



macromedia®
COLDFUSION®
MX

Developing ColdFusion MX Applications
with CFML



Trademarks

Afterburner, AppletAce, Attain, Attain Enterprise Learning System, Attain Essentials, Attain Objects for Dreamweaver, Authorware, Authorware Attain, Authorware Interactive Studio, Authorware Star, Authorware Synergy, Backstage, Backstage Designer, Backstage Desktop Studio, Backstage Enterprise Studio, Backstage Internet Studio, ColdFusion, Design in Motion, Director, Director Multimedia Studio, Doc Around the Clock, Dreamweaver, Dreamweaver Attain, Drumbeat, Drumbeat 2000, Extreme 3D, Fireworks, Flash, Fontographer, FreeHand, FreeHand Graphics Studio, Generator, Generator Developer's Studio, Generator Dynamic Graphics Server, JRun, Knowledge Objects, Knowledge Stream, Knowledge Track, Lingo, Live Effects, Macromedia, Macromedia M Logo & Design, Macromedia Flash, Macromedia Xres, Macromind, Macromind Action, MAGIC, Mediamaker, Object Authoring, Power Applets, Priority Access, Roundtrip HTML, Scriptlets, SoundEdit, ShockRave, Shockmachine, Shockwave, Shockwave Remote, Shockwave Internet Studio, Showcase, Tools to Power Your Ideas, Universal Media, Virtuoso, Web Design 101, Whirlwind and Xtra are 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, servicemarks, or tradenames of Macromedia, Inc. or other entities and may be registered in certain jurisdictions including internationally.

This product includes code licensed from RSA Data Security.

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.

Apple Disclaimer

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

Copyright © 1999–2002 Macromedia, Inc. All rights reserved. This manual may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without prior written approval of Macromedia, Inc.
Part Number ZCF60M800

Acknowledgments

Project Management: Stephen M. Gilson

Writing: Hal Lichtin, Stephen M. Gilson, Michael Stillman, David Golden

Editing: Linda Adler, Noreen Maher

First Edition: May 2002

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

CONTENTS

ABOUT THIS BOOK	XXI
Using this book	xxii
Book structure and contents	xxii
Approaches to using this book	xxii
Developer resources	xxiv
About Macromedia ColdFusion MX documentation	xxv
Printed and online documentation set	xxv
Viewing online documentation	xxvi
Getting answers	xxvi
Contacting Macromedia	xxvi
CHAPTER 1 Introduction to ColdFusion MX	1
About Internet applications and web application servers	2
About web pages and Internet applications	2
About web application servers	2
About ColdFusion MX	4
The ColdFusion scripting environment	4
ColdFusion Markup Language	4
ColdFusion application services	6
The ColdFusion MX Administrator	6
Using ColdFusion MX with Macromedia Flash MX	7
About J2EE and the ColdFusion architecture	8
About ColdFusion and the J2EE platform	8
J2EE infrastructure services and J2EE application server	8
ColdFusion features described in this book	10
PART I The CFML Programming Language	13
CHAPTER 2 Elements of CFML	15
Introduction	16
Character case	16
Tags	17
Tag syntax	17
Built-in tags	17
Custom tags	18

Functions	19
Built-in functions	19
User-defined functions	19
Expressions	21
Constants	21
Variables	22
Variable scopes	22
Data types	24
ColdFusion components	25
CFScript	26
Flow control	27
cfif, cfelseif, and cfelse	27
cfswitch, cfcase, and cfdefaultcase	28
cfloop and cfbreak	28
cfabort and cfexit	30
Comments	31
Special characters	31
Reserved words	32

CHAPTER 3 Using ColdFusion Variables 33

Creating variables	34
Variable naming rules	34
Variable characteristics	35
Data types	35
Numbers	36
Strings	37
Booleans	38
Date-Time values	39
Binary data type and Base64 encoding	40
Complex data types	41
Using periods in variable references	45
Understanding variables and periods	46
Creating variables with periods	47
Data type conversion	49
Operation-driven evaluation	49
Conversion between types	49
Evaluation and type conversion issues	51
Examples of type conversion in expression evaluation	54
About scopes	55
Scope types	55
Creating and using variables in scopes	57
Using scopes	59
Ensuring variable existence	60
Testing for a variable's existence	60
Using the cfparam tag	61
Validating data types	62
Using cfparam to validate the data type	62
Passing variables to custom tags and UDFs	64
Passing variables to CFML tags and UDFs	64
Passing variables to CFX tags	64

CHAPTER 4 Using Expressions and Pound Signs	65
Expressions	66
Operator types	66
Operator precedence and evaluation ordering	69
Using functions as operators	69
Using pound signs	71
Using pound signs in ColdFusion tag attribute values	71
Using pound signs in tag bodies	72
Using pound signs in strings	72
Nested pound signs	73
Using pound signs in expressions	74
Dynamic expressions and dynamic variables	74
About dynamic variables	74
About dynamic expressions and dynamic evaluation	74
Dynamic variable naming without dynamic evaluation	75
Using dynamic evaluation	77
Using the IIF function	80
Example: a dynamic shopping cart	82
CHAPTER 5 Using Arrays and Structures	87
About arrays	88
Basic array concepts	88
About ColdFusion arrays	88
Basic array techniques	90
Referencing array elements	90
Creating arrays	90
Adding elements to an array	92
Deleting elements from an array	93
Copying arrays	94
Populating arrays with data	95
Populating an array with the ArraySet function	95
Populating an array with the cfloop tag	95
Populating an array from a query	97
Array functions	98
About structures	99
Structure notation	99
Referencing complex structures	100
Creating and using structures	102
Creating structures	102
Adding data elements to structures	102
Updating values in structures	102
Getting information about structures and keys	103
Copying structures	105
Deleting structure elements and structures	107
Looping through structures	107
Structure example	109
Structure functions	113

CHAPTER 6 Extending ColdFusion Pages with CFML Scripting	115
About CFScript	116
Comparing tags and CFScript	116
The CFScript language	118
Identifying CFScript	118
Variables	118
Expressions	118
Statements	118
Statement blocks	119
Comments	119
Reserved words	120
Differences from JavaScript	120
CFScript limitation	120
CFScript functional equivalents to ColdFusion tags	120
Using CFScript statements	122
Using assignment statements and functions	122
Using conditional processing statements	122
Using looping statements	124
Handling exceptions	129
CFScript example	130
CHAPTER 7 Using Regular Expressions in Functions	133
About regular expressions	134
Using ColdFusion regular expression functions	134
Basic regular expression syntax	135
Regular expression syntax	136
Using character sets	136
Finding repeating characters	137
Case sensitivity in regular expressions	138
Using subexpressions	138
Using special characters	138
Using escape sequences	141
Using character classes	143
Using backreferences	144
Using backreferences in replacement strings	144
Omitting subexpressions from backreferences	146
Returning matched subexpressions	147
Specifying minimal matching	149
Regular expression examples	152
Regular expressions in CFML	152
Types of regular expression technologies	154
PART II Reusing CFML Code	155
CHAPTER 8 Reusing Code in ColdFusion Pages	157
About reusable CFML elements	158
Including pages with the cfinclude tag	158
Using the cfinclude tag	159

Recommended uses	160
Calling user-defined functions	161
Calling UDFs	161
Recommended uses	161
For more information	161
Using custom CFML tags	162
Calling custom CFML tags	162
Recommended uses	163
For more information	163
Using CFX tags	164
Calling CFX tags	164
Recommended uses	164
For more information	164
Using ColdFusion components	165
Creating and using ColdFusion components	165
Recommended uses	165
For more information	165
Selecting among ColdFusion code reuse methods	166

CHAPTER 9 Writing and Calling User-Defined Functions 167

About user-defined functions	168
Calling user-defined functions	169
Creating user-defined functions	169
Creating functions using CFScript	169
Creating functions using tags	170
Rules for function definitions	170
Defining functions in CFScript	174
Defining functions using the cffunction tag	177
Calling functions and using variables	180
Passing arguments	180
Referencing caller variables	180
Using function-only variables	181
Using arguments	181
A User-defined function example	182
Defining the function using CFScript	182
Defining the function using the cffunction tag	183
Using UDFs effectively	184
Using Application.cfm and function include files	184
Specifying the scope of a function	184
Using the Request scope for static variables and constants	186
Using function names as function arguments	186
Handling query results using UDFs	187
Identifying and checking for UDFs	188
Using the Evaluate function	188
Passing complex data	189
Using recursion	190
Handling errors in UDFs	191

CHAPTER 10 Creating and Using Custom CFML Tags	197
Creating custom tags	198
Creating and calling custom tags	198
Securing custom tags	201
Accessing existing custom tags	201
Passing data to custom tags	202
Passing values to and from custom tags	202
Using tag attributes summary	203
Custom tag example with attributes	204
Passing custom tag attributes using CFML structures	205
Managing custom tags	207
Securing custom tags	207
Encoding custom tags	207
Executing custom tags	208
Accessing tag instance data	208
Handling end tags	208
Processing body text	210
Terminating tag execution	211
Nesting custom tags	212
Passing data between nested custom tags	212
Variable scopes and special variables	213
High-level data exchange	213
CHAPTER 11 Building and Using ColdFusion Components	217
About ColdFusion components	218
Applying design patterns to component development	218
Building ColdFusion components	219
Defining component methods	220
Interacting with component methods	222
Invoking component methods	222
Passing parameters to component methods	226
Returning values from component methods	232
Using advanced ColdFusion component functionality	234
Building secure ColdFusion components	234
Using component packages	237
Using component inheritance	239
Using component metadata	240
CHAPTER 12 Building Custom CFXAPI Tags	243
What are CFX tags?	244
Before you begin developing CFX tags in Java	245
Sample Java CFX tags	245
Setting up your development environment to develop CFX tags in Java	245
Customizing and configuring Java	246
Writing a Java CFX tag	247
Calling the CFX tag from a ColdFusion page	247
Processing requests	248
Loading Java CFX classes	250
Automatic class reloading	250
Life cycle of Java CFX tags	251

ZipBrowser example	251
Approaches to debugging Java CFX tags	253
Outputting debugging information	253
Debugging in a Java IDE	253
Using the debugging classes	254
Developing CFX tags in C++.	256
Sample C++ CFX tags	256
Setting up your C++ development environment	256
Compiling C++ CFX tags	256
Locating your C++ library files on Unix	256
Implementing C++ CFX tags	256
Debugging C++ CFX tags	257
Registering CFX tags	257

PART III Developing CFML Applications 259

CHAPTER 13 Designing and Optimizing a ColdFusion Application 261

About applications.	262
Elements of a ColdFusion application	262
The application framework.	262
Application-level settings and functions	263
Reusable application elements	264
Shared variables	264
Application security and user identification	264
Mapping an application.	265
Processing the Application.cfm and OnRequestEnd.cfm pages.	265
Defining the directory structure	266
Creating the Application.cfm page.	268
Naming the application	268
Setting the client, application, and session variables options	268
Defining page processing settings	269
Setting application default variables and constants	269
Processing logins.	269
Handling errors	270
Example: an Application.cfm page	270
Optimizing ColdFusion applications	272
Caching ColdFusion pages that change infrequently	272
Caching parts of ColdFusion pages.	274
Optimizing database use.	277
Providing visual feedback to the user	280

CHAPTER 14 Handling Errors. 281

About error handling in ColdFusion	282
Understanding errors	283
About error causes and recovery	283
ColdFusion error types	284
About ColdFusion exceptions.	284
How ColdFusion handles errors	287

Error messages and the standard error format	289
Determining error-handling strategies	291
Handling missing template errors	291
Handling form field validation errors	291
Handling compiler exceptions	291
Handling runtime exceptions	292
Specifying custom error messages with cferror	293
Specifying a custom error page	293
Creating an error application page	294
Logging errors with the cflog tag	297
Handling runtime exceptions with ColdFusion tags	299
Exception-handling tags	299
Using cftry and cfcatch tags	299
Using cftry: an example	304
Using the cfthrow tag	308
Using the cfrethrow tag	309
Example: using nested tags, cfthrow, and cfrethrow	310

CHAPTER 15 Using Persistent Data and Locking 315

About persistent scope variables	316
ColdFusion persistent variables and ColdFusion structures	317
ColdFusion persistent variable issues	317
Managing the client state	318
About client and session variables	319
Maintaining client identity	320
Configuring and using client variables	323
Enabling client variables	323
Using client variables	325
Configuring and using session variables	328
What is a session?	328
Configuring and enabling session variables	329
Storing session data in session variables	330
Standard session variables	330
Getting a list of session variables	331
Creating and deleting session variables	331
Accessing and changing session variables	331
Ending a session	332
Configuring and using application variables	333
Configuring and enabling application variables	333
Storing application data in application variables	333
Using application variables	334
Using server variables	335
Locking code with cflock	336
Sample locking scenarios	336
Using the cflock tag with write-once variables	338
Using the cflock tag	338
Considering lock granularity	341
Nesting locks and avoiding deadlocks	341
Examples of cflock	343

CHAPTER 16 Securing Applications	347
ColdFusion security features	348
About resource security	349
About user security	351
Security tags and functions	353
About web server authentication and application authentication	353
Controlling ColdFusion login behavior	354
The cflogin structure	356
Using ColdFusion security without cookies	356
A basic authentication security scenario	356
An application authentication security scenario	357
Implementing user security	360
Basic authentication user security example	360
Application-based user security example	362
Using application-based security with a browser's login dialog	368
Using an LDAP Directory for security information	369
CHAPTER 17 Developing Globalized Applications	373
Introduction to globalization	374
Defining globalization	374
Importance of globalization ColdFusion applications	375
How ColdFusion supports globalization	375
Character sets and locales	375
About character encodings	377
The Unicode character encoding	377
Locales	378
Setting the locale	378
Processing a request in ColdFusion	379
Determining the character set of a ColdFusion page	379
Determining the character set of server output	380
Tags and functions for globalizing	382
Using tags for globalizing applications	382
Using functions for globalizing applications	382
Handling data in ColdFusion	385
Input data from URLs and HTML forms	385
Reading and writing file data	387
Databases	387
E-mail	387
HTTP	387
LDAP	388
WDDX	388
COM	388
CORBA	388
Searching and indexing	388
CHAPTER 18 Debugging and Troubleshooting Applications . . .	389
Configuring debugging in the ColdFusion MX Administrator	390
Debugging Settings page	390
Debugging IP addresses page	392

Using debugging information from browser pages	393
General debugging information	394
Execution Time	395
Database Activity	397
Exceptions	399
Trace points	399
Scope variables	400
Using the dockable.cfm output format	400
Controlling debugging information in CFML	402
Generating debugging information for an individual query	402
Controlling debugging output with the cfsetting tag	402
Using the IsDebugMode function to run code selectively	403
Using the cftrace tag to trace execution	404
About the cftrace tag	404
Using tracing	406
Calling the cftrace tag	407
Using the Code Compatibility Analyzer	409
Troubleshooting common problems	410
CFML syntax errors	410
Data source access and queries	411
HTTP/URL	411

PART IV Accessing and Using Data 413

CHAPTER 19 Introduction to Databases and SQL 415

What is a database?	416
Using multiple database tables	417
Database permissions	418
Commits, rollbacks, and transactions	418
Database design guidelines	419
Using SQL	420
SQL example	420
Basic SQL syntax elements	421
Reading data from a database	422
Modifying a database	425
Writing queries using an editor	428
Writing queries using Dreamweaver MX	428
Writing queries using ColdFusion Studio and Macromedia HomeSite+	430

CHAPTER 20 Accessing and Retrieving Data 433

Working with dynamic data	434
Retrieving data	435
The cfquery tag	435
The cfquery tag syntax	435
Building queries	436
Outputting query data	438
Query output notes and considerations	439
Getting information about query results	441
Query variable notes and considerations	442

Enhancing security with cfqueryparam	443
About query string parameters	443
Using cfqueryparam	443
CHAPTER 21 Updating Your Database	445
About updating your database	446
Inserting data	446
Creating an HTML insert form	446
Data entry form notes and considerations.	448
Creating an action page to insert data.	448
Updating data	452
Creating an update form.	452
Creating an action page to update data.	455
Deleting data	459
Deleting a single record	459
Deleting multiple records	460
CHAPTER 22 Using Query of Queries	461
About record sets	462
Referencing queries as objects	462
Creating a record set.	462
Creating a record set with a function	463
About Query of Queries	465
Benefits of Query of Queries	465
Performing a Query of Queries.	465
Query of Queries user guide	474
Using dot notation	474
Using joins	474
Using unions	474
Using conditional operators	477
Using aggregate functions.	480
Using group by and having expressions	481
Using ORDER BY clauses	481
Using aliases	482
Handling null values.	483
Escaping reserved keywords	483
BNF for Query of Queries.	486
CHAPTER 23 Managing LDAP Directories	489
About LDAP	490
The LDAP information structure	492
Entry	492
Attribute.	492
Distinguished name (DN)	493
Schema	493
Using LDAP with ColdFusion.	495
Querying an LDAP directory.	496
Scope	496
Search filter	496

