging output
cfquery tag 434
cfsetting tag 435
cftimer 440
classic 424
database activity 430
dockable 424, 433
exceptions 432
execution time 428
format 424
general 427
in browsers 426
IP address for 426
IsDebugMode function 435
programmatic control 434
queries 430
sample 425
scopes 433
SQL queries 430
trace 432, 436
debugging output, dockable
about 424
application page 434
debug pane 434
format 433
decision, or comparison, operators 77
declaring
arrays 100
structures and sequences 963
default values
of application variables 287
of variables 72
Define Query Fields panel 794

Define Report Grouping panel 794
DELETE SQL statement 453, 459, 487
DeleteClientVariablesList CFML function 351
deleting
client variables 351
data 487
database records 487, 488
e-mail 989
server files 1009
structures 116
delimiters
search expression 592
text file 994
deploying
event gateway applications 1037
event gateways 1100
descendant tags 254
development environment
C++ 269
Java 260
digest authentication 379
directories
indexing 548
information about 1013
securing access to 376
watching changes to 1041
directory operations 1012, 1014
directory structure, application 279, 281
DirectoryWatcher example gateway 1041
displaying
COM objects with cfdump 949
query results 469
query results, in tables 617
displaying, component output 213
distinguished name 518
Distributed Component Object Model. *See* DCOM
distributing CFX tags 271
do while loop, CFScript 133
document type definitions, validating XML with 857
document-literal web services
consuming 894
publishing 904
documenting, components 240
DollarFormat function 679
DOM node structure
XmlName 849
XmlType 849
XmlValue 849
DOM node view
node types 846
XML 845
dot notation
accessing structures in web services 913
calling helper class methods 1089
case-sensitive XML restriction 852
CFCs 208
evaluating file upload status 1008
Query of Queries 497
restriction with structures 109
structures 109
using CFScript to invoke web services 893
double-byte character set 406
Dreamweaver Login Wizard
about 388
generated code 389
modifying 390
Dreamweaver MX
BOM 411
debugging and 423
getting component information 238
SQL editor 460
web services and 888
WSDL files and 888
drop-down list boxes. *See* list boxes
DTD. *See* document type definitions
duration, Flash time style format 701
dynamic evaluation
about 84
example 92
functions 87
steps to 84
dynamic expressions
about 84
string expressions 84
dynamic variable names
about 84
arrays and 86
example 92
limitations 86
number signs in 85
selecting 85
structures and 86
using 87

E

e-mail
  adding custom header 982
  attachments 981
  character encodings 420
  ColdFusion and 973
  customizing 980
  deleting 989
  error logging 974
  form-based 978
  including images in 982
  indexing 548, 579
  multiple recipients 979
  query-based 978
  receiving 983
  retrieving attachments 988
  retrieving headers 985
  searching 579
  sending 974
  sending as HTML 976
  sending multipart 977
  undelivered 974
  using POP 985
e-mail addresses, validating 665
e-mail messages, retrieving 987
editing, data in cfgrid 649
EJB
  calling 938
  requirements for 938
  using 938
elements
  of CFML 28
  of components 204
embedding
  fonts 804
  Java applets 655, 656
  URLs in a cftree tag 641
Empty example gateway 1040
enabling, session variables 354
encoding
  custom tags 250
  See also character encodings
encodingStyle, consuming web services 894
encryption, PDF 768
Enterprise Java Beans. *See* EJB
error handling
  about ColdFusion 307
  custom 279
  in user-defined functions 186
  strategies 314
error messages
  Administrator settings 314
  COM 955
  for validation 669
  generating with cferror 317
error pages
  custom 317
  example 320
  rules for 318
  specifying 317
  variables 319
errors
  application events 278, 285
  categories 309
  causes 308
  ColdFusion types 309
  creating application pages 318
  custom pages 317
  form field validation 309
  handling in Application.cfc 290
  input validation 320
  logging 321
  logging event gateway 1037
  missing template 309
  recovery 308
  See also exception
  sending email 974
  web services and 895
EUC-KR 406
euro, supporting 417
Evaluate CFML function 88
evaluating
  CFML functions 88
  file upload results 1008
  strings in functions 198
event gateway
  Gateway interface 1087
event gateway application, defined 1022
event gateway applications
  configuring in administrator 1028
  debugging 1033, 1037
  deploying 1037
  developing 1023
  example 1034
  listener CFCs 1027
  Menu example 1044
  models for 1029
  sending information 1030

structure of 1027
using example 1040
See also listener CFCs
event gateway instances
configuring 1028
defined 1022
event gateway listener, defined 1022
event gateway type, defined 1021
event gateways
about 1020, 1088
architecture of 1085
building 1092
CFEvent structure 1031
CFML 1038
configuration file 1090
configuring 1026
configuring for IM 1050
configuring for SMS 1071
defined 1021
deploying 1100
development classes 1091
development tools 1025
DirectoryWatcher example 1041
Empty example 1040
error log file 1037
GatewayHelper class 1089
GatewayServices class 1088
JMS example 1043
log file 1026
sample gateways and applications 1025
SocketGateway example 1040
standard 1024
structure of 1023
synchronizing messages 1098
use examples 1020
using example 1040
See also Gateway classes
event, defined 1021
EventGateway, event gateway development class 1091
eventgateway.log file 1026
events, application 278
examples
ancestor data access 257
application-based security 394
Application.cfc 292
Application.cfm 296
Application.cfm file 296
caching a connection 1002
CFML Java exception handling 937
CFScript 137
cftry/cfcatch 327
declaring CORBA structures 969
exception-throwing class 936
Java objects 931
JSP pages 924
JSP tags 921
LDAP security 399
locking CFX tags 371
onError method 291
regular expressions 674
request error page 320
setting default values 72
synchronizing file system access 371
testing for variables 72
user-defined functions 192
using cftry, cfthrow, and cfrethrow 333
using Java objects 931, 932
using StructInsert 117
using structures 120
validating an e-mail address 680
validation error page 320
variable locking 368
web server-based authentication 391
web services, consuming 892
web services, publishing 902
Excel spreadsheet, from cfcontent tag 1017
exception handling
cfcatch tag 323
cftry tag 323
CORBA objects 968
error handler page 317
example 327, 333
in CFScript 136
in ColdFusion MX 322
Java and 936
nesting cftry tags 333
rules 324
tags 322
exception types 310
advanced 312
basic 311
custom 312
Java 312
missing include file 311
exceptions
about 310
application events 278, 285
ColdFusion error type 309
compiler 310
database 326

debugging output 432
expressions 326
handling 314
handling in Application.cfc 290
in user-defined functions 190
information returned 325
Java 936
locking 327
missing files 327
naming custom 332
runtime 310
types 310
exclusive locks
about 363
avoiding deadlocks 366
execution time 428
format 428
of ColdFusion pages 428
tree format 429
using 429
explicit queries 583, 584, 585, 586
exporting client variable database 351
Expression Builder, in Report Builder 799
expression exceptions 311, 326
expressions 34
CFScript 125
dynamic 84
number signs in 83
operands 34
operator types 76
operators 34, 76
extending CFML 259
Extensible Messaging and Presence Protocol. *See* XMPP

F

fields
database 448
dynamic 785
searches 603
Fields and parameters panel, Report Builder 777
file operations
cfftp actions 1002
using cffile 1003
using cfftp 999
file scope 250
file types, supported for searching 549
files 1041
appending 1012
character encodings 419
controlling type uploaded 1006
copying 1009
deleting 1009
downloading 1014
locking access to 365, 371
moving 1009
name conflicts 1006
on server 1003
reading 1010
renaming 1009
securing access to 376
updating 360
uploading 1003, 1004
writing 1011, 1012
Find CFML function 141
finding
a structure key 113
component ProgID and methods 949
with regular expressions 141
Finish Subreport Setup panel 801, 803
Flash forms
about 687
accordions and tabbed navigators in 696
ActionScript 704
best practices for 707
controlling appearance 700
data binding 699
example 703
grouping elements of 693
img tag 691
setting field values 699
sizing 695
style syntax 701
using HTML in text 690
using query data in 695
See also skins 700
See also styles 700
Flash Remoting
ColdFusion Java objects 819
invoking components 223
web services and 895
Flash Remoting service
arrays and structures 814
components 817
data types 812
Flash variable scope 812
handling errors 820
returning records in increments 816
separating display code from business logic 808
using with ColdFusion 807
Flash Remoting, logging users in with 385

Flash scope 34, 66
FlashPaper
  cfdocument tag 764
  displaying external web pages 768
flow control, tags 35
font management 804
fonts, embedding 804
fontSize style 701
for loop, CFScript 131
for-in loop, CFScript 134
form controls, cfform 632
form fields
  required 677
  validation errors 309, 312
Form scope
  about 33, 65
  not shared using getPageContext method 923
form variables
  considerations 617
  in queries 614
  naming 613
  processing 613
  referring to 613
  scope of 617
formatting
  data items 619
  query results 618
forms
  accordions and tabbed navigators in 696
  action pages 613
  caching Flash data 707
  character encodings 418
  check boxes 627
  considerations for 613
  creating with cfform 631
  creating XSLT skins 728
  data encoding 417
  deleting data 487
  designing 609
  drop-down list boxes 642
  dynamically populating 625
  hidden field validation 675
  inserting data 476
  invoking components with 222
  Java applets in 655
  limiting data length 666
  login 383
  mapping CFML tags to XML 720
  preserving data 633
  preventing blank input 666
  preventing multiple submissions 666
  requiring entries 616
  slider bars 643
  tree controls 634
  updating data 481
  using ActionScript 704
  validating field contents 667
  XML skinnable, 709
  See also XML skinnable forms
  See also Flash forms
FROM SQL clause, description 453
FTP 991
  actions and attributes 1002
  caching connections 1001
  using cfftp 999
function local scope 34
function local variables, in components 229
function variable, defined 174
function, CFScript statement 173
function-only variables 185
functions
  ActionScript 705
  application utility 286
  built in 31
  calling 175
  example custom 192
  for arrays 108
  for components 204
  for XML 850
  GetMetaData 238
  introduction 31
  IsValid 661, 669
  isvalid 663
  JavaScript, for validation 680
  securing access to 375
  SendGatewayMessage 1035
  structures 121
  syntax 80
  user-defined 32
  See also ColdFusion functions, user-defined functions

G

gateway applications
  DirectoryWatcher example 1043
  sending messages 1035
Gateway classes
  constructor 1092
  logging events 1099
  responding to incoming messages 1096

responding to messages from ColdFusion 1098
service and information routines 1093
start, stop, and restart methods 1094
gateway directory 1025
Gateway interface 1087
gateway services, defined 1022
GatewayHelper
example using IM 1060
methods. *See* individual method names
using IM 1058
GatewayHelper class 1089
GatewayHelper objects
role of 1028
sending information 1030
using 1036
GatewayHelpers
SocketHelper example 1041
gateways. *See* event gateways
GatewayServices class 1088
generated content 252
GenericGateway, event gateway development class 1091
Get method, cfhttp 836, 992
GetAuthUser CFML function 382
getBuddyInfo IM GatewayHelper method 1059
getBuddyList IM GatewayHelper method 1059
getCFCMethod, CFEvent class method 1089
getCFCPath, CFEvent class method 1089
getCFCTimeout. CFEvent class method 1089
GetClientVariablesList CFML function 350
getCustomAwayMessage IM GatewayHelper method 1059
getData, CFEvent class method 1089
getDenyList IM GatewayHelper method 1060
getGatewayID CFEvent class method 1089
getGatewayID Gateway interface method
implementing 1093
signature and description 1087
getGatewayServices, GatewayServices class method 1088
getHelper Gateway interface method
implementing 1093
signature and description 1087
GetLocale CFML function 409
GetLocaleDisplayName CFML function 409
getLogger GatewayServices method
signature and description 1088
using 1099
getMaxQueueSize, GatewayServices class method 1088
GetMetaData function 238
getName IM GatewayHelper method 1059
getNickName IM GatewayHelper method 1059
getOriginatorID, CFEvent class method 1089
GetPageContext 920
getPermitList IM GatewayHelper method 1060
getPermitMode IM GatewayHelper method 1060
getProtocolName IM GatewayHelper method 1059
getQueueSize, GatewayServices class method 1088
getSOAPRequest CFML function 914
GetSOAPRequestHeader CFML function 907
getSOAPResponse CFML function 914
GetSOAPResponseHeader CFML function 907
getStatus Gateway interface method
implementing in a Gateway class 1093
signature and description 1087
getStatusAsString IM GatewayHelper method 1059
getStatusTimeStamp IM GatewayHelper method 1059
globalization 403
applications 403
character encodings 406, 408
character sets 405
euro currency 417
input data 417
Java and 405
locales 405
multi-locale content 416
request processing 410
tags and functions 413
See also character encodings
See also locales
graphing
queries 736
See also charts
grids
navigating 645
See also cfgrid tag
GROUP BY, SQL clause 453
grouping, Report Builder 780
GSM, and SMS 1067
GUIDs, validating 665