Getting all the attributes of an entry	498
Example: querying an LDAP directory	498
Updating an LDAP directory	503
Adding a directory entry	503
Deleting a directory entry	509
Updating a directory entry	510
Adding and deleting attributes of a directory entry	512
Changing a directory entry's DN	513
Advanced topics	514
Specifying an attribute that includes a comma or semicolon	514
Using cldap output	514
Viewing a directory schema	514
Referrals	519
Managing LDAP security	520

CHAPTER 24 Building a Search Interface 521

About Verity	522
Using Verity with ColdFusion	522
Advantages of using Verity	523
Supported file types	523
Support for international languages	526
Creating a search tool for ColdFusion applications	528
Creating a collection with the ColdFusion MX Administrator	528
About indexing a collection	530
Indexing and building a search interface with the Verity Wizard	530
Creating a ColdFusion search tool programmatically	535
Using the cfsearch tag	542
Working with record sets	545
Indexing database record sets	545
Indexing cldap query results	549
Indexing cfpop query results	550
Using database-directed indexing	551

CHAPTER 25 Using Verity Search Expressions 553

About Verity query types	554
Using simple queries	555
Stemming in simple queries	555
Preventing stemming	557
Using explicit queries	558
Using AND, OR, and NOT	558
Using wildcards and special characters	559
Composing search expressions	562
Case sensitivity	562
Prefix and infix notation	562
Commas in expressions	562
Precedence rules	563
Delimiters in expressions	563
Operators and modifiers	563
Refining your searches with zones and fields	573

PART V Requesting and Presenting Information 577

CHAPTER 26 Retrieving and Formatting Data 579

Using forms to specify the data to retrieve 580

- HTML form tag syntax 580
- Form controls 581
- Form notes and considerations 584

Working with action pages 585

- Processing form variables on action pages 585
- Dynamically generating SQL statements 585
- Creating action pages 586
- Testing for a variable's existence 587
- Requiring users to enter values in form fields 588
- Form variable notes and considerations 588

Working with queries and data 589

- Using HTML tables to display query results 589
- Formatting individual data items 590
- Building flexible search interfaces 591

Returning results to the user 593

- Handling no query results 593
- Returning results incrementally 594

Dynamically populating list boxes 597

Creating dynamic check boxes and multiple-selection list boxes 599

- Check boxes 599
- Multiple selection lists 601

Validating form field data types 603

CHAPTER 27 Building Dynamic Forms 607

Creating forms with the cfform tag 608

- Using HTML and cfform 608
- The cfform controls 608
- Preserving input data with preservedata 609
- Browser considerations 610

Building tree controls with cftree 611

- Grouping output from a query 612
- The cftree form variables 613
- Input validation 614
- Structuring tree controls 614
- Image names in a cftree 616
- Embedding URLs in a cftree 617
- Specifying the tree item in the URL 618

Building drop-down list boxes 619

Building text input boxes 620

Building slider bar controls 621

Creating data grids with cfgrid 622

- Working with a data grid and entering data 622
- Creating an editable grid 624

Embedding Java applets	633
Registering a Java applet	633
Using cfapplet to embed an applet	635
Handling form variables from an applet	636
Input validation with cfform controls	637
Validating with regular expressions	637
Input validation with JavaScript	642
Handling failed validation	642
Example: validating an e-mail address	643
CHAPTER 28 Charting and Graphing Data	645
Creating a chart	646
Chart types	646
Creating a basic chart	647
Administering charts	649
Charting data	650
Charting a query	650
Charting individual data points	653
Combining a query and data points	654
Charting multiple data collections	654
Writing a chart to a variable	656
Controlling chart appearance	658
Common chart characteristics	658
Setting x-axis and y-axis characteristics	660
Creating a bar chart	661
Setting pie chart characteristics	662
Creating an area chart	664
Setting curve chart characteristics	666
Linking charts to URLs	667
Dynamically linking from a pie chart	667
Linking to JavaScript from a pie chart	670
CHAPTER 29 Using the Flash Remoting Service	673
About using the Flash Remoting service with ColdFusion	674
Planning your Flash application	674
Using the Flash Remoting service with ColdFusion pages	675
Using Flash with ColdFusion components	680
Using the Flash Remoting service with server-side ActionScript	682
Using the Flash Remoting service with ColdFusion Java objects	683
Handling errors with ColdFusion and Flash	684
PART VI Using Web Elements and External Objects	685
CHAPTER 30 Using XML and WDDX	687
About XML and ColdFusion	688
The XML document object	689
A simple XML document	689
Basic view	690

DOM node view	690
XML document structures	691
ColdFusion XML tag and functions	694
Using an XML object	696
Referencing the contents of an XML object	696
Assigning data to an XML object	697
Creating and saving an XML document object	698
Creating a new XML document object using the cfxml tag	698
Creating a new XML document object using the XmlNew function	698
Creating an XML document object from existing XML	699
Saving and exporting an XML document object	699
Modifying a ColdFusion XML object	700
Functions for XML object management	700
Treating elements with the same name as an array	701
XML document object management reference	702
Adding, deleting, and modifying XML elements	703
Using XML and ColdFusion queries	708
Transforming documents with XSLT	710
Extracting data with XPath	711
Example: using XML in a ColdFusion application	712
Moving complex data across the web with WDDX	717
Uses of WDDX	717
How WDDX works	718
Using WDDX	722
Using the cfwddx tag	722
Validating WDDX data	722
Using JavaScript objects	723
Converting CFML data to a JavaScript object	723
Transferring data from the browser to the server	723
Storing complex data in a string	726

CHAPTER 31 Using Web Services729

Web services	730
Accessing a web service	730
Basic web service concepts	731
Working with WSDL files	733
Creating a WSDL file	733
Viewing a WSDL file using Dreamweaver MX	733
Reading a WSDL file	734
Consuming web services	736
About the examples in this section	736
Passing parameters to a web service	736
Handling return values from a web service	737
Using cfinvoke to consume a web service	737
Using CFScript to consume a web service	739
Calling web services from a Flash client	740
Catching errors when consuming web services	740
Handling inout and out parameters	740
Configuring web services in the ColdFusion Administrator	741
Data conversions between ColdFusion and WSDL data types	741

Consuming ColdFusion web services	742
Publishing web services	744
Creating components for web services	744
Specifying data types of function arguments and return values	744
Producing WSDL files	745
Using ColdFusion components to define data types for web services	748
Securing your web services	749
Best practices for publishing web services	752
Handling complex data types	753
Consuming web services that use complex data types	753
Publishing web services that use complex data types	756

CHAPTER 32 Integrating J2EE and Java Elements in CFML Applications759

About ColdFusion, Java, and J2EE	760
About ColdFusion and client-side JavaScript and applets	760
About ColdFusion and JSP	760
About ColdFusion and Servlets	761
About ColdFusion and Java objects	761
Using JSP tags and tag libraries	762
Using a JSP tag in a ColdFusion page	762
Example: using the random tag library	763
Interoperating with JSP pages and servlets	764
Integrating JSP and servlets in a ColdFusion application	764
Examples: using JSP with CFML	766
Using Java objects	769
Using basic object techniques	769
Creating and using a simple Java class	771
Java and ColdFusion data type conversions	774
Handling Java exceptions	776
Examples: using Java with CFML	777

CHAPTER 33 Integrating COM and CORBA Objects in CFML Applications785

About COM and CORBA	786
About objects	786
About COM and DCOM	786
About CORBA	786
Creating and using objects	788
Creating objects	788
Using properties	788
Calling methods	788
Calling nested objects	789
Getting started with COM and DCOM	790
COM Requirements	790
Registering the object	790
Finding the component ProgID and methods	790
Using the OLE/COM Object Viewer	791

Creating and using COM objects	793
Connecting to COM objects	793
Setting properties and invoking methods	794
COM object considerations	794
Getting started with CORBA	797
Creating and using CORBA objects	797
Creating CORBA objects	797
Using CORBA objects in ColdFusion	799
Handling exceptions	804
CORBA example	805

PART VII Using External Resources 807

CHAPTER 34 Sending and Receiving E-Mail 809

Using ColdFusion with mail servers	810
Sending e-mail messages	811
Sending SMTP e-mail with cfmail	811
Sample uses of cfmail	813
Sending form-based e-mail	813
Sending query-based e-mail	813
Sending e-mail to multiple recipients	814
Customizing e-mail for multiple recipients	815
Using cfmailparam	817
Attaching files to a message	817
Adding a custom header to a message	817
Advanced sending options	818
Sending mail as HTML	818
Error logging and undelivered messages	818
Receiving e-mail messages	819
Using cfpop	819
The cfpop query variables	820
Handling POP mail	821

CHAPTER 35 Interacting with Remote Servers 829

About interacting with remote servers	830
Using cfhttp to interact with the web	830
Using the cfhttp Get method	830
Creating a query object from a text file	835
Using the cfhttp Post method	837
Performing file operations with cfftp	841
Caching connections across multiple pages	843
Connection actions and attributes	844

CHAPTER 36 Managing Files on the Server 845

About file management	846
Using cffile	846
Uploading files	846
Moving, renaming, copying, and deleting server files	852
Reading, writing, and appending to a text file	852

Using cfdirectory	856
Returning file information	856
Using cfcontent	858
About MIME types	858
Changing the MIME content type with cfcontent	858

INDEX863

ABOUT THIS BOOK

Developing ColdFusion Applications provides the tools needed to develop Internet applications using Macromedia ColdFusion MX. This book is intended for web application programmers who are learning ColdFusion MX or wish to extend 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 book](#)xxii
- [Developer resources](#) xxiv
- [About Macromedia ColdFusion MX documentation](#)..... xxv
- [Getting answers](#) xxvi
- [Contacting Macromedia](#) xxvi

Using this book

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

Book structure and contents

The book is divided into seven parts, as follows:

Part	Description
The CFML Programming Language	The Elements of CFML including variables, expressions, dynamic code, CFScript, and regular expressions.
Reusing CFML Code	Techniques for writing code once and using it many times, including the <code>cfinclude</code> tag, user-defined functions, custom CFML tags, ColdFusion components, and CFXAPI tags.
Developing CFML Applications	How to develop a complete ColdFusion application. Includes information on error handling, sharing data, locking code, securing access, internationalization, debugging, and troubleshooting.
Accessing and Using Data	Methods for accessing and using data sources, including an introduction to SQL and information on using SQL data bases, LDAP directory services, and the Verity search engine
Requesting and Presenting Information	How to dynamically request information from users and display results on the user's browser, including graphing data and providing data to Flash clients.
Using Web Elements and External Objects	How to use XML, Java objects including Enterprise JavaBeans, JSP pages, web services (including creating web services in ColdFusion), and COM and CORBA objects.
Using External Resources	Methods for getting and sending e-mail, accessing remote servers using HTTP and FTP, and accessing files and directories.

Each chapter includes basic information plus detailed coverage of the topic that should be of use to experienced ColdFusion developers.

Approaches to using this book

This section describes approaches to using this book 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 learning ColdFusion, a path such as the following might be most effective:

- 1 [Chapter 1](#) through [Chapter 4](#) to learn the basics of the XML language.
- 2 [Chapter 19](#) through [Chapter 21](#) to learn about using databases.
- 3 [Chapter 26](#) and [Chapter 27](#) to learn about requesting data from users.

At this point, you should have a basic understanding of the basic elements of ColdFusion and can create simple ColdFusion applications. To learn to produce more complete and robust applications, you could proceed with the following chapters.

- 4 [Chapter 13](#) through [Chapter 18](#) to learn how to build a complete ColdFusion application.
- 5 [Chapter 22](#) to learn how to use queries effectively.
- 6 [Chapter 5](#) through [Chapter 11](#) 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 MX features. In this case, you start with *Migrating ColdFusion 5 Applications*; then read selected chapters in this book. The following chapters document features that are new or substantially enhanced in ColdFusion MX:

- [Chapter 9, Writing and Calling User-Defined Functions](#)
- [Chapter 11, Building and Using ColdFusion Components](#)
- [Chapter 16, Securing Applications](#)
- [Chapter 17, Developing Globalized Applications](#)
- [Chapter 18, Debugging and Troubleshooting Applications](#)
- [Chapter 28, Charting and Graphing Data](#)
- [Chapter 29, Using the Flash Remoting Service](#)
- [Chapter 30, Using XML and WDDX](#)
- [Chapter 31, Using Web Services](#)
- [Chapter 32, Integrating J2EE and Java Elements in CFML Applications](#)

Nearly all chapters contain information that is new in ColdFusion MX, 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.

Developer resources

Macromedia, Inc. is committed to setting the standard for customer support in developer education, documentation, technical support, and professional services. The Macromedia website is designed to give you quick access to the entire range of online resources. The following table shows the locations of these resources:

Resource	Description	URL
Macromedia website	General information about Macromedia products and services	http://www.macromedia.com
Information on ColdFusion	Detailed product information on ColdFusion and related topics	http://www.macromedia.com/coldfusion
Macromedia ColdFusion Support Center	Professional support programs that Macromedia offers	http://www.macromedia.com/support/coldfusion
ColdFusion Online Forums	Access to experienced ColdFusion developers through participation in the Online Forums, where you can post messages and read replies on many subjects relating to ColdFusion	http://webforums.macromedia.com/coldfusion/
Installation Support	Support for installation-related issues for all Macromedia products	http://www.macromedia.com/support/coldfusion/installation.html
Training	Information about classes, on-site training, and online courses offered by Macromedia	http://www.macromedia.com/support/training
Developer Resources	All the resources that you need to stay on the cutting edge of ColdFusion development, including online discussion groups, Knowledge Base, technical papers, and more	http://www.macromedia.com/desdev/developer/
Reference Desk	Development tips, articles, documentation, and white papers	http://www.macromedia.com/v1/developer/TechnologyReference/index.cfm
Macromedia Alliance	Connection with the growing network of solution providers, application developers, resellers, and hosting services creating solutions with ColdFusion	http://www.macromedia.com/partners/

About Macromedia ColdFusion MX documentation

The ColdFusion documentation is designed to provide support for the complete spectrum of participants. The print and online versions are organized to let you quickly locate the information that you need. The ColdFusion online documentation is provided in HTML and Adobe Acrobat formats.

Printed and online documentation set

The ColdFusion documentation set consists of the following titles:

Book	Description
<i>Installing ColdFusion MX</i>	Describes system installation and basic configuration for Windows NT, Windows 2000, Solaris, Linux, and HP-UX.
<i>Administering ColdFusion MX</i>	Describes how to use the ColdFusion Administrator to manage the ColdFusion environment, including connecting to your data sources and configuring security for your applications.
<i>Developing ColdFusion MX Applications with CFML</i>	Describes how to develop your dynamic web applications, including retrieving and updating your data, using structures, and forms.
<i>Getting Started Building ColdFusion MX Applications</i>	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.
<i>Using Server-Side ActionScript in ColdFusion MX</i>	Describes how Macromedia Flash movies executing on a client browser can call ActionScript code running on the ColdFusion server. Includes examples of server-side ActionScript and a syntax guide for developing ActionScript pages on the server.
<i>Migrating ColdFusion 5 Applications</i>	Describes how to migrate a ColdFusion 5 application to ColdFusion MX. This book describes the code compatibility analyzer that evaluates your ColdFusion 5 code to determine any incompatibilities within it.
<i>CFML Reference</i>	Provides descriptions, syntax, usage, and code examples for all ColdFusion tags, functions, and variables.
<i>CFML Quick Reference</i>	A brief guide that shows the syntax of ColdFusion tags, functions, and variables.
<i>Working with Verity Tools</i>	Describes Verity search tools and utilities that you can use for configuring the Verity K2 Server search engine, as well as creating, managing, and troubleshooting Verity collections.
<i>Using ClusterCATS</i>	Describes how to use Macromedia ClusterCATS, the clustering technology that provides load-balancing and failover services to assure high availability for your web servers.

Viewing online documentation

All ColdFusion documentation is available online in HTML and Adobe Acrobat Portable Document Format (PDF) files. To view the HTML documentation, open the following URL on the web server running ColdFusion: http://web_root/cfdocs/dochome.htm.

ColdFusion documentation in Acrobat format is available on the ColdFusion product CD-ROM.

Getting answers

One of the best ways to solve particular programming problems is to tap into the vast expertise of the ColdFusion developer communities on the ColdFusion Forums. Other developers on the forum can help you figure out how to do just about anything with ColdFusion. The search facility can also help you search messages from the previous 12 months, allowing you to learn how others have solved a problem that you might be facing. The Forums is a great resource for learning ColdFusion, but it is also a great place to see the ColdFusion developer community in action.

Contacting Macromedia

Corporate
headquarters

Macromedia, Inc.
600 Townsend Street
San Francisco, CA 94103
Tel: 415.252.2000
Fax: 415.626.0554
Web: <http://www.macromedia.com>

Technical support

Macromedia offers a range of telephone and web-based support options. Go to <http://www.macromedia.com/support/coldfusion> for a complete description of technical support services.

You can make postings to the ColdFusion Support Forum (<http://webforums.macromedia.com/coldfusion>) at any time.

Sales

Toll Free: 888.939.2545
Tel: 617.219.2100
Fax: 617.219.2101
E-mail: sales@macromedia.com
Web: <http://www.macromedia.com/store>

CHAPTER 1

Introduction to ColdFusion MX

This chapter describes ColdFusion MX and the role it plays in Internet applications, including Flash MX based applications. It also provides an introduction to the topics discussed in this book.