H

handling
applet form variables 657
exceptions 322
failed validation 682
hbox, Flash form cfformgroup element 693
hdividedbox, Flash form cfformgroup element 693
headers, customizing e-mail 982

headers, retrieving e-mail 985
hidden field validation
  described 661
hidden fields, for validation 675
hidden form fields
  specifying error messages 669
  validation, considerations 663
horizontal, Flash form cfformgroup element 693
HTML
  using in Flash forms 690
  using tables 617
HTMLEditFormat CFML function 883, 986
HTTP
  about 991
  authentication 379
  character encodings 420
  headers, viewing 915
  requests, viewing headers 915
  responses, viewing 915
HTTP/URL problems 444
HttpServletResponse, viewing headers 915
hyperlinks, Report Builder 789

I

IBM Lotus Instant Messaging. *See* Sametime
if-else, CFScript statements 129
IIF CFML function 90
IM. *See* instant messages
images
  adding to Flash forms 690
  including in e-mail 982
  Report Builder 784
  using in Flash form text 691
img tag, in Flash forms 691
implementing
  C++ CFX tags 269
  Java CFX tags 263
IN SQL operator 454
including ColdFusion pages 162
index.cfm or mm_wizard_index.cfm file 390
indexing
  cfldap query results 578
  database query results 573
  directories 548
  e-mail 548, 579
  LDAP query results 578
  query results 548
  updating 555
  websites 548
indexing collections
  about 554
  defined 547
  with Administrator 560
  with cfindex tag 559
infix notation, search string 591
inheritance
  of components 231
  Super keyword and 230
initiator applications, event gateway 1029
inout parameters 895
input parameters
  defining 783
  passing at runtime 792
  Report Builder 782
input validation
  cftree tag 638
  with JavaScript 680
INSERT SQL statement 453, 458
inserting data
  description 475
  with cfinsert 477
  with cfquery 479
instance data, custom tag 250
instance data, of components 207
instance, invoking methods of a component 218
instant messages
  buddy and permission management methods 1059
  configuration and status helper methods 1059
  configuring the event gateway 1050
  development and deployment process 1049
  example application 1054
  example using GatewayHelper 1060
  GatewayHelper object 1058
  handling incoming 1049, 1052
  handling status and request messages 1056
  sending 1050, 1053
  using ColdFusion for 1047
instantiating, components 218
integer variables 47
Intermediate Deliver Notification, described 1074
international languages, search support 551
internationalization
  about 404
  See also globalization
Internet
  applications 19
  ColdFusion and 20
  domain and security 383
  dynamic applications 19

HTML and 19
introspection, of components 236
invalid data, handling 669
invoking
COM methods 952
component methods 218
component objects 947
components directly 220
components from Flash 223
methods in cfobject 952
methods using dynamic names 220
objects 928
web services 892
IP address, debugging and 426
IsCustomFunction CFML function 198
IsDebugMode CFML function, debugging with 435
IsDefined CFML function 70, 113, 616, 678
isOnline IM GatewayHelper method 1059
IsSOAPRequest CFML function 907
IsStruct CFML function 113
IsUserInRole CFML function 382
IsValid function
described 661
handling invalid data 669
using 683
validation considerations 663
IsWDDX CFML function 851
IsXML CFML function 851
IsXmlAttribute CFML function 851
IsXmlDoc CFML function 851
IsXmlElem CFML function 851
IsXmlNode CFML function 851
IsXmlRoot CFML function 851

J

J2EE application server
benefits 24
ColdFusion MX and 24
GetPageContext 920
infrastructure 24
introduction 24
PageContext 920
Session management, Sessions
J2EE 352
J2EE configuration 24
J2EE session management
about 352
J2EE, applications and ColdFusion 276
Jabber. *See* XMPP
Java
alternate constructor 933
class loading mechanism 919
ColdFusion data and 933
considerations 933
custom class 940
customizing and configuring 261
data-type conversions with 63
development environment 260
EJB 938
exception classes 312
exceptions 936
getting started 930
globalization and 405
JavaCast function 935
objects 919
proxies for COM objects 955
user-defined functions 937
variables and CFML 920
WDDX and 875
Java applets
embedding 655, 656
form variables 657
overriding default values 656
registering 656
Java CFX tags
cfx.jar 262
debugging 266, 267
example 264
life cycle of 264
registering 262
writing 261
Java classes
custom 940
loading 919
Java exceptions 312
handling 936
tags for 936
Java logical fonts, in printable output 804
Java Messaging Service, event gateway for 1043
Java objects 919
calling 926
considerations 933
example 931
exception handling 936
invoking 928
methods, calling 929
properties 929
using 928
JavaCast CFML function 63, 935

JavaScript
considerations for validation 663
differences from CFScript 127
for validation 661
in charts 761
validating with 680
JMS example gateway 1043
joins, queries of queries 498
JSP pages
accessing 922
calling from ColdFusion 926
example 924
sharing data with 923
sharing scopes with 923
JSP tags
ColdFusion and 918
example 921, 922
in ColdFusion applications 922
standard 921
tag libraries 921
using 921
JSP, variables and CFML 920
JVM locale 409

K

keys, listing structure 114
keystore 544
keytool utility 544
keywords
ActionScript 705
Super, components and 230
Var, in functions 229

L

language, and locales 408
Latin-1 406
layered controls, Report Builder 789
LD_LIBRARY_PATH
about 269
C++ CFX tags 269
LDAP
adding attributes 537
asymmetric directory structure 517
attribute values 538
attributes 517, 518, 538
character encodings 420
deleting attributes 537
deleting entries 533
description of 516
directory attributes 537
directory DN 538
distinguished name 518
DN 538
entry 518
for authentication 399
object classes 519
querying directories 521
referrals 543
schema 519
schema attribute type 520
scope 521
search filters 521
symmetrical directory structure 516
updating directories 527, 535
LDAP query results
indexing 578
searching 578
length format, Flash styles 701
LIKE SQL operator 454
linking from charts 758
links, Report Builder 789
list boxes
populating 642
populating dynamically 625
list variables 48
listener CFCs
debugging 1033
defined 1022
example 1034
listener methods 1030
role of 1027
using persistent scopes in 1032
listing
Application variables 359
client variables 350
ListQualify CFML function 629, 630
ListSort CFML function 114
locales
about 408
and JVM 409
defined 405
functions for 414
generating multi-locale content 416
SetLocale function 409
specifying names 408
supported 408

localization
applications 404
dates 50
See also globalization
lock management 366
locking
about 341
Application scope 287
avoiding deadlocks 366
CFX tags 371
exceptions 327
file access 371
granularity 366
scopes 364
with cflock 360
write-once variables 363
locking exceptions 311
locks
controlling time-outs 365
exclusive 363
naming 365
read-only 363
scopes and names 364
types 363
log files
example 321
using 321
Logger class, using 1100
logging
e-mail errors 974
errors 321
gateway events 1026
in event gateway applications 1037
using the CFML gateway 1039
logical fonts, mapping to physical 804
login
applicationToken 382
browser support for 384
getting User ID and password 383
Internet domains 383
logging out users 385
persistence of information 386
scope of 382
using Flash remoting 385
using forms for 383
logout, performing 385
looping through structures 117
Lotus Instant Messaging. *See* Sametime

M

Macromedia Dreamweaver MX. *See* Dreamweaver MX
Macromedia Flash Remoting. *See* Flash Remoting
Macromedia HomeSite+, SQL editor 462
mail servers, and ColdFusion MX 973
managing
custom tags 249
mapping, application framework 279
mask validation
considerations 663
described 660
masking, text input 670
matched subexpressions
len array 154
minimal matching 155
pos array 154
result arrays 154
matches, pattern 674
MBCS 406
Menu example gateway application 1044
message disposition, SMS 1074
message, defined 1021
messages
handling in Gateway classes 1096, 1098
logging 1039
retrieving e-mail 987
See also SMS
messges
See also instant messages
metadata, component 237
metadata, Query of Queries 504
method attribute, cfhttp tag 838, 992, 996
methods
defining in components 205
invoking using dynamic names 220
parameters of in components 209
returning, component results 213
using multiple files for 207
migrating
to Application.cfc 294
migration, Code Compatibility Analyzer 442
MIME type 1014
minoccurs, web services 893
missing files, exceptions 327
missing template errors 309, 312
mm_wizard_application_include.cfm file 389
mm_wizard_authenticate.cfc file 389
mm_wizard_login.cfm file 390
mobile phone, simulator for SMS 1081

modifiers
explicit queries 584
searching 601
MonthAsString CFML function 105
moving, data across the web 875
multicharacter regular expressions
for searching 144
for validation 673
multipart e-mail 977
multiple selection lists 629
multiple-byte character set 406
multiserver configuration 24

N

naming
applications 283, 295
components 216
conventions, for custom exceptions 332
methods dynamically 220
variables 246
navigating grids 645
nested number signs in expressions 83
nested objects, calling 948
nesting
cflock tags 366
cfloops for arrays 106
custom tags 253
object calls 930
tags, using Request scope 255
nillable argument, web services 893
NOT SQL operator 454
notification, of SMS message disposition 1080
NT authentication 382
NTLM authentication 379
number signs
in cfoutput tags 82
in general expressions 83
inside strings 82
inside tag attributes 81
nested 83
using 81
numberOf MessagesReceived IM GatewayHelper method 1059
numberOfMessagesSent IM GatewayHelper method 1059
numbers
globalization functions 414
validating 665
numeric variables
about 46
converting 61

O

object data type 45
object exceptions 311
object-oriented programming, and components 202
objects
calling methods 929, 947
calling nested 930, 948
COM 946
CORBA 946
DCOM 946
invoking 928
Java 919, 928
Java case considerations 920
nesting object calls 948
query 264
Request 263
Response 263
using properties 929, 947
OLE/COM Object Viewer 950
onAddBuddyRequest method
described 1052
example 1056
onAddBuddyResponse method, example 1056
onApplicationEnd event handler 285
onapplicationEnd method, using 287
onApplicationStart event handler 285
onBlur validation
considerations 663
described 660
onBuddyStatus method
described 1052
example 1056
onError event handler 285
onError method
using 290
onIMServerMessage method
described 1052
example 1056
onIncomingMessage method, for instant messages 1052
onIncomingMessage method, of listener CFCs 1030
onRequest event handler 285
onRequest method, using 289
onRequestEnd event handler 285
onRequestEnd method, using 290

onRequestStart event handlerrequests
application events 285
onRequestStart method, using 289
onServer validation
considerations 663
described 661
onSessionEnd method
using 288
onSessionStart method
locking Session scope 288
using 288
onSubmit validation
considerations 663
described 661
opening, SQL Builder 462
operands 34
operators 34
alternative notation 77
arithmetic 76
Boolean 76
comparison 77
concatenation 78
concept 593
decision, or comparison 77
evidence 597
explicit queries 584
precedence 79
proximity 598
relational 594
score 600
search 592
SQL 454
string operators 78
types 76
optimizing
applications 298
caching 298
database access 303
optional arguments
about 174, 184
in functions 80
OR SQL operator 454
ORDER BY SQL clause 453, 456
out parameters 895
outgoingMessage Gateway interface method
example 1099
signature and description 1087
outgoingMessage method, implementing gateways with 1098
output, displaying in components 213
outputting
debug information 266
query data 469
overriding default Java applet values 656