Contents

- [About Internet applications and web application servers](#) 2
- [About ColdFusion MX](#)..... 4
- [Using ColdFusion MX with Macromedia Flash MX](#)..... 7
- [About J2EE and the ColdFusion architecture](#) 8
- [ColdFusion features described in this book](#)..... 10

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 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.

However, a web server's capabilities is 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.

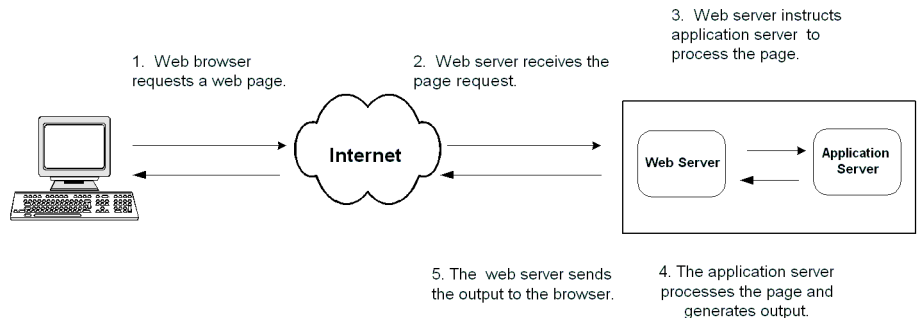
A web server, basically, locates information and returns it to a web browser.

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 Rich Internet Applications. ColdFusion MX 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
- ColdFusion Application Services
- The ColdFusion Administrator

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 consisting 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 15](#).

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 100 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, <code>cfpop</code> , 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 components, Java objects such as Enterprise JavaBeans, or CORBA objects.

CFML Reference describes the CFML tags in detail.

CFML functions and CFScript

CFML includes approximately 270 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 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, 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 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.

Server-side ActionScript

Another feature of the ColdFusion scripting environment is server-side ActionScript. ActionScript is the JavaScript-based language used to write application logic in Macromedia Flash MX. By bringing this language to the server, ColdFusion MX enables Flash developers to use their familiar scripting environment to connect to ColdFusion resources and deliver the results to client-side applications using the integrated Macromedia Flash Remoting service. Using server-side ActionScript Flash programmers can create ColdFusion services, such as SQL queries, for use by Flash clients.

For more information about using Server-Side ActionScript in ColdFusion MX, see *Using Server-Side ActionScript in ColdFusion MX*.

ColdFusion application services

The ColdFusion application services are a set of built-in services that extend the capabilities of the ColdFusion scripting environment. These services include the following:

- **Charting and graphing service** Generates visual data representations including line, bar, pie, and other charts.
- **Full-text search service** Searches documents and databases using the Verity search engine.
- **Flash Remoting service** Provides a high performance protocol for exchanging data with Flash MX clients.

The ColdFusion MX Administrator

The 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.

You can manage the following options with the ColdFusion Administrator:

- ColdFusion data sources
- Debugging and logging output
- Server settings
- Application security

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

Using ColdFusion MX with Macromedia Flash MX

Macromedia Flash MX is designed to overcome the many limitations of HTML and solve the problem of providing efficient, interactive, user interfaces for Internet applications. ColdFusion MX is designed to provide a fast efficient environment for developing and providing data-driven Internet applications on your server. Using the following features, ColdFusion MX and Flash MX can work together in a seamless manner to provide complete interactive Internet applications:

- **ColdFusion MX native Flash connectivity** Lets Flash MX clients interact with ColdFusion MX in an efficient, secure, and reliable way. Flash MX includes ActionScript commands that connect to ColdFusion components (CFC) and ColdFusion pages. Flash clients communicate with ColdFusion applications using Action Message Format protocol over HTTP, which provides fast, lightweight, binary data transfer between the Flash client and ColdFusion.
- **Flash MX development application debugger** Lets you trace your application logic as it executes between Flash and ColdFusion.
- **ColdFusion MX Server-Side ActionScript** Lets Flash programmers familiar with ActionScript create ColdFusion services, such as SQL queries, for use by Flash clients.

Together, these features let developers build integrated applications that run on the Flash client and the ColdFusion scripting environment.

For more information about using Server-Side ActionScript in ColdFusion MX, see *Using Server-Side ActionScript in ColdFusion MX*. For more information on developing Flash applications in ColdFusion, see [Chapter 29, “Using the Flash Remoting Service” on page 673](#). For more information about using Flash MX, go to the following URL: <http://www.macromedia.com>.

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.

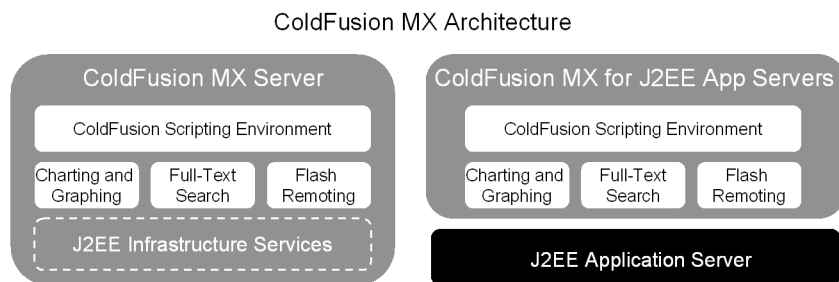
About ColdFusion and the J2EE platform

ColdFusion MX uses the J2EE infrastructure for many of its base services. By implementing the ColdFusion scripting environment on top of the J2EE platform, ColdFusion MX takes advantage of the power of the J2EE platform, but provides this power through the easy-to-use ColdFusion scripting environment.

ColdFusion MX consists of a family of products that differ in how they integrate with and use J2EE services. (Some ColdFusion editions might not be available at the time of the first ColdFusion MX release.)

- ColdFusion MX Server is a standalone servers that includes the entire infrastructure necessary to run ColdFusion applications, including an embedded Java™ server based on Macromedia JRun technology.
- ColdFusion MX for J2EE Application Servers consists of editions of ColdFusion MX that enable you to add ColdFusion MX capabilities to a J2EE server installation, including Macromedia JRun and other J2EE application servers.

The following figure shows how these editions are structured. Each edition supports the same ColdFusion scripting environment and includes the built-in application services, while the different editions enable developers to deploy ColdFusion MX in the configuration of their choice.



J2EE infrastructure services and J2EE application server

ColdFusion MX is implemented on the Java technology platform provided by a J2EE application server. It uses either an integrated J2EE infrastructure that uses Macromedia JRun technology, or an independent J2EE application server. The Java technology

platform provides much of the core functionality required by ColdFusion, including database connectivity, naming and directory services, and other runtime services.

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:

- Use custom JSP (Java Server Pages) tags from JSP tag libraries
- Interoperate with JSP pages
- Use Java servlets
- Use Java objects, including the J2EE Java API, JavaBeans, and Enterprise JavaBeans

For more information on using J2EE features in ColdFusion, see [Chapter 32](#), “Integrating J2EE and Java Elements in CFML Applications” on page 759.

ColdFusion features described in this book

ColdFusion provides a comprehensive set of features for developing and managing Internet applications. These features enhance the 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 book, and lists the chapters that describe them. This table is only a summary of major CFML features; this book also includes information about other features. Also, this table does not include features that are described in other books.

Feature	Description	Chapters
CFML language	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, 25
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 <code>cffunction</code> tag to create your own functions. These functions can incorporate all of the built-in ColdFusion tags and functions, plus other extensions.	9
Custom CFML tags	You can create custom ColdFusion tags using CFML. These tags can have bodies and can call other custom tags.	10
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.	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
ColdFusion application structure	ColdFusion supports many ways of building an application, and includes specific features, such as the <code>Application.cfm</code> 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 <code>cftry</code> and <code>cfcatch</code> .	14

Feature	Description	Chapters
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 <code>cflogin</code> 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 <code>cftrace</code> 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 directory information, and other data.	22
LDAP directory access and management	ColdFusion applications can access and manipulate data in LDAP (Lightweight Directory Access Protocol) 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 <code>cfform</code> tag to enrich your forms with sophisticated graphical controls, and perform input data validation.	26–27
Data graphing	You can use the <code>cfchart</code> tag to display your data graphically.	28
Macromedia Flash integration	You can use native Flash connectivity built into Macromedia ColdFusion MX to help build dynamic Flash user interfaces for ColdFusion applications.	29
XML document processing and creation	ColdFusion applications can create, use, and manipulate XML documents. ColdFusion also provides tools to use Web Distributed Data Exchange (WDDX), an XML dialect for transmitting structured data.	30

Feature	Description	Chapters
Web services	ColdFusion applications can use available SOAP-based web services, including Microsoft .NET services. ColdFusion applications can also use ColdFusion components to provide web services to other applications over the Internet.	31
Java and J2EE Integration	You can integrate J2EE elements, including JSP pages, JSP tag libraries, and Java objects, including Enterprise JavaBeans (EJBs), into your ColdFusion application.	32
COM and CORBA objects	The <code>cfobject</code> 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.	33
E-mail messages	You can add interactive e-mail features to your ColdFusion applications using the <code>cfmail</code> and <code>cfpop</code> tags.	34
HTTP and FTP	The <code>cfhttp</code> and <code>cfftp</code> tags provide simple methods of using HTTP (Hypertext Transfer Protocol) and FTP (File Transfer Protocol) communications in your application.	35
File and directory access	You can use the <code>cffile</code> , <code>cfdirectory</code> , and <code>cfcontent</code> tags to read, write, and manage files and directories on the server.	36

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:

Elements of CFML.....	15
Using ColdFusion Variables.....	33
Using Expressions and Pound Signs	65
Using Arrays and Structures	87
Extending ColdFusion Pages with CFML Scripting	115
Using Regular Expressions in Functions.....	133

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 book describe these topics in detail.

Contents

- Introduction 16
- Character case..... 16
- Tags 17
- Functions..... 19
- Expressions 21
- Constants 21
- Variables 22
- Data types 24
- ColdFusion components..... 25
- CFScript..... 26
- Flow control 27
- Comments..... 31
- Special characters 31
- Reserved words 32

Introduction

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 data types, comments, escape characters, and reserved words.

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

Character case

The ColdFusion Server 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.

If you follow these rules, you will prevent errors on application pages where you use both CFML and case-sensitive languages, such as JavaScript,

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. ColdFusion 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.

Built-in tags

Over 80 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 programming 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.

Currently, over 1,000 custom tags are available on the Macromedia developer's exchange (<http://www.coldfusion.com/Developer/Gallery/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 10, “Creating and Using Custom CFML Tags” on page 197](#). For information about, and comparisons among, ways to reuse ColdFusion code, including CFML custom tags, user-defined functions, and CFX tags, see [Chapter 8, “Reusing Code in ColdFusion Pages” on page 157](#).

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 243](#).

Functions

Functions typically manipulate data and return a result. CFML includes over 250 built-in functions. You can also use CFScript to 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 *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(), "mmm 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.

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 <http://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 167.

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)
```

The following sections briefly describe constants and variables. For detailed information on using variables, see [Chapter 3, “Using ColdFusion Variables” on page 33](#). For detailed information on expressions and operators, see [Chapter 4, “Using Expressions and Pound Signs” on page 65](#).

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 *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 24](#) 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 33](#).

Variable scopes

The following table briefly lists ColdFusion variable scopes:

Scope	Description
Variables (local)	The default scope for variables of any type that are created with the <code>cfset</code> and <code>cfparam</code> 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 <code>cfassociate</code> 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 <code>cfapplication</code> tag <code>name</code> 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 <code>cffunction</code> 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.

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: <ul style="list-style-type: none">• strings, such as "This is a test."• integers, such as 356• real numbers, such as -3.14159• Boolean values, True or False• date-time values, such as 3:00 PM July 12, 2001
Complex	A container for data. Complex variables generally represent more than one value. ColdFusion built-in complex data types are: <ul style="list-style-type: none">• 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 <code>cfobject</code> tag and other specialized tags.

For more information on ColdFusion data types, see [Chapter 3, “Using ColdFusion Variables”](#) on page 33.

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 providing 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 11, “Building and Using ColdFusion Components”](#) on page 217

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. While 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 115. For more information on user-defined functions, see [Chapter 9, “Writing and Calling User-Defined Functions”](#) on page 167.

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 115](#). 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 to 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,” in Chapter 10 and *CFML Reference*](#).

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 pound 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.

Note: You cannot embed comments inside a tag names 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")`.

Special characters

The double quotation marks ("), single quotation mark ('), and pound 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-quote (apostrophe) characters. Inside a single-quoted string, you do not escape double-quote characters.

The following example illustrates escaping special characters, including the use of mixed single and double quotes.

```
<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 pound 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 pound 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 [“Validating form field data types,” in Chapter 26.](#))

- _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.

CHAPTER 3

Using ColdFusion Variables

This chapter provides detailed information on ColdFusion variables and their use. 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 variables. It provides information on how variables can represent different types of data and how the data types get converted. It also discusses how variables exist in different scopes and provides an introduction to how the scopes are used. Finally, it provides additional information required to use variables correctly.

Contents

- [Creating variables](#) 34
- [Variable characteristics](#) 35
- [Data types](#) 35
- [Using periods in variable references](#)..... 45
- [Data type conversion](#) 49
- [About scopes](#)..... 55
- [Ensuring variable existence](#) 60
- [Validating data types](#)..... 62
- [Passing variables to custom tags and UDFs](#)..... 64

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 *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 60](#).

For more information on how to create variables, see [“Creating and using variables in scopes” on page 57](#).

Variable naming rules

Variable names must conform to Java naming rules. When naming ColdFusion variables and form fields, follow these guidelines:

- A variable name must begin with a letter, underscore, or Unicode currency symbol.
- The initial character can be followed by any number of letters, numbers, and underscore characters. Unicode currency symbols are also allowed.
- A variable name cannot contain spaces.
- A query result is a type of variable, so it cannot have the same name as another local variable in the current ColdFusion application page.
- 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.
- 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 some cases, you must prefix the scope. For more information, see [“About scopes” on page 55](#).
- 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 45](#)

Note: In some cases, when you use an existing variable name, you must put pound signs (#) around the name 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 the section [“Using pound signs,” in Chapter 4](#).

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. ColdFusion simple data types include numbers, strings, Booleans, and date-time variables. You can use simple data types directly in ColdFusion expressions.
- **Complex** A container for data. Complex variables 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, objects 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 33, “Integrating COM and CORBA Objects in CFML Applications” on page 785
Common Object Request Broker Architecture (CORBA)	Chapter 33, “Integrating COM and CORBA Objects in CFML Applications” on page 785
Java	Chapter 32, “Integrating J2EE and Java Elements in CFML Applications” on page 759

Object type	See
ColdFusion component	Chapter 11, “Building and Using ColdFusion Components” on page 217
Web service	Chapter 31, “Using Web Services” on page 729

Data type notes

Although ColdFusion variables do not have types, it is often convenient to refer to a variable’s 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. Form more information on form field data type validation, see [“Validating form field data types,” in Chapter 26](#). For more information on query parameter validation, see [“Using cfqueryparam,” in Chapter 20](#).

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 another simple data type:

- `isDate`
- `isNumeric`

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 until you use it as a simple value. At that time, ColdFusion converts the null to an empty string (“”).

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 code 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 xE^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. There is, however, a 64K limit on the size of text data that can be read from and written to a ColdFusion database or HTML text area. The ColdFusion 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 quotes and pound signs

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

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

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

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

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

This is a single quote: ' This is a double quote: "

Note: To insert a pound sign in a string, you must escape the pound sign, as in:
"This is a pound 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".

Booleans

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.

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 are accurate to the second. The following table lists valid date and time formats:

To specify	Use these formats
Date	October 30, 2001 Oct 30, 2001 Oct. 30, 2001 10/30/01 2001-10-30 10-30-2001

To specify	Use these formats
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, 2001 02:34:12 Oct 30, 2001 2:34a Oct. 30, 2001 2:34am 10/30/1 02:34am 2001-10-30 2am 10-30-2001 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, mmmm dd, yyyy")#</CFOUTPUT>
```

This code outputs a line like the following:

```
ven., juin 15, 2001
```

For more information on international functions, see *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.

Binary data type and Base64 encoding

Binary data 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 Base64 encoding before transmitting the file by e-mail.

Base64 format encodes the data in the lowest six bits of each byte. It ensures that binary data and non-ANSI character data can be transmitted by e-mail without corruption. The MIME specification defines the Base64 encoding method.

ColdFusion does not have a Base64 data type; it processes Base64 encoded data as string data.

ColdFusion provides the following functions that convert among string data, binary data, and Base64 encoded string data:

Function	Description
ToBase64	Converts string and binary data to Base64 encoded data.
ToBinary	Converts Base64 encoded data to binary data.
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.) It cannot convert binary data that includes bytes that are not printable characters.

The `ToString` function cannot convert Base64 encoded data directly to an unencoded string. Use the following procedure to convert Base64 encoded data that was originally a string back to a readable string:

- 1 Use the `ToBinary` function to convert the Base64 data into binary format.
- 2 Use the `ToString` function to convert the binary data to string.

For example, the following two lines print the same results:

```
<cfoutput>This is a test</cfoutput>
<cfoutput>#ToString(ToBinary(ToBase64("This is a test")))#</cfoutput>
```

Do not use binary data or Base64 data directly in ColdFusion expressions.

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 87](#).

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 code 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 87](#).

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 `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 code creates a new variable `myStructure2` that references 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 87](#).

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 queries.

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 code 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 code assigns the contents of the second row of the `Employee` column in the `myQuery` query to the variable `myVar`:

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

You cannot use array notation to refer to a row (of all columns) of a query.

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="CompanyInfo" 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; as the two reference formats produce different results.

Here are the rules for these references:

If you refer to `myQuery.Firstname`, ColdFusion automatically converts it to the first row in the column. For example, the following line prints out 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 code prints out 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 code displays the name "ben":

```
<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.