P

packages, component 233
page character encoding, determining 411
page execution time
about 428
tree format 429
page numbers, Report Builder 788
page processing settings 284
page settings 296
page, Flash form cfformgroup element 693
PageContext 920
pages
cache flushing 300
caching 299
panel, Flash form cfformgroup element 693
parameters
of component methods 209
passing in direct method invocations 226
passing using cfinvoke 225
parent tags 254
passing
arguments 179
arrays to user-defined functions 179
custom tag attributes 245, 246
custom tag data 254
parameters to a report 792
queries to user-defined functions 197
password
getting 383
PDF 768
paths, custom tags 242
PDF
advanced security options 768
cfdocument tag 764
cfreport tag 791
displaying external web pages 768
PDU
defined 1066
SMS gateway handling of 1070
perform a query on a query 492
performance, improving COM object 955, 958
Perl
regular expression compliance 158
WDDX and 875

permissions, IM GatewayManager management methods 1060
persistent scope variables 340
persistent scopes
components in 234
in event gateway listener CFCs 1032
persistent variables
in clustered system 342
issues 341
scopes 341
using 341
phone directory lookup, example CFC 1054
physical fonts, mapping from logical 804
POP, getting e-mail with 985
populating
arrays from queries 107
arrays with ArraySet 105
arrays with cfloop 105
arrays with nested loops 106
ports
securing access to 376
Post method, cfhttp 836, 838, 992, 996
pound signs. *See* number signs
precedence rules, search 592
precedence, operator 79
prefix notation, search strings 591
preservedata cfform attribute 633
preview, Report Builder 787
printable output
about 763
cfdocument tag 764
cfreport tag 770
font management 804
saving to a file 770
PrintWhen expression, Report Builder 789
problems, troubleshooting 443
processing
form variables on action pages 613
Java CFX requests 263
profiling, cftimer tag 440
protecting data 360
proxies, for COM objects 955
proximity operators 598
punctuation, searching 586
Python, WDDX and 875

Q

queries
about 736
as function parameters 197
as objects 489
as variables 53
building 453, 468
charting 736
converting to XML 868
creating from text files 994
defining query fields in Report Builder 782
graphing 736
grouping output 636
guidelines for outputting 470
outputting 469
referencing 54
scopes 55
syntax 466
troubleshooting 444
using form variables 614
validating 666
web services, consuming 897
web services, publishing 912, 913
XML and 867
queries of queries
aggregate functions 506
aliases 499, 508
benefits 491
case sensitivity 503
cfdump tag and 495
combining record sets 496
conditional operators 500
displaying record sets 493
escaping reserved words 509
escaping wildcards 503
evaluation order 499
example 492
joins 498
non-SQL record sets and 495
null support 509
ORDER BY clause 507
performing 492
reserved words 509
unions 498
user guide 497
using 489
Query Builder
about 795
advanced query mode 797
cfquery attributes 797

Query CFX object 264
query columns 55
query fields, defining in Report Builder 782
query functions 489
Query object 264
query objects 53, 489
Query of Queries
  column comparison 512
  data types for columns 504
  dates 511
  performance 511
  string concatenation operators 509
query properties, guidelines for 472
query results
  about 471
  cfpop 579
  columns in 471
  displaying 469
  indexing 548
  LDAP 578
  no records 621
  records returned 471
  returning 621
  returning incrementally 623
  variables 471
query variables 53
QueryAddColumn() CFML function 504
querying, LDAP directories 521
queryNew() CFML function 490, 504
quotation marks
  for IsDefined CFML function 71
  using 71, 467

R

Rand CFML function 625
RandRange CFML function 625
RDN (Relative Distinguished Names) 518
RDS
  configuring for Report Builder 792
  NTLM security 379
  Report Builder installation 775
read-only locks 363
reading, a text file 1010
real number variables 47
receiving e-mail 983
record sets 455
  combining 496
  creating 489
  displaying 493
  example 490
  queries of queries 489
  searching 572
  with functions 490
records 447
recoverable expressions 310
recursion, with user-defined functions 198
referencing array elements 100
referrals, LDAP 543
REFind CFML function 153
REFindNoCase CFML function 153
registering
  CFX tags 270
  COM objects 949
  CORBA objects 962
  Java applets 656
regular expressions
  backreferences 150, 674
  basic syntax 143
  case sensitivity 145
  character classes 149
  character sets 143
  common uses 157
  escape sequences 148
  examples 157, 674
  for form validation 671
  for searching and replacing text 141
  for validating 666
  hyphens in 148
  minimal matching 155
  partial matches 674
  Perl compliance 158
  repeating characters 144
  replacing with 141
  returning matched subexpressions 153
  single-character 143, 672
  special characters 144, 146
  technologies 158
  validating data with 671
relational operators 594
release, COM objects 953
ReleaseCOMObject function 953
remote component access 223
remote servers 991
removeBuddy IM GatewayHelper method 1059
removeDeny IM GatewayHelper method 1060
removePermit IM GatewayHelper method 1060
renaming server files 1009
Replace CFML function 141
replacing using regular expressions 141

report bands
adding fields 784
placing toolbox elements 784
Report Builder
advanced query mode 797
alignment 786
calculated fields 782
CFML 797
charting 800
common tasks 779
configuration 776
definition guidelines 779
displaying CFRs in a browser 791
displaying reports 791
expressions 799
fields 782
grouping 780
hyperlinks 789
input parameters 782
installing 775
layered controls 789
page numbers 788
passing variables to a report 792
preview 787
Properties sheet 790
Query Builder 795
RDS configuration 792
report creation steps 778
Report Creation Wizard 794
Setup Wizard 776
subreports 800
text styles 777, 786
user interface 776
Report Creation Wizard 794
Report Function Editor 798
report functions, Report Builder 798
Report Layout panel 794
Report Name panel 795
Report Style panel 794
Report Theme panel 795
Report to Subreport Parameter Binding panel 801
reporting
cfdocument 765
cfreport 770
request headers, web services 907
Request object
about 263
Java CFX 263
viewing headers for web services 915
Request scope
about 33, 65, 255
Java case considerations 920
sharing with JSP pages, servlets 923
user-defined functions and 196
request, error handler page 317
requests
application events 278
managing with Application.cfc 288
requests, globalization and 410
requiring form entries 616
reserved words
CFScript 127
in CFML 39
list of 39
reset buttons 611
resolving
ambiguous data types 935
custom tag file conflicts 244, 249
filename conflicts 1006
resource security
about 375
resources 375
sandbox security and 375
resources, regular expressions 675
resources, securing access to 375
responder applications, event gateway 1029
response headers, web services 907
Response object 263, 264
restart
Gateway interface method
signature and description 1087
restart Gateway interface method
implementing 1096
results
providing from components 213
returning from components 213
returning incrementally 623
retrieving
binary files 992
files 999
query data 466
text 992
retrieving, e-mail messages 987
return CFScript statement 174
returning
file information 1012
query results 621
results incrementally 623
subexpressions 153

reusing code
cfinclude 241
custom tags 241
method comparison 169
methods 161
options 161
techniques 161
with components 230
role-based security, in components 235
roles
about 377
checking 387
described 377
obtaining 377
setting 387
rollbacks 451
rows in tables 448
rpc web services, consuming 894
runtime exceptions 310

S

Sametime, about 1048
SAMETIMEGateway class
defined 1048
sample CFX tags
C++ 269
Java 260
sandbox security
about 375
applications and 376
resources 375
using 376
saving
binary files 994
components 215
web pages 993
SBCS 406
schema, LDAP directory 539
schemas, validating XML with 857
scopes
about 64
Application 66, 277, 340, 357
Arguments 66
as structures 70
Attributes 65
Caller 65
CFX tags 69
CGI 65
Client 66, 277, 340, 343, 346
Cookie 66
debug output 433
evaluating 69
File 250
Flash 66
Form 65
function local 66
LDAP 521
locking 364
managing locking of 366
of Form variables 617
persistent components 234
persistent variables 340
Request 65, 255
Server 66, 277, 340, 359
Session 66, 277, 340, 343, 352
This 66
This, in components 227
ThisTag 65
types 65
URL 65
user-defined functions and 185
using 69
variables 55, 65, 67
Variables, in components 228
score search operators 600
scriptprotect, cfapplication attributes. 664
search criteria, multiple 619
search expressions
case sensitivity 591
commas in 591
composing 591
delimiters 592
operators 592
with wildcards 585
searching
case sensitivity 591
cfsearch tag 561
character encodings 421
collections 548
collections, creating 553
database records 572
fields 601
file types 549
for special characters 586
full-text 547
index summaries, creating 564
international languages 551
LDAP query results 578
modifiers 601
numeric values 627, 629

operators 592
performing 561
prefix and infix notation 591
punctuation 586
query results 578
record sets 572
refining 601
results of 561
search expressions 591
special characters 586
string values 628, 630
wildcards for 585
zones 601
searching e-mail 579
Secure Sockets Layer (SSL)
keytool utility 544
LDAP server security 544
web server authentication 379
securing, custom tags 244, 249
security
and components 235
and data validation 664
application 279
authentication 379
authentication storage and persistence 380
ColdFusion features for 374
ColdFusion MX features 374
cross site-scripting 664
Flash form data 707
flow of control 377
functions 381
implementing application-based 394
implementing web server-based 388
LDAP and 399
logout 385
of sessions 353
of validation techniques 663
resource and sandbox 375
resources 375
role-based, in components 235
roles 377
scenarios 386
scope of login 382
specifying resources 375
tags 381
types of 374
user security 376
web servers and 378, 906
web services 905, 906
without cookies 381
See also authentication
See also login
See also resource security
See also sandbox security
See also user security
security exceptions 311
SELECT SQL statement 453, 455
selection lists, multiple 629
SendGatewayMessage function
about 1035
for sending instant messages 1053
sending
e-mail 974, 975, 979
form-based e-mail 978
mail as HTML
HTML
sending e-mail as 976
multipart e-mail 977
query-based e-mail 978
server configuration 24
Server scope 34, 66, 277, 340
server variables
about 277, 340
built-in 359
using 359
servers
remote 992, 999
retrieving files from 992
securing access to 376
uploading files 1004
servlets
ColdFusion MX and 918
in ColdFusion applications 922
variables and CFML 920
Session scope
about 34, 66, 277, 340
for authentication information 380
in event gateway listener CFCs 1033
sharing with JSP pages, servlets 923
Session variables
about 34, 66, 277, 340, 343, 354
built-in 355
enabling 354
using 352
sessions
and applications 276
application events 278, 285
defined 352
Java case considerations for unnamed 920
managing with Application.cfc 288

SessionStart event handler 285
setCFCListeners Gateway interface method
  implementing in a gateway class 1093
  signature and description 1087
setCFCMethod, CFEvent class method 1089
setCFCPath, CFEvent class method 1089
setCFCTimeout, CFEvent class method 1089
setData, CFEvent class method 1089
SetEncoding CFML function 418
setGateway Gateway interface method
  signature and description 1087
setGatewayID Gateway interface method
  implementing in a gateway class 1093
SetLocale CFML function 409
setNickName IM GatewayHelper method 1059
setOriginatorID, CFEvent class method 1089
setPermitMode IM GatewayHelper method 1060
setStatus IM GatewayHelper method 1059
setting
  application defaults 287, 296
  file and directory attributes 1007
setting up
  C++ development environment 269
  Java development environment 260
  Report Builder setup wizard 776
settings, application-level 278
Setup Wizard, Report Builder 776
SetVariable CFML functions 90
Shift-JIS 406
SHLIB_PATH
  about 269
  C++ CFX tags 269
shopping cart, example 92
Short Message Service. *See* SMS
shorthand notation, for Boolean operators 77
simple queries 582, 583
simple variables 45
single-byte character set 406
single-character regular expressions 143, 672
single-quotation marks, in SQL 467, 630
size, setting Flash form 695
skins
  extending ColdFusion 729
  Flash attributes 700
  role of XSLT 710
  setting in Flash forms 700
  XSLT types 711
slider bar controls 643
SME Delivery Acknowledgment, described 1074
SME Manual/User Acknowledgment, described 1074
SMPP, defined 1066
asynchronous mode
SMS
  about 1066
  client simulator 1081
  ColdFusion application tools 1068
  configuring the event gateway 1071
  determining message type 1074
  development and deployment process 1068
  handling incoming messages 1073
  interaction between gateway and SMSC 1069
  message disposition notification 1080
  message validity period 1080
  providers 1069
  purpose of synchronous mode 1079
  requesting message disposition information 1074
  sample application 1082
  See also synchronous mode
  sending messages 1070, 1075
  test SMSC server 1081
  uses of 1065
  using ColdFusion for 1065
SMSC
  ColdFusion simulator 1081
  defined 1066
  interaction with SMS event gateway 1069
SMSC Delivery Receipt, described 1074
SMTP 974
SOAP
  about 887
  defined 887
  headers 907
  troubleshooting 914
  web services and 887
Social Security Numbers, validating 665
SocketGateway class
  listener code 1097
SocketGateway example gateway 1040
SocketHelper GatewayHelper example class 1041
special characters 586, 672
  entering 39
  explicit queries 586
  in regular expressions 672
  list 39
specifying, tree items in URLs 642
SQL
  AVG function 739
  case sensitivity 454
  column aliases 457
  debugging output 430