Note: For more information on scopes, see [“About scopes” on page 55](#). For more information on references to arrays and structures, see [Chapter 5, “Using Arrays and Structures” on page 87](#). For more information on references to XML document objects, see [Chapter 30, “Using XML and WDDX” on page 687](#).

Understanding variables and periods

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

Getting a variable

ColdFusion can correctly get variable values even if a simple 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, it 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, it checks whether a is a complex object,
- 4 If a or myVar.a is the name of a complex object, it 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, it checks whether a.b is the name of a simple variable and returns its value.
If myVar.a is not a complex object, it checks whether myVar.a.B is the name of a simple variable and returns its value.

This way, even if myVar.a.b is a simple variable name, 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 bracket.

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 or not 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 creates a structure named `myVar`, creates a structure named `a` in the structure `myVar`, creates a key named `b` in `myVar.a`, and gives it the value `"This is a test"`. If either `myVar` or `myVar.a` exist and is not a structure, ColdFusion generates an error.

In other words, ColdFusion uses the same rules as in the [Getting a variable](#) section 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. As a result 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 simple variables (including arrays) 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 [“Structure notation,” in Chapter 5](#). 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 55](#)
- 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>
```

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 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"

Value	As Boolean	As number	As date-time	As string
True	True	1	Error	"Yes"
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.
String	If "Yes" or "No", or if the string can be converted to a number, it is treated as listed above.	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 US date or time format, including the use of full or abbreviated month names, it is converted to the corresponding date-time value. Days of week or unusual punctuation result in 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 notes

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 Base64 encoding” on page 40](#).

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 5:34 PM on November 7, 2001, its output looked like this:

Using `cfoutput` to display the result of the `now` function:

```
{ts '2001-11-07 17:34:01'}
```

Now Add 0 to the result and display it:

```
37202.731956
```

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` tags 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 contents 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	Interpreted as
<code><cfif "1a" EQ "01:00"></code>	If 1:00am is 1:00am.
<code><cfif "1P" GT "2A"></code>	If 1:00pm is later than 2:00am.
<code><cfset age="4a"></code> <code><cfset age=age + 7></code>	Treat the variable <code>age</code> as 4:00 am, convert it to the date-time value 0.166666666667, and add 7 to make it 7.166666666667.
<code><cfif "0.0" is "0"></code>	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 (VM) 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 VM 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*.

The effect of quotes

To ensure that ColdFusion properly interprets string data, surround strings in single or double quotes. 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

```
True AND 2 * 3
```

Result value as string: “YES”

Explanation: 6 is converted to Boolean True because it is nonzero; True AND True is True.

Example 3

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

Result value as string: “Five is 5”

Explanation: 5 is converted to the string “5”.

Example 4

```
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 the source of the data, 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 on user-defined function scopes see [“Specifying the scope of a function,” in Chapter 9](#).

Scope types

The following table lists the types of ColdFusion scopes and describes their uses:

Scope	Description
Variables (local)	The default scope for variables of any type that are created with the <code>cfset</code> and <code>cfparam</code> 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 <code>form</code> tag, you must use <code>method="post"</code> .) For information on using the Form scope, see Chapter 26, “Retrieving and Formatting Data” on page 579 .
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 <code>?variablename = value[&variablename=value...]</code> ; for example <code>www.MyCompany.com/inputpage.cfm?productCode=A12CD1510&quantity=3</code> .
Attributes	Used only in custom tag pages. Contains the values passed by the calling page in the custom tag's attributes. For information on using the Attributes scope, see Chapter 10, “Creating and Using Custom CFML Tags” on page 197 .
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 information on using the Caller scope, see Chapter 10, “Creating and Using Custom CFML Tags” on page 197 .
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 <code>cfassociate</code> tag to return values to the calling tag's ThisTag scope. For more information on the ThisTag scope, see “Accessing tag instance data,” in Chapter 10 .

Scope	Description
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 <i>CFML Reference</i> .
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 information on using the Client scope, see Chapter 15, "Using Persistent Data and Locking" on page 315 .
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. You cannot use application variables on server clusters where more than one computer can process requests from a single session. For information on using the Session scope, see Chapter 15, "Using Persistent Data and Locking" on page 315 .
Application	Contains variables that are associated with one, named application on a server. The <code>cfapplication</code> tag <code>name</code> attribute specifies the application name. For information on using the Application scope, see Chapter 15, "Using Persistent Data and Locking" on page 315 .
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 information on using the Server scope, see Chapter 15, "Using Persistent Data and Locking" on page 315 .
Flash	Variables sent by a Macromedia Flash movie to ColdFusion and returned by ColdFusion to the movie. For more information on the Flash scope, see Chapter 29, "Using the Flash Remoting Service" on page 673 .
Arguments	Variables passed in a call to a user-defined function or ColdFusion component method. For more information see "About the Arguments scope," in Chapter 9 .

Scope	Description
This	Variables that are declared inside a ColdFusion component or in a <code>cffunction</code> tag that is not part of a ColdFusion component. These variables exist only while a function executes.
function local	Contains variables that are declared inside a user-defined function that you create using CFScript and exist only while a function executes. For information on using function local variables, see Chapter 9, “Writing and Calling User-Defined Functions” on page 167 .

Caution: To prevent data corruption, you lock code that uses Session, Application, or Server scope variables. For more information on using these scopes and locking access to code, see [Chapter 15, “Using Persistent Data and Locking” on page 315](#).

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 34](#).

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 <code>form</code> or <code>cfform</code> 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.

Scope prefix (type)	Prefix required to reference	Where available	Created by
ThisTag	Yes	On the custom tag page.	Specifying the prefix ThisTag when you create the variable in the tag or using the <code>cfassociate</code> 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.
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.
Cookie	No	For one client in one or more applications and pages, over multiple browser sessions.	A <code>cfcookie</code> 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 all code that uses application variables in <code>cflock</code> blocks.	Specifying the prefix Session when you create the variable.
Application	Yes	For multiple clients in one application over multiple browser sessions. Surround all code that uses application variables in <code>cflock</code> 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 <code>cflock</code> 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 <code>Flash.result</code> and <code>Flash.pagesize</code> 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.
This	Yes	Within the body of a user-defined function or ColdFusion component method that was created using the <code>cffunction</code> tag, only while the function executes.	Specifying the prefix This when you create the variable.
(function local, no prefix)	Prohibited	Within the body of a user-defined function that was created using <code>CFScript</code> , only while the function executes.	A <code>var</code> statement in the function body.

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 Arguments
- 2 Variables (local scope)
- 3 CGI
- 4 URL
- 5 Form
- 6 Cookie
- 7 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 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 to 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 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, Server.)

You can reference the variables in these 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 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 87](#).

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, 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:

- 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 also 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 [“Requiring users to enter values in form fields,” in Chapter 26](#).

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.

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 quotes. 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, use the interactive debugger in ColdFusion Studio or turn on debugging in the ColdFusion Administrator. The Administrator debugging information shows which variables are being passed to your application pages.

Variable existence notes

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 may 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, copy the element to a simple variable and use `IsDefined` to test whether the simple variable exists.

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 281](#).

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 `cfparam` 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 `SelectDepts` variable exists:

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

Validating data types

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 any variable.
- Use a form `input` tag with a `hidden` attribute to validate the contents of a form input field. For information on this technique, see [“Validating form field data types,” in Chapter 26](#).
- Use `cfform` controls that have validation attributes. For information on using `cfform` tags, see [Chapter 27, “Building Dynamic Forms” on page 607](#).
- Use the `cfqueryparam` tag in a SQL WHERE clause to validate query parameters. For information on this technique, see [“Using cfqueryparam,” in Chapter 20](#).

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

Using cfparam to validate the data type

The `cfparam` `type` attribute lets you validate the type of a parameter. You can specify that the parameter type must be any of the following values:

Type value	Meaning
any	any value
array	any array value
binary	any binary value
boolean	true, false, yes, or no

Type value	Meaning
date	any value in a valid date, time, or date-time format
numeric	any number
query	a query object
string	a text string or single character
struct	a structure
UUID	a Universally Unique Identifier (UUID) formatted as XXXXXXXX-XXXX-XXXX-XXXXXXXXXXXXXXXX where X stands for a hexadecimal digit (0-9 or A-F).
variableName	a valid variable name

For example, you can use the following code to validate the variable BirthDate:

```
<cfparam name="BirthDate" type="date">
```

If the variable is not in a valid date format, an error occurs and the page stops processing.

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 `cfoject` 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 put the variable name in pound signs. To pass a variable to a function, do *not* put the variable name in pound 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 `cfoject` 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 allows you to get a passed integer value.

CHAPTER 4

Using Expressions and Pound Signs

This chapter discusses how to use expressions in CFML. It discusses the elements of ColdFusion Expressions and how to create expressions. It also describes the correct use of pound 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 66
- Using pound signs 71
- Dynamic expressions and dynamic variables 74

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 an integer causes an error, and if you specify a real number ColdFusion ignores the fractional part; for example, 11 MOD 4 is 3.

Operator	Description
\	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.
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 is contained in the value on the right.

Operator	Description
DOES NOT CONTAIN	Opposite of CONTAINS. Return True if the value on the left is not contained in 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
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,”](#) in [Chapter 3](#).
- 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.

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(), "mmm 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 pound signs” on page 71](#).

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".

- Concatenates "we all need" with the string " more sleep!" to get "we all need more sleep!".
 - Passes the string "we all need more sleep!" to the `UCase` function.
 - Executes the `UCase` function on the string argument "we all need more sleep!" to get "WE ALL NEED MORE SLEEP!".
 - Assigns the string value "WE ALL NEED MORE SLEEP!" to the variable `myStringVar`.
- ColdFusion completes steps 1-3 before invoking the function.

Using pound signs

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

Is so, it replaces the text and surrounding pound 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 pound 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 pound signs:

- Use pound signs to distinguish variables or functions from plain text.
- Surround only a single variable or function in pound signs; for example, `#Variables.myVar#` or `#Left(myString, position)#`. (However, a function in pound signs can contain nested functions, such as `#Left(trim(myString), position)#`.)
- Do not put complex expressions, such as `1 + 2` in pound signs.
- Use pound signs *only* where necessary, because unneeded pound signs slow processing.

The following sections provide more details on how to use pound signs in CFML. For a description of using pound signs to create variable names, see [“Using pound signs to construct a variable name in assignments” on page 76](#)

Using pound signs in ColdFusion tag attribute values

You can put variables, functions, or expressions inside tag attributes by enclosing the variable or expression with pound 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 pound 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 pound 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 pound 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 pound signs, the text, rather than the value, appears in the output generated by the `cfoutput` statement.

Two expressions inside pound 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 pound sign as an escaped `#` character.

Using pound signs in strings

You can put variables or functions freely inside strings by enclosing each variable or expression with pound 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 pound 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 quotes around "Moon" and "Monkey" do *not* need to be escaped (as in ""Moon"" and ""Monkey""). This is because the text between the pound signs is treated as an expression; it is evaluated before its value is inserted inside the string.

Nested pound signs

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

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

In this example, pound 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 pound 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 pound 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 pound 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 pound signs around function arguments.

Using pound signs in expressions

Use pound signs in expressions only when necessary, because unneeded pound 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 pound 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 80](#).

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 87](#).

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 82](#).

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 pound 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 quotes 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 quotes 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,” in Chapter 5](#).) 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 82](#).

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 quotes in the argument and wraps the result in double quotes. 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 80 .
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 78 .
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 <code>cfif</code> tag in-line in HTML. For an example of using this function, see “Using the IIF function” on page 80 .
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 80 .

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 quotes) 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 following example can help you understand 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
<pre><cfset myVar2="myVar"> <cfset myVar="27/9"></pre>	Sets the two variables to the following strings: myVar 27/9
<pre><cfoutput> #myVar2#
 #myVar#
 #Evaluate("myVar2")#
</pre>	Displays the values assigned to the variables, myVar and 27/9 respectively.
<pre> #Evaluate("myVar")#
</pre>	Passes the string "myvar" (without the quotes) to the Evaluate function, which does the following: <ol style="list-style-type: none">1 Evaluates it as the variable myVar.2 Returns the value of the myVar variable, the string "myvar" (without the quotes).
<pre> #Evaluate(myVar2)#
</pre>	Evaluates the variable myVar2 as the string "myVar" and passes the string (without the quotes) to the Evaluate function. The rest of the processing is the same as in the previous line.
<pre> #Evaluate(myVar)#
 </cfoutput></pre>	Evaluates the variable myVar as the string "27/9" (without the quotes), and passes it to the Evaluate function, which does the following: <ol style="list-style-type: none">1 Evaluates the string as the expression 27/92 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](#),” in [Chapter 5](#).)

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 pound 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 pound signs to construct a variable name in assignments” on page 76](#).

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>


```

2 Save the page as `ShoppingCartForm.cfm`.

Reviewing the code

The following table describes the code:

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

Code	Description
<pre> <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> </pre>	<p>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_".</p> <p>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.</p> <p>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 quotes. (For more information on the cfinput tag passthrough attribute, see CFML Reference.) The check boxes are selected by default.</p>
<pre> <input type="submit" name="submit" value="Submit"> </form> </pre>	<p>Create the Submit button and end the form.</p>

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
<code><cfif isDefined("Form.submit")></code>	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.
<code><cfparam name="Form.itemID" default=""></code>	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).
<code><cfoutput></code> You have ordered the following items: <code>
</code> <code><cfloop index="i" list="#Form.itemID#"></code> Product Name: #Form["product_" & i]# Product Code: #Form["sku_" & i]# Quantity: #Form["qty_" & i]# <code></cfloop></code> <code></cfoutput></code> <code></cfif></code>	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

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 its 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](#) 88
- [Basic array techniques](#) 90
- [Populating arrays with data](#) 95
- [Array functions](#) 98
- [About structures](#) 99
- [Creating and using structures](#) 102
- [Structure example](#) 109
- [Structure functions](#) 113

About arrays

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

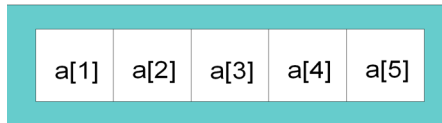
In ColdFusion, you typically use arrays to temporarily store data. For example, if your site lets users order goods online, you can store their shopping cart contents in an array. This lets you make changes easily without committing the information, which the user can change before completing the transaction,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

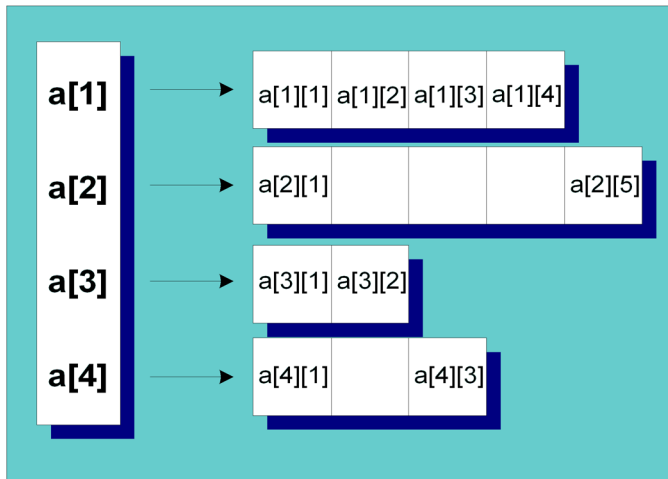
ColdFusion arrays differ from traditional arrays, because they are dynamic. For example, in a conventional array, array size is constant and symmetrical, whereas in a ColdFusion array, you can have rows of differing lengths based on the data that has been added or removed.

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:

a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][1]	a[2][2]	a[2][3]	a[2][4]
a[3][1]	a[3][2]	a[3][3]	a[3][4]
a[4][1]	a[4][2]	a[4][3]	a[4][4]

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. To test whether data exists at an array index, copy the array element to a simple variable and use the `IsDefined` function to test the existence of the copy.

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 256) 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
<pre><cfset myarray=ArrayNew(1)> <cfset myotherarray=ArrayNew(2)> <cfset biggerarray=ArrayNew(3)></pre>	Create three empty arrays, a 1D array, a 2D array, and a 3D array.
<pre><cfset biggerarray[1][1][1]=myarray> <cfset biggerarray[1][1][1][10]=3></pre>	Make element <code>[1][1][1]</code> of the 3D <code>biggerarray</code> array be a copy of the 1D array. Assign 3 to the <code>[1][1][1][10]</code> element of the resulting array. The <code>biggerarray</code> array is now asymmetric. For example, it does not have a <code>[1][1][2][1]</code> element.
<pre><cfset biggerarray[2][1][1]= myotherarray> <cfset biggerarray[2][1][1][4][2]= "reality"></pre>	Make element <code>[2][1][1]</code> of the 3D array be the 2D array and assign the <code>[2][1][1][4][2]</code> element the value "reality". The <code>biggerarray</code> array is now even more asymmetric.

Code	Description
<pre><cfset biggestarray=ArrayNew(3)> <cfset biggestarray[3][1][1] =biggerarray> <cfset biggestarray[3][1][1][2][3][1] ="This is complex"></pre>	<p>Create a second 3D array. Make the [3][1][1] element of this array be a copy of the biggerarray array, and assign element [3][1][1][2][3][1].</p> <p>The resulting array is very complex and asymmetric.</p>
<pre><cfset myarray[3]="Can you see me"></pre>	<p>Assign a value to element [3] of myarray.</p>
<pre><fdump var=#biggestarray#>
 <fdump var=#myarray#></pre>	<p>Use cfdump to view the structure of biggestarray and myarray.</p> <p>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.</p>

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
<code>ArrayAppend</code>	Creates a new array element at the end of the array.

Function	Description
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="cfsnippets">
    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. Which notation you use depends on your requirements:

Notation	Description
Object.property	<p>You can refer to a property, prop, of an object, obj, as obj.prop. This notation is useful for simple assignments, as in this example:</p> <pre>depts.John="Sales"</pre> <p>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.</p>
Associative arrays	<p>If you do not know the key name is 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,</p> <pre>depts["John"]="Sales"</pre> <pre>depts[employeeName]="Sales"</pre> <p>You can use a variable (such as employeeName) as an associative array index. Therefore, you must enclose any literal key names in quotes.</p> <p>For information on using associative array references containing variables, see “Dynamically constructing structure references,” in Chapter 4.</p>

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
<pre><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></pre>	Create a structure with three entries: a string, an array, and an embedded structure.
<pre><cfdump var=#myStruct#>
</pre>	Display the complete structure.
<pre><cfset key1Var="key1"> <cfset key2Var="key2"> <cfset key3Var="key3"> <cfset var2="2"></pre>	Create variables containing the names of the myStruct keys and the number 2.
<pre><cfoutput> Value of the first key
 #myStruct.key1#
 #myStruct["key1"]#
 #myStruct[key1Var]#

</pre>	Output the value of the structure's key1 (string) entry using the following notation: <ul style="list-style-type: none">• object.property notation• associative array notation with a constant• associative array notation with a variable
<pre>Value of the second entry in the key2 array
 #myStruct.key2[2]#
 #myStruct["key2"][2]#
 #myStruct[key2Var][2]#
 #myStruct[key2Var][var2]#

</pre>	Output the value of the second entry in the structure's key2 array using the following notation: <ul style="list-style-type: none">• 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
<pre>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></pre>	Output the value of second entry in the structure's key3 embedded structure using the following notation: <ul style="list-style-type: none">• 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">
```

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 quotes. 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 return 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	<p>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.</p> <p>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</p>
StructCopy function	<p>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.</p> <p>The Duplicate replaces this function for most, if not all, purposes.</p>
Variable assignment	<p>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.</p> <p>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.</p>

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 example

Structures are particularly useful for grouping together a set of variables under a single name. The example in this section uses structures 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 10, “Creating and Using Custom CFML Tags”](#) on page 197.

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 firstaname 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:&nbsp;
<input name="firstname" type="text" hspace="30" maxlength="30"><br>
Last Name:&nbsp;
<input name="lastname" type="text" hspace="30" maxlength="30"><br>
EMail:&nbsp;
<input name="email" type="text" hspace="30" maxlength="30"><br>
Phone:&nbsp;
<input name="phone" type="text" hspace="20" maxlength="20"><br>
Department:&nbsp;
<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
<pre> <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=""> </pre>	Set default values of all form fields so that they exist the first time this page is displayed and can be tested.
<pre> <cfif #form.firstname# eq ""> Please fill out the form.
 </pre>	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.
<pre> <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> </pre>	<p>If Form.firstname contains text, the user submitted the form.</p> <p>Use CFScript to create a new structure named employee and fill it with the form field data.</p> <p>Then display the contents of the structure</p>

Code	Description
<pre>cf_addemployee empinfo="#duplicate(employee)#" </cfif></pre>	<p>Call the cf_addemployee custom tag and pass it a copy of the employee structure in the empinfo attribute.</p> <p>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.</p>
<pre><form action="newemployee.cfm" method="Post"> First Name:&nbsp; <input name="firstname" type="text" hspace="30" maxlength="30">
 Last Name:&nbsp; <input name="lastname" type="text" hspace="30" maxlength="30">
 EMail:&nbsp; <input name="email" type="text" hspace="30" maxlength="30">
 Phone:&nbsp; <input name="phone" type="text" hspace="20" maxlength="20">
 <p>Department:&nbsp; <input name="department" type="text" hspace="30" maxlength="30">

 <input type="Submit" value="OK"> </form></pre>	<p>The data form. When the user clicks Submit, the form posts the data to this ColdFusion page.</p>

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="cfsnippets">
    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
<pre> <cfif StructIsEmpty(Attributes.empinfo)> <cfoutput> Error. No employee data was passed. </cfoutput> <cfexit method="ExitTag"> </pre>	<p>If the custom tag was called without an <code>empinfo</code> attribute, display an error message and exit the tag.</p>
<pre> <cfelse> <cfquery name="AddEmployee" datasource= "cfsnippets"> 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> </pre>	<p>Add the employee data passed in the <code>empinfo</code> structure to the <code>Employees</code> table of the <code>cfsnippets</code> database.</p> <p>Use direct references to the structure entries, not <code>structfind</code> functions.</p> <p>If the database does not support automatic generation of the <code>Emp_ID</code> key, you must add an <code>Emp_ID</code> entry to the form and add it to the query.</p>
<pre> <cfoutput> <hr>Employee Add Complete </cfoutput> </pre>	<p>Display a completion message. This code does not have to be inside the <code>cfelse</code> block because the <code>cfexit</code> tag prevents it from being run if the <code>empinfo</code> structure is empty.</p>

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 <code>java.util.Map</code> 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 <code>Duplicate</code> function has replaced this function for most purposes.
StructCount	Returns the number of keys in the specified structure.
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 <code>IsDefined</code> 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.

Function	Description
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

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](#) 116
- [The CFScript language](#) 118
- [Using CFScript statements](#) 122
- [Handling exceptions](#) 129
- [CFScript example](#) 130

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 120](#).

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>
  </cfif>
  Adding #Form.firstname# #Form.lastname#<br>
</cfif>
<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

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 Pound Signs” on page 65](#).

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	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 `var` and `return`. For information on these statements, see [“Defining functions in CFScript,” in Chapter 9.](#)

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 > 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.

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

Although you cannot use ColdFusion tags in CFScript, CFScript and ColdFusion functions provide equivalents to several commonly-used CFML tags. The following table lists ColdFusion tags with equivalent functions or CFScript statements:

Tag	CFScript equivalent
<code>cfset</code>	Direct assignment, such as <code>Myvar=1;</code>
<code>cfoutput</code>	<code>WriteOutput</code> function
<code>cfif</code> , <code>cfelseif</code> , <code>cfelse</code>	<code>if</code> and <code>else</code> statements
<code>cfswitch</code> , <code>cfcase</code> , <code>cfdefaultcase</code>	<code>switch</code> , <code>case</code> , and <code>default</code> statements
Indexed <code>cfloop</code>	<code>for</code> loops
Conditional <code>cfloop</code>	<code>while</code> loops and <code>do while</code> loops

Tag	CFScript equivalent
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.
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

Using CFScript statements

The following sections describe how to use these CFScript statements:

- Assignment statements and functions
- Conditional processing statements
- 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 have 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);
```

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, you cannot use the CFScript for-in loops to loop over a query, list, or object.

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 displays 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 281.

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:<br>");
  for (j=1;j lte ArrayLen(applicants.accepted);j=j+1)
  {
    WriteOutput(applicants.accepted[j] & "<br>");
  }
  WriteOutput("<br>");

  //sort and display rejected applicants with reaons information

  ArraySort(applicants.rejected,"text","asc");
  WriteOutput("The following applicants were rejected:<br>");
  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
<pre> <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"; </pre>	<p>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.</p> <p>Creates a structure and assign each array to an element of the structure.</p> <p>Creates a structure with rejection codes for rejected applicants. The <code>rejectCode</code> 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.</p>
<pre> ArraySort(applicants.accepted,"text","asc"); WriteOutput("The following applicants were accepted:<hr>"); for (j=1;j <= ArrayLen(applicants.accepted);j=j+1) { WriteOutput(applicants.accepted[j] & "
"); } WriteOutput("
"); </pre>	<p>Sorts the accepted applicants alphabetically.</p> <p>Displays a heading.</p> <p>Loops through the accepted applicants and writes their names. Braces enhance clarity, although they are not needed for a single statement loop.</p> <p>Writes an additional line break at the end of the list of accepted applicants.</p>

Code	Description
<pre>ArraySort(applicants.rejected,"text","asc"); WriteOutput("The following applicants were rejected:<hr>");</pre>	<p>Sorts rejectedApplicants array alphabetically and writes a heading.</p>
<pre>for (j=1;j <= ArrayLen(applicants.rejected);j=j+1) { applicant=applicants.rejected[j]; WriteOutput(applicant & "
");</pre>	<p>Loops through the rejected applicants.</p> <p>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.</p> <p>Writes the applicant name.</p>
<pre>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</pre>	<p>Checks the rejectCode structure for a rejection code for the applicant.</p> <p>If a code exists, enters a switch statement that examines the rejection code value.</p> <p>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.</p> <p>Comments at the end of blocks help clarify the control flow.</p>
<pre>else { WriteOutput("Reject reason was not defined.
"); }</pre>	<p>If there is no entry for the applicant in the rejectCode structure, displays a message indicating that the reason was not defined.</p>
<pre> WriteOutput("
"); } //end for </cfscript></pre>	<p>Displays a blank line after each rejected applicant.</p> <p>Ends the for loop that handles each rejected applicant.</p> <p>Ends the CFScript.</p>

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 `cfinput` 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”](#) on page 607.

Contents

- [About regular expressions](#) 134
- [Regular expression syntax](#) 136
- [Using backreferences](#)..... 144
- [Returning matched subexpressions](#) 147
- [Regular expression examples](#) 152
- [Types of regular expression technologies](#) 154

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 147](#).

`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 152](#).

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") - 7H7e71717o
```

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
```

You must be aware of two other considerations when using special characters:

- If you want 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.
- If you want to include] (closing square bracket) in the square brackets of a character set it must be the first character. Otherwise, it does not work even if you use \]. The following example shows this:

```
<!-- Want to replace closing square bracket and all a's with * -->  
<cfset strSearch = "[Test message]">  
<!-- Next line does not work since ] is not the FIRST character  
within [] -->  
<cfset re = "[a\]">  
<cfoutput>REReplace(#strSearch#,#re#,"*","ALL") -  
#REReplace(strSearch,re,"*","ALL")#<br>  
Neither '[' nor 'a' was replaced because we searched for 'a'  
followed by ']'<br>  
</cfoutput>  
<!-- Next line works since ] is the FIRST character within [] -->  
<cfset re = "[]a]">  
<cfoutput>REReplace(#strSearch#,#re#,"*","ALL") -  
#REReplace(strSearch,re,"*","ALL")#<br>  
Both 'a' and ']' were Replaced with *<br></cfoutput>
```

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 IndexOf0ccurrence=REFind("a{2,4}", "hahaaahaaahaaahhh")>  
<!-- The value of IndexOf0ccurrence 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
\	<p>A backslash followed by any special character matches the literal character itself, that is, the backslash escapes the special character.</p> <p>For example, "\+" matches the plus sign, and "\\\" matches a backslash.</p>
.	<p>A period matches any character, including newline.</p> <p>To match any character except a newline, use <code>[^#chr(13)##chr(10)#]</code>, which excludes the ASCII carriage return and line feed codes. The corresponding escape codes are <code>\r</code> and <code>\n</code>.</p>
[]	<p>A one-character character set that matches any of the characters in that set.</p> <p>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.</p> <p>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.</p> <p>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.</p>
^	<p>If the caret is at the beginning of a regular expression, the matched string must be at the beginning of the string being searched.</p> <p>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."</p>
\$	<p>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.</p> <p>For example, the regular expression "ColdFusion\$" matches the string "I like ColdFusion" but not the string "ColdFusion is fun."</p>
?	<p>A character set or subexpression followed by a question mark matches zero or one occurrences of the character set or subexpression.</p> <p>For example, xy?z matches either "xyz" or "xz".</p>
	<p>The OR character allows a choice between two regular expressions.</p> <p>For example, jell(y ies) matches either "jelly" or "jellies".</p>
+	<p>A character set or subexpression followed by a plus sign matches one or more occurrences of the character set or subexpression.</p> <p>For example, [a-z]+ matches one or more lowercase characters.</p>
*	<p>A character set or subexpression followed by an asterisk matches zero or more occurrences of the character set or subexpression.</p> <p>For example, [a-z]* matches zero or more lowercase characters.</p>
()	<p>Parentheses group parts of a regular expression into subexpressions that you can treat as a single unit.</p> <p>For example, (ha)+ matches one or more instances of "ha".</p>

Special Character	Description
-------------------	-------------

(?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 141](#).

(?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
(?=...)	<p>If at the beginning of a regular expression, it specifies to use positive lookahead when searching for the regular expression.</p> <p>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.</p> <p>For example, consider the expression to extract the protocol from a URL:</p> <pre><cfset regex = "http(?:=://)"> <cfset string = "http://"> <cfset result = reFind(regex, string, 1, "yes")> mid(string, result.pos[1], result.len[1])</pre> <p>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 "://".</p> <p>Lookahead parentheses do not capture text, so backreference numbering will skip over these groups. For more information on backreferencing, see "Using backreferences" on page 144.</p>
(?!...)	<p>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.</p> <p>Lookahead parentheses do not capture text, so backreference numbering will skip over these groups. For more information on backreferencing, see "Using backreferences" on page 144.</p>
(?:...)	<p>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.</p>

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 supported in ColdFusion:

Escape Sequence	Description
<code>\b</code>	<p>Specifies a boundary defined by a transition from an alphanumeric character to a nonalphanumeric character, or from a nonalphanumeric character to an alphanumeric character.</p> <p>For example, the string " Big" contains boundary defined by the space (nonalphanumeric character) and the "B" (alphanumeric character).</p> <p>The following example uses the <code>\b</code> 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".</p> <pre>reFindNoCase("\bBig\b", "Don't be ambiguous about Big.") <!-- The value of IndexOfOccurrence is 26 --></pre> <p>When used inside of a character set (e.g. <code>[\b]</code>), it specifies a backspace</p>
<code>\B</code>	<p>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 <code>\b</code>.</p>
<code>\A</code>	<p>Specifies a beginning of string anchor, much like the <code>^</code> special character. However, unlike <code>^</code>, you cannot combine <code>\A</code> with <code>(?m)</code> to specify the start of newlines in the search string.</p>
<code>\Z</code>	<p>Specifies an end of string anchor, much like the <code>\$</code> special character. However, unlike <code>\$</code>, you cannot combine <code>\Z</code> with <code>(?m)</code> to specify the end of newlines in the search string.</p>
<code>\n</code>	Newline character
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\f</code>	Form feed
<code>\d</code>	Any digit, similar to <code>[0-9]</code>
<code>\D</code>	Any nondigit character, similar to <code>[^0-9]</code>
<code>\w</code>	Any alphanumeric character, similar to <code>[:alnum:]</code>
<code>\W</code>	Any nonalphanumeric character, similar to <code>[^:alnum:]</code>
<code>\s</code>	Any whitespace character including tab, space, newline, carriage return, and form feed. Similar to <code>[\t\n\r\f]</code> .
<code>\S</code>	Any nonwhitespace character, similar to <code>[^\t\n\r\f]</code>
<code>\xdd</code>	A hexadecimal representation of character, where <code>d</code> is a hexadecimal digit
<code>\ddd</code>	An octal representation of a character, where <code>d</code> is an octal digit, in the form <code>\000</code> to <code>\377</code>

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:

Character class	Matches
:alpha:	Matches any letter. Same as <code>[A-Za-z]</code> .
:upper:	Matches any uppercase letter, including accented uppercase characters.
:lower:	Matches any lowercase letter, including accented lowercase characters.
:digit:	Matches any digit. Same as <code>[0-9]</code> and <code>\d</code> .
:alnum:	Matches any alphanumeric character. Same as <code>[A-Za-z0-9]</code> or <code>\w</code> .
:xdigit:	Matches any hexadecimal digit. Same as <code>[0-9A-Fa-f]</code> .
:blank:	Matches space or a tab.
:space:	Matches space, tab, new line, line feed, or carriage return. Same as <code>\s</code> .
:print:	Matches any printable character.
:punct:	Matches any punctuation character, that is, one of <code>! ' # \$ % & ` () * + , - . / : ; < = > ? @ [/] ^ _ { } -</code>
:graph:	Matches any of the characters defined as a printable character except those defined as part of the space character class.
:cntrl:	Matches any character not part of the character classes <code>[[:upper:]]</code> , <code>[[:lower:]]</code> , <code>[[:alpha:]]</code> , <code>[[:digit:]]</code> , <code>[[:punct:]]</code> , <code>[[:graph:]]</code> , <code>[[:print:]]</code> , or <code>[[:xdigit:]]</code> .

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 "pairs" 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 144](#).