DELETE statement 459, 487
Dreamweaver MX for 460
example 452
filtering 456
generating dynamically 614
guidelines 455
INSERT statement 458, 479
introduction 447, 452
nonstandard 455
operators 454
ORDER BY clause 456
ordering results 456
query editors 460
Query of Queries 497
record sets 455
results 455
SELECT statement 455
single quotation marks in 467, 630
sorting 456
statement clauses 453
statements 453
SUM function 753
syntax 453
text literals in 467
UPDATE statement 459, 480
use in cfquery 466
WHERE clause 456, 614
writing 452
standard variables. *See* built-in variables
start Gateway interface method
implementing 1094
signature and description 1087
startGateway, GenericGateway class method 1091
statement clauses, SQL 453
statements
CFScript 125
SQL 453
status output, with user-defined functions 187
stemming
preventing 583
simple queries 583
stop Gateway interface method
implementing 1095
signature and description 1087
stopGateway, GenericGateway class method 1091
stored procedures 303
string concatenation operators, Query of Queries 509
string operators 78
string variables 47
strings
empty 47
escaping 47
evaluating in functions 198
quoting 47
storing complex data in 883
validating 666
variables 47
StructClear CFML function 116
StructCount CFML function 113
StructDelete CFML function 116
StructIsEmpty CFML function 113
StructKeyArray CFML function 114
StructKeyExists CFML function 113
StructKeyList CFML function 114
StructNew CFML function 112
structures
about 108
adding data to 112
as variables 52
associative array notation 109
copying 114
creating 111
custom tag 246
declaring 963
deleting 116
dot notation 109
example 117
finding keys 113
functions 121
getting information on 113
in dynamic expressions 86
listing keys in 114
looping through 117
notation for 109
passing tag arguments 248
referencing 54
scopes and 55, 70
sorting keys 114
updating 112
validating 666
web services, consuming 910
web services, publishing 912
structuring, component code 230
styles
Flash syntax for 701
inheritance in Flash 702
setting in Flash forms 700
value formats in Flash 701
sub tags, defined 254

submit buttons 611
submit command, SMS 1075
submitMulti command, SMS 1077
Subreport Binding panel 803
Subreport Filename panel 803
Subreport Source panel 801, 802
subreports
about 800
adding new 802
defining 801
modifying 803
parameters, with input parameters 792
using existing 801
SUM SQL function 753
summaries, search 564
Super keyword 230
switch-case, CFScript 130
synchronization, SMS message sending 1079
synchronous mode
defined 1070
example 1079
purpose of 1079
sending SMS messages using 1079
syntax
errors in CFML 443
for Flash styles 701

T

tables
about 447
displaying queries 617
using HTML 617
tabnavigator
example 697
Flash form cfformgroup element 693
tag libraries 921
tags
built in 29
cfargument 662, 664, 670
cfformgroup in Flash forms 693
cffunction 382
cflogin 381, 382
cfloginuser 381
cflogout 382
cfNTauthenticate 382
cfparam 662, 664, 670
custom 30
for components 204
for security 381
securing access to 375
syntax 29
TCP network directory services 520
TCPMonitor 915
telephone numbers, validating 665
TemperatureService web service 891
template errors 311
testing, a variable's existence 616
text box, Report Builder 784
text control 611
text files
column headings 995
creating queries from 994
delimiters 994
text styles, Report Builder 777, 786
text, adding to Flash forms 690
textformat tag, in Flash forms 691
This scope
about 34, 66
in components 227
ThisTag scope 33, 65
throwOnTimeout, cflock attribute 365
tile, Flash form cfformgroup element 693
time
globalization functions 414
validating 665
time zone processing, WDDX 878
time-out attribute, cflock 365
timing, cftimer tag 440
toolbox, Report Builder 776
tools
for developing event gateways 1025
SMS application development 1068
ToString CFML function 851
totalpagecount, cfdocument scope variable 767
tracing
cftrace tag 436
considerations for 438
enabling 424
format 437
messages 437
options 436
output 432, 436
transactions 451
transferring data, from browser to server 881
tree controls, structuring 638

troubleshooting
CFML syntax 443
character encoding issues 408
common problems 443
data sources 444
HTTP 444
SOAP headers 914

U

UCS-2 406
UDDI
about 887
defined 888
UDF. *See* user-defined functions
UIDs, validating 665
Unicode
character encoding 407
ColdFusion and 407
unions, queries of queries 498
Universal Description, Discovery and Integration 888
UNIX
permissions 1008
UPDATE SQL statement 459
updating
data using forms 480
database with cfgridupdate 651
database with cfquery 652
files 360
values in structures 112
uploading files
about 1004
controlling file type 1006
URL scope 33, 65
URLEncodedFormat CFML function 444
URLs
character sets 417
encoding 417
invoking components with 222
validating 665
user authentication, login forms 383
user edits, returning 648
user ID, getting 383, 387
user roles 377
user security
about 376
application-based authentication example 394
authenticating users 378
defined 374
Dreamweaver login wizard 388
LDAP authentication example 399
RDS and NTLM 379
web server authentication example 391
See also authentication
See also security
user-defined functions
argument naming 178
arguments 179, 196
Arguments scope and 181, 182
array arguments 179
calling 164, 175, 178
CFML tags in 177
CFScript syntax 173
creating 172, 174
creation rules 177
defining 173
description 171
effective use of 194
error handling 186
evaluating strings 198
example 175, 192
exception handling 190
function-only variables 185
generating exceptions 191
identifying 198
in Application.cfm 194
Java and 937
passing arrays 179
queries as arguments 197
recommendations for 164
recursion 198
report functions in Report Builder 798
Request scope and 196
status output 187
using with queries 194
variables 185
users
authenticating 289
keeping track of 342
UTF-8 406
utility functions, for applications. 286