```
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:



LEN	15
POS	15

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:



LEN	15
POS	19

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:

LEN	1	5
	2	2
POS	1	7
	2	7

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:

LEN	1	16
	2	6
	3	5
POS	1	1
	2	1
	3	12

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 "onetwo". 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:

POS	1	1
LEN	1	21

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:

POS	1 1
	2 4
LEN	1 10
	2 3

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
<code>[\?&]value=</code>	A URL parameter value in a URL.
<code>[A-Z]:(\\[A-Z0-9_]+)</code>	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.
<code>[A-Za-z][A-Za-z0-9_]*</code>	A ColdFusion variable with no qualifier.
<code>([A-Za-z][A-Za-z0-9_]*)(\\. [A-Za-z][A-Za-z0-9_]*)?</code>	A ColdFusion variable with no more than one qualifier; for example, <code>Form.VarName</code> , but not <code>Form.Image.VarName</code> .
<code>(\\+ -)?[1-9][0-9]*</code>	An integer that does not begin with a zero and has an optional sign.
<code>(\\+ -)?[1-9][0-9]*(\\. [0-9]*)?</code>	A real number.
<code>(\\+ -)?[1-9]\\.[0-9]*E(\\+ -)?[0-9]+</code>	A real number in engineering notation.
<code>a{2,4}</code>	Two to four occurrences of “a”: <code>aa</code> , <code>aaa</code> , <code>aaaa</code> .
<code>(ba){3,}</code>	At least three “ba” pairs: <code>bababa</code> , <code>babababa</code> , and so on.

Regular expressions in CFML

The following examples of CFML show some common uses of regular expression functions:

Expression	Returns
<code>REReplace (CGI.Query_String, "CFID=[0-9]+[&]*", "")</code>	The query string with parameter CFID and its numeric value stripped out.
<code>REReplace("I Love Jellies", "[[:lower:]]", "x", "ALL")</code>	<code>I Lxxx Jxxxxxx</code>
<code>REReplaceNoCase("cabaret", "[A-Z]", "G", "ALL")</code>	<code>GGGGGGG</code>
<code>REReplace (Report, "\\\$[0-9,]*\\. [0-9]*", "\$***.***)", "")</code>	The string value of the variable Report with all positive numbers in the dollar format changed to <code>"\$***.***"</code> .
<code>REFind ("[Uu]\\.[?][Ss]\\.[?][Aa]\\.", Report)</code>	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.

Expression	Returns
<code>REFindNoCase("a+c", "ABCAACCDD")</code>	4
<code>REReplace("There is is coffee in the the kitchen", "([A-Za-z]+)[]+\1", "**", "ALL")</code>	There * coffee in * kitchen
<code>REReplace(report, "<[>]*>", "", "All")</code>	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 `\uU` 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 <http://www.oreilly.com>.

PART II

Reusing CFML Code

This part describes techniques for reusing code in ColdFusion pages. These techniques let you write your code once and use it, without copying it, in many places. These techniques include the `cfinclude` tag, user-defined functions, custom tags, ColdFusion components, and ColdFusion Extension (CFX) tags.

The following chapters are included:

Reusing Code in ColdFusion Pages.....	157
Writing and Calling User-Defined Functions	167
Creating and Using Custom CFML Tags.....	197
Building and Using ColdFusion Components	217
Building Custom CFXAPI Tags	243

CHAPTER 8

Reusing Code in ColdFusion Pages

This chapter describes techniques for reusing code in ColdFusion pages. These techniques let you write your code once and use it, without copying it, in many places. This chapter describes the techniques and their features, and provides advice on selecting among the techniques.

Contents

- [About reusable CFML elements..... 158](#)
- [Including pages with the cfinclude tag 158](#)
- [Calling user-defined functions 161](#)
- [Using custom CFML tags..... 162](#)
- [Using CFX tags 164](#)
- [Using ColdFusion components..... 165](#)
- [Selecting among ColdFusion code reuse methods 166](#)

About reusable CFML elements

ColdFusion provides you with several types of reusable elements, sections of code that you can create once and use multiple times in an application. Many of these elements also let you extend the built-in capabilities of ColdFusion. ColdFusion provides the following reusable CFML elements:

- ColdFusion pages you include using the `cfinclude` tag
- User-defined functions (UDFs)
- Custom CFML tags
- CFX (ColdFusion Extension) tags
- ColdFusion components

The following sections describe the features of each of these elements and provide guidelines for determining which of these tools to use in your application. Other chapters describe the tools in detail. The last section in this chapter includes a table that helps you choose among these techniques for different purposes.

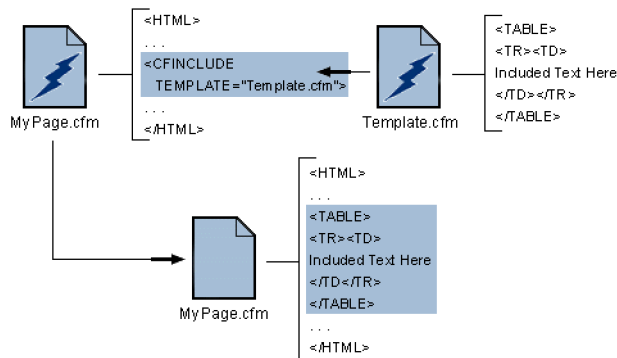
ColdFusion can also use elements developed using other technologies, including the following:

- JSP tags from JSP tag libraries
For information on using JSP tags, see [Chapter 32, “Integrating J2EE and Java Elements in CFML Applications” on page 759](#).
- Java objects, including objects in the Java runtime environment and JavaBeans
For information on using Java objects, see [Chapter 32, “Integrating J2EE and Java Elements in CFML Applications” on page 759](#).
- Microsoft COM (Component Object Model) objects
For information on using COM objects, see [Chapter 33, “Integrating COM and CORBA Objects in CFML Applications” on page 785](#).
- CORBA (Common Object Request Broker Architecture) objects
For information on using CORBA objects, see [Chapter 33, “Integrating COM and CORBA Objects in CFML Applications” on page 785](#).
- Web services
For information on using web services, see [Chapter 31, “Using Web Services” on page 729](#)

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 searches for the included file in directories that you specify on the Mappings page of the ColdFusion 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:

```

<br>
```

(For this code 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

Calling 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 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 use UDFs in your application pages as you use standard ColdFusion functions. You can create a function for an algorithm or procedure that you use frequently, and then use the function wherever you need the procedure.

As with custom tags, you can easily distribute UDFs to others. For example, the Common Function Library Project at <http://www.cflib.org> is an open-source collection of CFML user-defined functions.

Calling UDFs

To call a UDF, use it as you would a ColdFusion built-in function. For example, the following line calls the function `MyFuncnt` and passes it two arguments:

```
<cfset returnValue=MyFuncnt(Arg1, Arg2)>
```

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

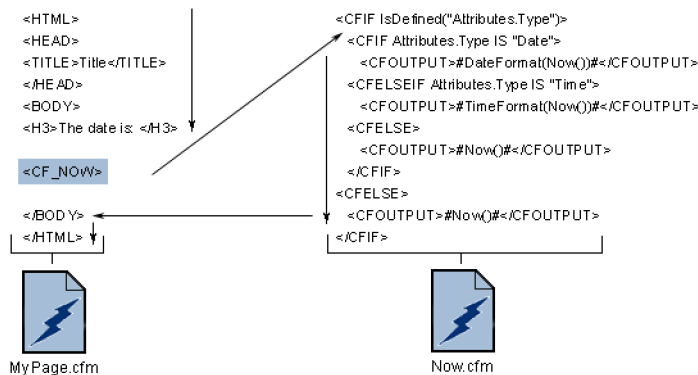
For more information on user-defined functions, see [Chapter 9, “Writing and Calling User-Defined Functions”](#) on page 167.

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.
- Can do specific processing when ColdFusion encounters the beginning tag.
- Can 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 10, “Creating and Using Custom CFML Tags” on page 197](#).

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 10, “Creating and Using Custom CFML Tags” on page 197](#).

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 `cfoject` tag. CFX tags, however, provide some built-in features that the `cfoject` 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 `cfoject` 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 243](#).

Using ColdFusion components

Unlike other ColdFusion reusable elements, ColdFusion components encapsulate multiple, related, functions. A **ColdFusion component** is essentially a set of related UDFs 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 Macromedia Flash clients can call directly.
- They have several features that are familiar to object-oriented programmers including data hiding, inheritance, packages, and introspection.

Creating and using ColdFusion components

Creating and using a component is more complex than creating and using a user-defined function (UDF). For example, you specify a component and one or more functions. You can invoke ColdFusion components in many ways, including using the `cfinvoke` and `cfobject` tags. You can also use forms, URLs, and the Flash client-side `ActionScript`.

To invoke a component method with a `cfinvoke` tag, use code such as the following:

```
<cfinvoke component="componentName" method="methodName"
    returnVariable="variableName" argumentCollection="argumentStruct">
```

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 11, “Building and Using ColdFusion Components” on page 217](#).

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 164](#).

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.....](#) 168
- [Calling user-defined functions.....](#) 169
- [Creating user-defined functions.....](#) 169
- [Calling functions and using variables.....](#) 180
- [A User-defined function example.....](#) 182
- [Using UDFs effectively.....](#) 184

About user-defined functions

You can create **user-defined functions**, or UDFs (also known as custom functions), and use them in your application pages as you do standard ColdFusion functions. This lets you create a function for an algorithm or procedure that you use frequently, and 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 pound (#) 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, custom tags, and ColdFusion components, see [Chapter 8, “Reusing Code in ColdFusion Pages” on page 157](#).

Note: The Common Function Library Project at <http://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. Typically you define the function on your ColdFusion page or a page that you include. You can also define the function on one page and put it in a scope that is shared with the page that calls it. (For more information on UDF scoping, see [“Specifying the scope of a function” on page 184](#).) You can also put commonly used functions on a single ColdFusion page and include it in your Application.cfm page.

Calling user-defined functions

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. For more information on calling functions with and without argument names, see [“Calling functions and using variables” on page 180](#).

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 182](#)). 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>
```

Creating user-defined functions

You can use tags or CFScript to create a UDF. Each technique has advantages and disadvantages.

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 174](#).

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 177](#).

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.
- 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 define a function in the following places:

- 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.
- On the `Application.cfm` page.

For recommendations on selecting where you define functions, see the sections [“Using Application.cfm and function include files” on page 184](#) and [“Specifying the scope of a function” on page 184](#).

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 176](#). 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 179](#).

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. To test whether an array index contains data, copy the array element to a simple variable and use the IsDefined function to test the existence of the copy.

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 an index number as the key to reference the argument in the structure. 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`.

Note: The `IsDefined` function always returns false when you reference an unnamed optional arguments using structure notation. For example, `IsDefined(Arguments.3)` for the function described in the preceding paragraph always returns false.

- 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.

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.

For more information on using function-only variables, see [“Using function-only variables” on page 181](#).

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>
```

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 115](#).

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
<i>functionName</i>	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.
<i>argName1...</i>	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 consist of one or more valid CFScript statements. The body must be in curly braces, even if it is a single statement.

The following two statements are allowed only in function definitions:

Statement	Description
var <i>variableName</i> = <i>expression</i> ;	<p>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.</p> <p>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.</p> <p>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.</p>
return <i>expression</i> ;	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.

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 171](#).

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 11, “Building and Using ColdFusion Components” on page 217](#). 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
    .
    .
    <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 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	<p>(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.</p> <p>If you use this attribute, the function executes only if the current user is logged in using the <code>cfloginuser</code> 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 347.</p>
output	<p>(Optional) Determines how ColdFusion processes displayable output in the function body.</p> <p>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 <code>cfoutput</code> tags in the function definition body are displayed each time the function executes.</p> <p>If you specify True or "yes", the body of the function is processed as if it were in a <code>cfoutput</code> tag. ColdFusion displays variable values and expression results if you surround the variables and expressions with pound signs.</p> <p>If you specify False or "no" the function is processed as if it were in a <code>cfsilent</code> tag. The function does not display any output. The code that calls the function is responsible for displaying any function results.</p>

You must use `cfargument` tags for required function arguments and named optional arguments. All `cfargument` tags must precede any other CFML code in `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	<p>(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.</p> <p>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 <code>type</code> attribute is numeric, you cannot call the function with an array.</p>
required	<p>(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.</p> <p>Because you do not identify arguments when you call a function, all <code>cfargument</code> tags that specify required arguments must precede any <code>cfargument</code> tags that specify optional arguments in the <code>cffunction</code> definition.</p>
default	<p>(Optional) The default value for an optional argument if no argument value is passed.</p> <p>If you specify this attribute, an error occurs if you specify this attribute and set the <code>required</code> attribute to True.</p>

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 172](#).

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.

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 `CompanyInfo` Employee table:

```
<cffunction name="getDeptID" >
  <cfargument name="empID" required="true" type="numeric">
  <cfquery dataSource="CompanyInfo" name="deptID">
    SELECT Dept_ID
    FROM Employee
    WHERE Emp_ID = #empID#
  </cfquery>
  <cfreturn deptID.Dept_ID>
</cffunction>
```

Note: The `cfquery` tag automatically puts the query result in the Variables scope, so you cannot limit its result to the This scope.

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 171](#).

Calling functions and using variables

You can call a function anywhere that you can use an expression, including in pound 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 call user-defined functions the same way you call any built-in ColdFusion functions.

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 189](#).

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 186.](#)
- 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 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.

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.)

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
<pre>function TotalInterest(principal, annualPercent, months) {</pre>	Starts the <code>TotalInterest</code> function definition. Requires three variables: the principal amount, the annual percentage rate, and the loan duration in months.
<pre> Var years = 0; Var interestRate = 0; Var totalInterest = 0;</pre>	Declares intermediate variables used in the function and initializes them to 0. All <code>var</code> statements must precede the rest of the function code.

Code	Description
<pre>principal = trim(principal); principal = REReplace(principal,"[\\$,]","", "ALL"); annualPercent = Replace(annualPercent,"%","", "ALL"); interestRate = annualPercent / 100; years = months / 12;</pre>	<p>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.</p> <p>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.</p> <p>Converts the loan from months to years.</p>
<pre>totalInterest = principal*(((1+ interestRate)^years)-1); Return DollarFormat(totalInterest); }</pre>	<p>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.</p> <p>Ends the function definition.</p>

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 years = 0>
    <cfset interestRate = 0>
    <cfset 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 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. Unlike with functions defined in <code>Application.cfm</code> or included from other ColdFusion pages, all pages use the same in-memory copy of the function. Using an Application scope function can save memory and the processing required to define a function multiple times. However, Application scope functions have the following limitations: <ul style="list-style-type: none">• You must lock the code that puts the function name in the Application scope, but you do not have to lock code that calls the function.• Application scope functions can cause processing bottlenecks because the server can only execute one copy of the function at a time. All requests that require the function must wait their turn.
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 <code>cflock</code> 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 180. 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 `CompanyInfo` database to get the first and last names of all employees, 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="CompanyInfo">
    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>
```

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 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 of 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.

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 $4 \times 3 \times 2 \times 1 = 24$.

```
function Factorial(factor)
{
    If (factor LTE 1)
        return 1;
    else
        return factor * Factorial(factor -1);
}
```


Reviewing the code

The following table describes the code:

Code	Description
<pre><cfform method="POST" action="#CGI.script_name#"> <p>Enter your Name:&nbsp; <input name="name" type="text" hspace="30" maxlength="30"> <input type="Submit" name="submit" value="OK"> </cfform></pre>	<p>Creates a simple form requesting you to enter your name.</p> <p>Uses the script_name CGI variable to post to this page without specifying a URL.</p> <p>If you do not enter a name, the form posts an empty string as the name field.</p>
<pre><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></pre>	<p>Defines a function to display "Hello name!" First, checks whether the argument is an empty string. If so, displays an error message.</p> <p>Otherwise displays the hello message.</p> <p>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.</p> <p>If this page has been called by submitting the form, calls the HelloFriend function. Otherwise, the page just displays the form.</p>

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. Which technique 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 182, 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 182 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 182](#).

Code	Description
<pre>function TotalInterest(principal, annualPercent, months, status)</pre>	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.
<pre>if ((principal LE 0) OR (annualPercent LE 0) OR (months LE 0)) { Status.errorMessage = "All values must be greater than 0"; Return -1; }</pre>	Checks to make sure the principal, percent rate, and duration are all greater than zero. If any is not, sets the errorMessage 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 182](#) is in bold>.

```
<cfset status = StructNew(>
<cfset myInterest = TotalInterest(Form.Principal,
    Form.AnnualPercent,Form.Months, status)>
<cfif myInterest EQ -1>
    <cfoutput>
        ERROR: #status.errorMessage#<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
<pre><cfset status = StructNew(></pre>	Creates a structure to hold the function status.
<pre><cfset myInterest = TotalInterest (Form.Principal, Form.AnnualPercent, Form.Months, status)></pre>	Calls the function. This time, the function requires four arguments, including the status variable.

Code	Description
<pre><cfif myInterest EQ -1> <cfoutput> ERROR: #status.errorMessage#
 </cfoutput></pre>	If the function returns -1, there must be an error. Displays the message that the function placed in the status.errorMessage structure key.
<pre><cfelse> <cfoutput> Loan amount: #Form.Principal#
 Annual percentage rate: #Form.AnnualPercent#
 Loan duration: #Form.Months# months
 TOTAL INTEREST: #myInterest#
 </cfoutput> </cfif></pre>	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 191](#).
- Return an error status, as described in [“Providing status information” on page 192](#).
- 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,” in Chapter 14](#).

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 192 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 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 years = 0>
  <cfset interestRate = 0>
  <cfset totalInterest = 0>
  <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>
```

CHAPTER 10

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 198](#)
- [Passing data to custom tags 202](#)
- [Managing custom tags 207](#)
- [Executing custom tags 208](#)
- [Nesting custom tags..... 212](#)

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 208](#).

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 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
<code>template</code>	Required if the <code>name</code> attribute is not used. Same as the <code>template</code> attribute in <code>cfinclude</code> . This attribute: <ul style="list-style-type: none">• Specifies a path relative to the directory of the calling page.• If the path value is prefixed with <code>"/</code>, ColdFusion searches directories explicitly mapped in the ColdFusion Administrator for the included file. Example: <code><cfmodule template=" ../MyTag.cfm"></code> identifies a custom tag file in the parent directory.
<code>name</code>	Required if the <code>template</code> attribute is not used. Use period-separated names to uniquely identify a subdirectory under the CustomTags root directory. Example: <code><cfmodule name="MyApp.GetUserOptions"></code> identifies the file <code>GetUserOptions.cfm</code> in the <code>CustomTags\MyApp</code> directory under the ColdFusion root directory.
<code>attributes</code>	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 208](#).

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 *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 243](#).

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.

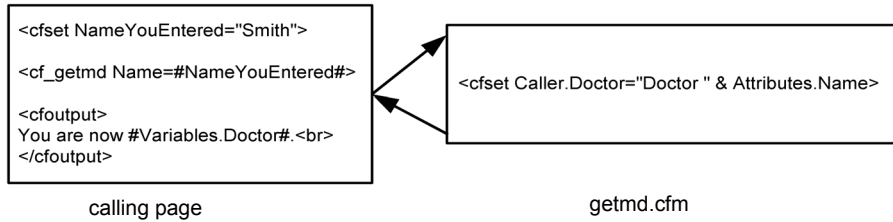
Variables created within a custom tag are deleted when the processing of the tag terminates. Therefore, if you want to pass information back to the calling page, you must write that information back to the `Caller` scope of the calling page. You cannot access the custom tag's variables outside the custom 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-quotes.
- 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
<pre><cfset NameYouEntered="Smith"></pre>	In the calling page, create a variable <code>NameYouEntered</code> and assign it the value "Smith."
<pre><cfoutput> Before you leave this page, you're #Variables.NameYouEntered#.
 </cfoutput></pre>	In the calling page, display the value of the <code>NameYouEntered</code> variable before calling the custom tag.
<pre><cf_getmd Name="#NameYouEntered#"></pre>	In the calling page, call the <code>getmd</code> custom tag and pass it the <code>Name</code> attribute whose value is the value of the local variable <code>NameYouEntered</code> .
<pre><cfparam name="Attributes.Name" default="Who"></pre>	The custom tag page normally gets the <code>Name</code> variable in the <code>Attributes</code> scope from the calling page. Assign it the value "Who" if the calling page did not pass an attribute.
<pre><cfset Caller.Doctor="Doctor " & Attributes.Name></pre>	In the custom tag page, create a variable called <code>Doctor</code> in the <code>Caller</code> 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 <code>Attributes.Name</code> variable.
<pre><cfoutput> You are now #Variables.Doctor#.
 </cfoutput></pre>	In the calling page, display the value of the <code>Doctor</code> variable returned by the custom tag page. (This example uses the <code>Variables</code> 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 347](#).

Encoding custom tags

You can use the command-line utility `cfencode` to encode any ColdFusion application page. By default, the utility is installed in the `cf_root/bin` directory. It is especially useful for securing custom tag code before distributing it.

The `cfencode` tag uses the following syntax:

```
cfencode infile outfile [/r /q] [/h "message"] /v"2"
```

The following table describes the options:

Option	Description
<code>infile</code>	The file you want to encode. The <code>cfencode</code> tag does not process an encoded file.
<code>outfile</code>	Path and filename of the output file. Warning: If you do not specify an output filename, a warning message asks if you want to continue, and the encoded file will overwrite the source file.
<code>/r</code>	Recursive, when used with wildcards, recurses through subdirectories to encode files.
<code>/q</code>	Suppresses warning messages.
<code>/h</code>	Header, allows custom header to be written to the top of the encoded file(s).
<code>/v</code>	Required parameter that allows encoding using a specified version number. Use "1" for pages you want to run on ColdFusion 3.x. Use "2" for pages you want to run strictly on ColdFusion 4.0 and later.

Note: Although it is possible to encode binary files with `cfencode`, it is not recommended.

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
<code>ExecutionMode</code>	Contains the execution mode of the custom tag. Valid values are "start", "end", and "inactive".
<code>HasEndTag</code>	Distinguishes between custom tags that are called with and without end tags. Used for code validation. If the user specifies an end tag, <code>HasEndTag</code> is set to <code>True</code> ; otherwise, it is set to <code>False</code> .
<code>GeneratedContent</code>	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.
<code>AssocAttribs</code>	Contains the attributes of all nested tags if you use <code>cfassociate</code> to make them available to the parent tags. For more information, see "High-level data exchange" on page 213 .

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 212](#).

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 bolds text 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 text to bold:

```
<cf_bold>This is bolded 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 210](#).
- 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 <code>cfexit</code> call	Behavior
ExitTag (default)	Base page	Acts like <code>cfabort</code>
	ExecutionMode=start	Continue after end tag
	ExecutionMode=end	Continue after end tag
ExitTemplate	Base page	Acts like <code>cfabort</code>
	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 its 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 <code>cfassociate</code> .

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 11

Building and Using ColdFusion Components

ColdFusion components let you encapsulate and re-use code in ColdFusion development, generate web services, and create Flash interfaces for your application.

Contents

- [About ColdFusion components](#) 218
- [Building ColdFusion components](#) 219
- [Interacting with component methods](#) 222
- [Using advanced ColdFusion component functionality](#) 234

About ColdFusion components

ColdFusion components encapsulate application functionality and provide a standard interface for client access to that functionality. Clients access component functionality by invoking methods on components. Components support a variety of client interfaces, including web pages, Flash movies, web services, and other objects accessible from ColdFusion components and pages. Component method invocation serves as the gateway to component functionality, including passing parameters and receiving component method results.

Like other ColdFusion Markup Language (CFML) code reuse techniques, such as user-defined functions (UDFs) and custom CFML tags, components let you create application functionality that can be reused wherever you need it. If you want to modify, add, or remove component functionality, you only need to make changes in one component file.

Note: For more information about UDFs, custom tags, and other ColdFusion code reuse techniques, see [Chapter 8, “Reusing Code in ColdFusion Pages”](#) on page 157.

Applying design patterns to component development

As your development projects grow larger and teams of developers become involved, ColdFusion components can structure CFML to serve as building blocks for design pattern methodologies.

Established design pattern specifications represent the accumulated knowledge of veteran software developers, which is used to establish guidelines for application development. When applied correctly, design patterns streamline software production, manage the application development process, and ensure code maintainability for the life cycle of the application.

When making the decision about whether to use a design pattern methodology for a development project, keep the following points in mind:

- While implementing a design pattern methodology involves more planning initially, you will save time and money later in the development cycle.
- Each design pattern methodology has strengths and weaknesses. Select the methodology that best fits your development project needs.

For more information about design patterns, see *Rapid Development: Taming Wild Software Schedules*, Steve McConnell, 1996: Microsoft Press.

Building ColdFusion components

Just like ColdFusion pages, you store component files in a domain accessible by your web server and ColdFusion. Unlike ColdFusion pages, you save component files with the CFC suffix, such as `componentName.cfc`.

Save your component files in one of the following locations:

- Directories accessible from the web server, which includes the web root and web server virtual directories.
- Directories accessible from ColdFusion mappings.
- Subdirectories of custom tag roots.

Note: For more information about saving components and component naming conventions, see [“Using component packages” on page 237](#).

All ColdFusion variable scopes are available to components, including `Session`, `Client`, `Server`, and `Application`. In addition, the `This` scope is available during component method execution.

You use the `cfcomponent` and `cffunction` tags to create ColdFusion components. By itself, the `cfcomponent` tag does not provide functionality. Rather, the `cfcomponent` tag provides an envelope that describes the functionality that you build in CMFL and enclose in `cffunction` tags.

Syntax for the `cfcomponent` tag

```
<cfcomponent extends="anotherComponent">
```

The following table displays the tag attribute, data type, and description:

Attribute	Type	Required	Description	For more information
<code>extends</code>	string	no	Name of parent component.	See “Using component inheritance” on page 239 .

Note: The `cfcomponent` tag is optional.

Syntax for the `cffunction` tag

```
<cffunction name="methodName" returnType="dataType"  
  roles="securityRoles" access="methodAccess" output="yes/no">
```

The following table displays the tag attribute, data type, and description:

Attribute	Type	Required	Description	For more information
<code>name</code>	string	yes	Name of component method	See “Defining component methods” on page 220 .
<code>returnType</code>	string	no	Data type validation for returned values.	See “Returning values from component methods” on page 232 .
<code>roles</code>	string	no	Assigns component method to ColdFusion security roles.	“Building secure ColdFusion components” on page 234 .

Attribute	Type	Required	Description	For more information
access	string	no	Restricts component method access by client type.	See “Building secure ColdFusion components” on page 234
output	Boolean	no	Suppresses component method output	See “Building secure ColdFusion components” on page 234.

The following example creates a component with two methods:

```
<cfcomponent>
  <cffunction name="getEmp">
    <cfquery name="empQuery" datasource="ExampleApps" dbtype="ODBC" >
      SELECT FIRSTNAME, LASTNAME, EMAIL
      FROM tblEmployees
    </cfquery>
    <cfreturn empQuery>
  </cffunction>
  <cffunction name="getDept">
    <cfquery name="deptQuery" datasource="ExampleApps" dbtype="ODBC" >
      SELECT *
      FROM tblDepartments
    </cfquery>
    <cfreturn deptQuery>
  </cffunction>
</cfcomponent>
```

In the example, two `cffunction` tags 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. For more information, see [“Invoking component methods” on page 222](#).

Defining component methods

Component method definitions exist between opening and closing `cffunction` tags. To separate the component method code from the component file, use the `cfinclude` tag to call the page that contains the component method code.

To create a component method:

- 1 Create a new ColdFusion component, and save it as `tellTime.cfc` in a directory below your web-root directory.
- 2 Modify the code so that it appears as follows:

```
<cfcomponent>
  <cffunction name="getLocalTime">
    <cfscript>
      serverTime=now();
      localStructure=structNew();
      localStructure.Hour=DatePart("h", serverTime);
      localStructure.Minute=DatePart("n", serverTime);
    </cfscript>
  </cffunction>
```

```

        <cfoutput>
            #localStructure.Hour#: #localStructure.Minute#
        </cfoutput>
    </cffunction>
</cfcomponent>

```

In the example, the `cfscript` and `cfoutput` statements execute during component method processing.

3 Save your work.

By placing the method execution code in a separate file, template methods separate execution and markup code from the component method definitions.

To create component method using the `cfinclude` tag:

1 Open the `tellTime.cfc` file, and modify the code so that it appears as follows:

```

<cfcomponent>
    <cffunction name="getLocalTime">
        <cfinclude template="getTime.cfm">
    </cffunction>
</cfcomponent>

```

In the example, the `getLocalTime` method definition calls the `getTime.cfm` file with the `cfinclude` tag.

2 Save your work.

3 Create a ColdFusion page, and save it as `getTime.cfm` in the same directory as `tellTime.cfc`.

4 Modify `getTime.cfm` so that the code appears as follows:

```

<cfscript>
    serverTime=now();
    localStruct=structNew();
    localStruct.Hour=DatePart("h", serverTime);
    localStruct.Minute=DatePart("n", serverTime);
</cfscript>
<cfoutput>#localStruct.Hour#: #localStruct.Minute#</cfoutput>

```

In the example, a CFScript statement uses the `now()` and `DatePart()` functions to populate a structure with hour and minute values. The values are then displayed with the `cfoutput` tag. Notice that no value is returned to the client. Instead, the `getTime` method displays the variable.

5 Save your work.

Interacting with component methods

The vast majority of ColdFusion applications require data to be passed back and forth between a number of pages. For example, a typical web shopping cart application uses multiple ColdFusion pages to gather user data, access databases, and confirm credit card information.

ColdFusion components support passing and returning simple and complex values using the `cfinvoke` tag, URL and form controls, CFScript, the Macromedia Flash Remoting service, and web services. Whether you are receiving registration information from a simple HTML page or passing a query object back to a sophisticated web service, interacting with ColdFusion components means that you must be able to pass data into and out of a component.

Interacting with components consists of the following operations:

- **Invoke a component method** Use the `cfinvoke` tag in ColdFusion pages and components, the HTTP form methods GET and POST, CFScript invocation, Flash Remoting invocation, or web service invocation. For more information, see [“Invoking component methods” on page 222](#).
- **Pass a parameter to a component method** Do three things: define the parameter in the component method definition, choose a parameter-passing technique, and access the data passed in the parameter. For more information, see [“Passing parameters to component methods” on page 226](#).
- **Return a value from a component method** Do two things: insert the `cfreturn` tag into the component method definition to specify a variable to return to the client, and access the returned values in the client. For more information, see [“Returning values from component methods” on page 232](#).

Invoking component methods

To interact with ColdFusion components, you invoke component methods from the client. Components support many client types, including web pages, ColdFusion pages, Flash movies, web services, and other components. The invocation process depends on what type of client invokes a component method.

The following table displays the different ways to invoke component methods:

Invocation	Description	For more information
<code>cfinvoke</code> tag	The <code>cfinvoke</code> tag instantiates and invokes component methods from within ColdFusion pages and components.	See “Invoking component methods using the <code>cfinvoke</code> tag” on page 223 .
<code>cfobject</code> tag	The <code>cfobject</code> tag instantiates a component. However, you must still use the <code>cfinvoke</code> tag or CFScript to invoke component methods, pass parameters, and return results.	See “Invoking component methods using the <code>cfobject</code> tag” on page 225 .

Invocation	Description	For more information
URL control (HTTP GET)	You use the component and method names in the URL string to invoke component methods.	See “Invoking component methods using a URL” on page 225 .
Form control (HTTP POST)	HTML and ColdFusion forms invoke component methods using the HTML form and input tags and their attributes.	See “Invoking component methods using a form” on page 225 .
CFScript	CFScript instantiates component methods using the <code>createObject</code> function. The component method can then be called using <code>componentName.componentMethod()</code> syntax.	See “Invoking component methods with CFScript” on page 226 .
Flash Remoting	In client-side ActionScript, you use the NetServices functions to invoke component methods.	See Chapter 29, “Using the Flash Remoting Service” on page 673 .
Web services	You use the <code>cfinvoke</code> tag and CFScript to consume web services in ColdFusion.	See Chapter 31, “Using Web Services” on page 729 .

Note: To restrict component method invocation, you use the `access` and `roles` attributes of the `cffunction` tag. For more information, see [“Using web server authentication” on page 234](#).

Note: To invoke components within the component method definition, you use the `cfinvoke` tag with its `method` attribute. In CFScript, you use the method name in standard function syntax, such as `methodName()`.

Invoking component methods using the `cfinvoke` tag

In ColdFusion pages or components, use the `cfinvoke` tag to invoke component methods. You can place multiple `cfinvoke` tags in a ColdFusion page to invoke multiple component methods.

Syntax for the `cfinvoke` tag

```
<cfinvoke component="componentName" method="methodName"
returnVariable="variableName" argumentCollection="argumentStruct">
```

The following table displays the tag attribute, data type, and description:

Attribute	Type	Required	Description	For more information
<code>component</code>	string	yes	Name of component	
<code>method</code>	string	yes	Name of component method	See “Invoking component methods using the <code>cfinvoke</code> tag” on page 223 .

Attribute	Type	Required	Description	For more information
returnVariable	string	no	Creates a variable by the name entered and assigns the component method results into that variable	See “Returning values from component methods” on page 232.
argumentCollection	structure	no	Passes structure to component method as parameters	See “Passing parameters to component methods” on page 226

To invoke a component method using the cfinvoke tag:

- 1 Open the tellTime.cfc file, and modify the code so that it appears as follows:

```

<cfcomponent>
  <cffunction name="getLocalTime">
    <cfscript>
      serverTime=now();
      localStruct=structNew();
      localStruct.Hour=DatePart("h", serverTime);
      localStruct.Minute=DatePart("n", serverTime);
    </cfscript>
    <cfoutput>#localStruct.Hour#: #localStruct.Minute#</cfoutput>
  </cffunction>
  <cffunction name="getUTCTime">
    <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;
    </cfscript>
    <cfoutput>#utcStruct.Hour#: #utcStruct.Minute#</cfoutput>
  </cffunction>
</cfcomponent>

```

The example defines two component methods, getLocalTime and getUTCTime.