V

validating
data types 73
form field data types 630, 675
form input 638
JavaScript functions 680
using regular expressions 671
XML text 857

validation
about 659
error handler page 317
security considerations 664
techniques 660
validation, error handling 682
validity period, of SMS messages 1080
var keyword 228, 229
var, CFScript statement 174
variable names, periods in 56, 58
variable naming 44
variable scopes
about 33
Application 34
Arguments 34
Attributes 33
Caller 33
CGI 33
Client 34
Cookie 33
Flash 34
Form 33
function local scope 34
Request 33
Server 34
Sessions 34
This 34
ThisTag 33
URL 33
Variables 33
variables
Application 358
Application scope 277, 340, 357
array 52
Base64 51
binary 45, 51
Boolean 49
caching 351
CFScript 125
client 58
Client scope 277, 340, 343
complex 52
complex data type 45
component 227
configuring client 346
cookie 58
creating 44
data types 33, 35
date-time 49
default 71, 72, 296
dynamic naming 84
ensuring existence of 70
evaluating 59
formatting 619
forms 613
function local in components 229
getting 56
in user-defined functions 185
integer 47
kinds of 33
lists 48
locking example 368
naming 246
naming rules 44
numeric 46
objects 45
passing 836, 992
persistent 340
queries 53
real numbers 47
Request scope 255
scopes 33, 55, 67
scopes for custom pages 255
sending 838, 996
Server 359
Server scope 277, 340
Session scope 277, 340, 343, 352, 354
setting 57
setting default values 72
shared 277
simple 45
string 47
structures 52
testing for existence 70, 72, 616
using CFML with Java and JSP 920
validating names 666
See also built-in variables
Variables scope
about 33, 65
and onRequestStart 289
in components 228
vbox, Flash form cfformgroup element 693
vdividedbox, Flash form cfformgroup element 693
verbs, SQL 453
Verity
case sensitivity 591
explicit queries 583
query types 581
refining search 601
searching with 549

simple queries 582
zone filter 602
Verity Search engine exception 311
vertical, Flash form cfformgroup element 693

W

watching changes to 1041
WDDX
character encodings 420
components 875
converting CFML to JavaScript 880
exchanging data 876
operation of 876
purpose of 875
storing data in strings 883
time zone processing 878
transferring data 881
web
accessing with cfhttp 875, 992
application framework 342
web application servers
request handling 21
tasks 21
web servers and 21
web applications, and ColdFusion applications 276
web pages
dynamic 465
saving 993
static 465
web server-based authentication
example 391
scenario 386
web servers
about 20
Apache 20
basic authorization 379
IIS 20
security 378
user authentication 378
web services
accessing 886
basic authentication and 906
CFScript and 893
ColdFusion MX Administrator 896
complex data types 909
components for 898
concepts 887
consuming 886, 891
document-literal, consuming 894
document-literal, publishing 904
Dreamweaver MX and 888
error handling 895
Flash Remoting and 895
introduction 885
invoking component methods as 224
omitting an attribute 893
parameter passing 891
publishing 886, 898
request headers 907
response headers 907
return values 892
rpc-encoded, consuming 894
securing 905
SOAP and 887
TemperatureService 891
type conversions 896
UDDI and 887
WSDL file
Web Services Description Language file
about 886
See also WSDL
web services, consuming
about 886
cfinvoke tag 891, 892
CFScript for 893
ColdFusion MX 897
ColdFusion MX Administrator 896
complex data types 909
CreateObject function 893, 894
error handling 895
example 892
inout parameters 895
methods for 891
not ColdFusion 894
out parameters 895
parameter passing 891
queries 897
return values 892
structures 897, 910
type conversions 896
web services, publishing
about 886, 898
best practices for 907
complex data types 912
components and 898
components as data types 902
data types for 899
example 902
queries 912
requirements 899

securing 905
structures 912
WSDL files 900
web services, security
about 906
example 906
in ColdFusion 906
programmatic 236
using web servers 906
websites, indexing 548
WHERE SQL clause
about 456
comparing with 614
description 453
while loop, CFScript 133
wildcards, in searches 585
Windows file attributes 1007
Windows NT, debugging C++ CFX tags 270
writing SQL statements 463
WSDL files
components 890
creating 888
defined
described 888
reading 889
viewing in Dreamweaver MX 888
web services, publishing 900

X

XForms, and ColdFusion forms 710
XML
basic document view 845
bind elements for skinnable forms 718
ColdFusion and 843
converting to query 867
data model for skinnable forms 716
DOM node view 845
elements, locating 862
example 870
example from CFML form 726
form control element structure 721
format for skinnable forms 715
functions 850
mapping CFML tags to 720
queries and 867
structure for cfform, example 726
using 844
validating 857
xf:submission element for forms 717
XML document object 844
See also XML skinnable forms
XML document object
assigning data to 853
basic view 845
changing 861
converting to query 867
creating 856, 857
defined 844
deleting 860
DOM node view 845
example 870
exporting 857
extracting data with XPath 869
modifying 858
reference syntax 853
referencing case-sensitive objects 852
referencing summary 860
saving 856
structure 847, 848
syntax for referencing 852
transforming, XSLT 869
using 852
XmlComment 848
XmlDocType 848
XmlRoot 848
XSLT 869
XML elements
accessing using an array 859
adding 863
attributes 866
child elements 862
copying 864
counting 862
deleting 865
finding 862
properties, modifying 866
replacing 866
XmlAttributes 848
XmlChildren 848
XmlComment 848
XmlName 848
XmlNodes 848
XmlNsPrefix 848
XmlNsURI 848
XmlParent 848
XmlText 848
XML instance element for skinnable forms 717
XML schemas, validating with 857

XML skinnable forms
about 709
attribute passthrough to XML 729
building 712
creating skins 728
example 714
extending ColdFusion skins 729
generated XML example 726
styling with CSS files
validating data 668
XML format 715
XML structure for CFML tags 722
See also XML
XmlAttributes 848
XmlChildPos CFML function 851
XmlChildren 848
XmlComment 848
XmlDocType 848
XmlElemNew CFML function 850
XmlFormat CFML function 851
XmlGetNodeType CFML function 851
XmlName 848
XmlNew CFML function 850
XmlNew function 856
XmlNodes 848
XmlNsPrefix 848
XmlNsURI 848
XmlParent 848
XmlParse CFML function 850
XmlParse function 857
XmlRoot 848
XmlSearch CFML function 850
XmlText 848
XmlTransform CFML function 850
XmlType 849
XmlValidate CFML function 851
XmlValue 849
XMPP, about 1048
XMPPGateway class, defined 1048
XPath
extracting XML data 869
XSL transformation with 869
XSLT
and ColdFusion forms 710
example 869
transforming XML documents 869
XSLT skins, creating 728

Z

ZIP codes, validating 665
zone searches 602