- 2 Create a new ColdFusion page, and save it as timeDisplay.cfm in the same directory as the tellTime component.
- 3 Modify the ColdFusion page so that it appears as follows:

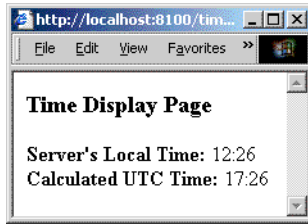
```

<h3>Time Display Page</h3>
<b>Server's Local Time:</b>
<cfinvoke component="tellTime" method="getLocalTime"><br>
<b>Calculated UTC Time:</b>
<cfinvoke component="tellTime" method="getUTCTime">

```

Using the cfinvoke tag, the example invokes the getLocalTime and getUTCTime component methods.

- 4 The following figure shows the results when you execute `timeDisplay.cfm` in a web browser:



Invoking component methods using the `cfoject` tag

To separate the instantiation of the component and the invocation of the component method, use the `cfoject` tag. First, use the `cfoject` tag to instantiate the component and assign the component to a variable; for example:

```
<cfoject name="tellTimeComp" component="tellTime">
```

To invoke component methods, use the `cfinvoke` tag. The `cfinvoke` tag's `name` attribute references the variable name in the `cfoject` tag's `name` attribute; for example:

```
<cfoject name="tellTimeComp" component="tellTime">
<cfinvoke component="#tellTimeComp#" method="getLocalTime">
<cfinvoke component="#tellTimeComp#" method="getUTCtime">
```

Invoking component methods using a URL

To invoke a component method using a URL, you must append the method name to the URL in the standard URL query-string, name-value syntax. You can only invoke one component method per URL request; for example:

```
http://localhost:8500/tellTime.cfc?method=getLocalTime
```

Note: To use URL invocation, you must set the `cffunction` tag's `access` attribute to `remote`.

Invoking component methods using a form

To invoke a method using a ColdFusion or an HTML form, you must enter the file path to the ColdFusion component in the `action` attribute and the method name as a form variable that is submitted.

Note: To use form invocation, you must set the `cffunction` tag's `access` attribute to `remote`.

To invoke component methods using a form:

- 1 Open `timeDisplay.cfm`, and modify the page so that it appears as follows:

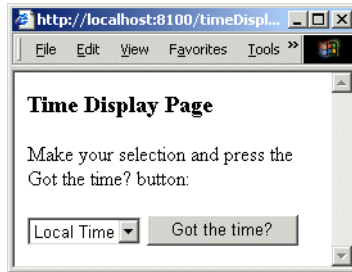
```
<h3>Time Display Page</h3>
<p>Make your selection and press the Got the time? button:</p>
<cfform action="tellTime.cfc" method="POST">
<cfselect name="Method" required="Yes">
  <option value="getLocalTime" selected>Local Time</option>
  <option value="getUTCtime">UTC Time</option>
</cfselect>
<input type="submit" value="Got the time?">
</cfform>
```

In the example, the `cfform` tag's `action` attribute points toward the `tellTime` component file. The `cfselect` statement passes the component method name.

- 2 Save your work.
- 3 Start your web browser, and browse to the following URL:

`http://localhost:8500/timeDisplay.cfm`

The following figure shows the results:



Make a selection from the drop-down box, and click the Got the Time? button. Depending on your selection, the server's local or UTC time displays.

Invoking component methods with CFScript

To invoke a component method using CFScript, use the `createObject` function or `cfobject` tag to instantiate the component. After you instantiate the component, you use normal function syntax to invoke component methods; for example:

```
<!-- instantiate once and reuse the instance-->
<cfscript>
    tellTimeCFC=createObject("component","tellTime");
</cfscript>
<b>Server's Local Time:</b>
    <cfscript>
        tellTimeCFC.getLocalTime();
    </cfscript>
<br>
<b>Calculated UTC Time:</b>
    <cfscript>
        tellTimeCFC.getUTCtime();
    </cfscript>
```

In the example, the two CFScript statements assign the `tellTimeCFC` variable to the `tellTime` component using the `createObject` function. Next, you use normal function syntax to invoke the component method.

Passing parameters to component methods

To perform conditional processing in ColdFusion components based on data sent from the client, you pass parameters to component methods. In ColdFusion applications, parameters typically consist of user name and password information, session state data, keywords for database queries, and so on.

To pass parameters in ColdFusion components:

- 1 Define the parameter in the component method definition using the `cfargument` tag. For more information, see [“Defining the parameter in the component method definition” on page 227](#).
- 2 Choose your parameter-passing technique. Use the parameter-passing technique best suited for your client type. For more information, see [“Choosing a parameter-passing technique” on page 228](#).

Defining the parameter in the component method definition

In the component method, you create parameter definitions using the `cfargument` tag within the component method definition. You define multiple parameter with multiple `cfargument` tags. To access the parameter values in the component method definition, you use structure- or array-like notation with the `argument` variable.

Syntax for the `cfargument` tag

```
<cfargument name="parameterName" type="dataType"
required="true/false" default="defaultValue">
```

The following table displays the tag attribute, data type, and description:

Attribute	Type	Required	Description
name	string	yes	Name of parameter
type	data type	no	Validates all valid data types
required	Boolean	no	Specifies whether the parameter is required to execute the component method
argumentCollection	all types	no	Provides a default value when a parameter is not passed

Also, if the `required` attribute is not set to `true`, you can specify a default value for the parameter value using the `default` attribute. The following example defines two parameters and references the parameter values in the component method definition.

Note: For the following procedures to work, you must have the example applications installed with ColdFusion. For more information, see *CFML Reference*.

To define parameters in the component method definition:

- 1 Create a new component, and save it as `corpQuery.cfc` in a directory under your web root directory.
- 2 Modify the code in `corpQuery.cfc` so that it appears as follows:

```
<cfcomponent>
  <cffunction name="getEmp">
    <cfargument name="lastName" required="true">
      <cfquery name="empQuery" datasource="ExampleApps" dbtype="ODBC">
        SELECT LASTNAME, FIRSTNAME, EMAIL
        FROM tblEmployees
        WHERE LASTNAME LIKE '#arguments.lastName#'
      </cfquery>
    </cfargument>
  </cffunction>
</cfcomponent>
```

```

        <cfoutput>Results filtered by #arguments.lastName#:</cfoutput><br>
        <cfdump var=#empQuery#>
    </cffunction>
    <cffunction name="getCat">
    <cfargument name="cost" required="true">
        <cfquery name="catQuery" datasource="ExampleApps" dbtype="ODBC">
            SELECT ItemName, ItemDescription, ItemCost
            FROM tblItems
            WHERE ItemCost <= #arguments.cost#
        </cfquery>
        <cfoutput>Results filtered by #arguments.cost#:</cfoutput><br>
        <cfdump var=#catQuery#>
    </cffunction>
</cfcomponent>

```

In the example, the `cfargument` tag's name attribute defines the parameter's name. The required attribute indicates that the parameter is required or an exception will be thrown. The `arguments` variable scope provides access to the parameter values.

Note: You can also reference multiple parameter values using array- and structure-like syntax. For example, `arguments.cost` is the same as `argument[1]`. Array and structure-like notation also lets you loop over multiple parameters. In addition, you can access `arguments` directly using pound signs, such as `#cost#`.

3 Save your work.

Choosing a parameter-passing technique

Like ColdFusion pages, you can pass parameters using a URL or the HTTP GET and POST form methods with ColdFusion components. Components also accept passing parameters using the `cfinvoke` tag.

The following table describes your parameter-passing options:

Parameter type	Description	For more information
<code>cfinvoke</code> tag	Specify the parameters as <code>cfinvoke</code> tag attributes or the <code>argumentsCollection</code> attribute.	See "Passing parameters using the <code>cfinvoke</code> tag" on page 229.
<code>cfinvokeargument</code> tag	Specify parameter name and values using the <code>cfinvokeargument</code> tag.	See "Passing parameters using the <code>cfinvokeargument</code> tag" on page 229.
URL	Specify the parameters in the standard URL query-string, name-value pair syntax.	See "Passing parameters using a URL" on page 230.
Form	Specify the parameters as form input values.	See "Passing parameters using a form" on page 230.
CFScript	Specify the parameters as ordered arguments or named arguments.	See "Passing parameters using CFScript" on page 232.

Parameter type	Description	For more information
Flash Remoting	Specify the parameters in client-side ActionScript.	See Chapter 29, “Using the Flash Remoting Service” on page 673.
Web services	Specify the parameters as cfinvoke tag attributes or the argumentsCollection attribute.	See Chapter 31, “Using Web Services” on page 729.

Passing parameters using the cfinvoke tag

You can pass a single or multiple parameters in one cfinvoke tag as tag attribute name-value pairs. The following example passes a single parameter:

```
<cfinvoke component="authQuery" method="getAuth" lastName=session.username>
```

In the example, the lastName attribute passes the value of the session scope variable to the component method. To pass multiple parameters, use an attribute name-value pair for each parameter; for example:

```
<cfinvoke component="authQuery" method="getAuthSecure"
    lastName=session.username password=#url.password#>
```

In the example, the parameters are passed as the lastName and password attributes. Notice that different variable scopes are used in the attribute values.

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*.

If you save attributes to a structure, you can directly pass the structure using the cfinvoke tag’s argumentCollection attribute.

The following example invokes a component that performs simple addition and subtraction:

```
<cfscript>
    exampleStruct = StructNew();
    exampleStruct[1] = 1;
    exampleStruct[2] = 2;
</cfscript>
<cfinvoke component="arithCFC" method="add" argumentCollection=exampleStruct>
```

This example passes two parameters to the component method as a structure. Notice the use of the argumentCollection attribute of the cfinvoke tag.

Passing parameters using the cfinvokeargument tag

To pass parameters independently of the cfinvoke tag, use the cfinvokeargument tag.

Using the cfinvokeargument tag, for example, you can build conditional processing that passes a different parameter based on user input.

Syntax for the cfinvokeargument tag

```
<cfinvokeargument name="parameterName" value="anyValue">
```

The following table displays the tag attribute, data type, and description:

Attribute	Type	Required	Description
name	string	yes	Name of parameter
value	all types	yes	Value of parameter

The following example invokes the corpQuery component:

```
<cfinvoke component="corpQuery" method="getEmp">  
<cfinvokeargument name="lastName" value="camden">
```

Notice that the cfinvokeargument tag passes the lastName parameter to the component method.

Note: For more information about parameter precedence, see *CFML Reference*.

Passing parameters using a URL

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. Here is an example:

```
http://localhost:8500/  
corpQuerySecure.cfc?method=getAuth&store=women&dept=shoes
```

Note: Due to security concerns, Macromedia strongly recommends that you do 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.

Passing parameters using a form

To pass parameters to components using an HTML or ColdFusion form, the names of the client input controls must match the names of the parameter definition in the component file.

To pass parameters using a form:

- 1 Open the corpFind.cfm file and modify the code so that it appears as follows:

```
<h2>Find People and Products</h2>  
<form action="components/corpQuery.cfc" 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>  
<form action="components/corpQuery.cfc" method="post">  
  <p>Enter maximum product price:</p>  
  <input type="Text" name="cost">  
  <input type="Hidden" name="method" value="getCat">
```



```
<input type="Submit" title="Submit Query">
</form>
```

In the example, the form tag action attribute points to the corpQuery component. The input tags invoke the component method.

- 2 Open corpQuery.cfc and add access="remote" to each cffunction tag, as the following example shows:

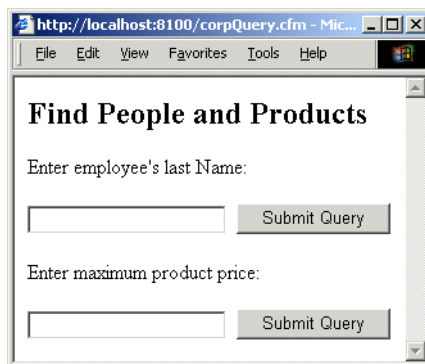
```
<cfcomponent>
  <cffunction name="getEmp" access="remote">
    <cfargument name="lastName" required="true">
      <cfquery name="empQuery" datasource="ExampleApps" dbtype="ODBC">
        SELECT LASTNAME, FIRSTNAME, EMAIL
        FROM tblEmployees
        WHERE LASTNAME LIKE '#arguments.lastName#'
      </cfquery>
      <cfoutput>Results filtered by #arguments.lastName#:</cfoutput><br>
      <cfdump var=#empQuery#>
    </cffunction>
    <cffunction name="getCat" access="remote">
      <cfargument name="cost" required="true">
        <cfquery name="catQuery" datasource="ExampleApps" dbtype="ODBC">
          SELECT ItemName, ItemDescription, ItemCost
          FROM tblItems
          WHERE ItemCost <= #arguments.cost#
        </cfquery>
        <cfoutput>Results filtered by #arguments.cost#:</cfoutput><br>
        <cfdump var=#catQuery#>
      </cffunction>
</cfcomponent>
```

In this example, the cffunction access attribute lets remote clients, such as web browsers and Flash applications, to access component methods.

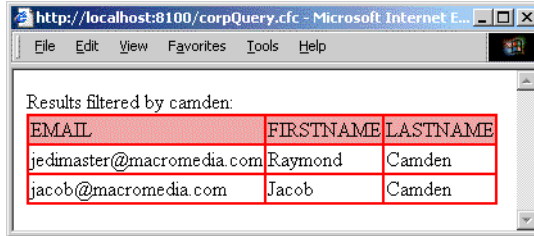
- 3 Save your work.
- 4 Open a web browser and enter the following URL:

<http://localhost:8100/corpQuery.cfm>

The following figure shows the results:



Depending on what you enter, after you click the Submit Query button, the web browser displays the results, as shown in the following figure:



Passing parameters using CFScript

The following example instantiates a component, invokes the `getAuth` component method in three different ways, and passes parameters in each method invocation:

```
<cfscript>
corpQCFC = createObject("component", "corpSecurity");
corpQCFC.getAuth(username="skippy" password="dippy");
tempStruct = structNew();
tempStruct.username = "skippy"
tempStruct.password = "dippy"
corpQCFC.getAuth(argumentsCollention = tempStruct);
corpQCFC.getAuth("skippy", "dippy");
</cfscript>
```

Returning values from component methods

In the component method definition, you return the results to the client using the `cfreturn` tag. The equivalent to the `return` CFScript statement, the `cfreturn` tag only accepts one variable to return at a time. Therefore, if you want to return more than one result value at a time, populate a structure with name-value pairs and return the structure using the `cfreturn` tag.

To access the result values returned to the client, use the variable scope specified as the value of the `cfinvoke` tag's `returnVariable` attribute.

Returning component method results to the client

To return component method results to the client, use the `cfreturn` tag in the component method definition. You can pass values of all data types, including strings, integers, arrays, and structures.

To prepare the component method definition to return a value:

- 1 Open the `corpQuery.cfc` file, and modify the code so that it appears as follows:

```
<cfcomponent>
  <cffunction name="getEmp">
    <cfquery name="empQuery" datasource="ExampleApps" dbtype="ODBC">
      SELECT LASTNAME, FIRSTNAME, EMAIL
      FROM tblEmployees
    </cfquery>
  </cffunction>
```

```

    <cfreturn empQuery>
</cffunction>
<cffunction name="getCat">
    <cfquery name="catQuery" datasource="ExampleApps" dbtype="ODBC">
        SELECT ItemName, ItemDescription, ItemCost
        FROM tblItems
    </cfquery>
    <cfreturn catQuery>
</cffunction>
</cfcomponent>

```

In the example, the `cfreturn` tags return the query objects created by the component methods.

2 Save your work.

3 Open the `corpFind.cfm` file, and modify the code so that it appears as follows:

```

<cfinvoke component="corpQuery" method="getEmp" returnVariable="empResult">
<cfdump var="#empResult#">

```

In the example, the `cfinvoke` tag's `returnVariable` attribute specifies the variable scope name that holds the component method results. The `cfdump` tag displays the contents of the `empResult` variable.

4 Open a web browser and browse to the following URL:

<http://localhost/corpFind.cfm>

The following figure shows the results:

EMAIL	FIRSTNAME	LASTNAME
scheng@macromedia.com	Stephen	Cheng
jberrey@macromedia.com	Joe	Berrey
alipinsky@macromedia.com	Adam	Lipinsky
lteague@macromedia.com	Lynne	Teague
vgilson@macromedia.com	Victoria	Gilson
creiff@macromedia.com	Charles	Reiff
vchin@macromedia.com	Vicki	Chin

Using advanced ColdFusion component functionality

Beyond basic component functionality, ColdFusion components offer advanced functionality to streamline application development, deployment, and extensibility. The following table displays advanced component functionality:

Feature	Description	For more information
Component method security	Using the <code>roles</code> and <code>access</code> attributes of the <code>cffunction</code> tag, you build component method-level security measures.	See “Building secure ColdFusion components” on page 234 .
Component packages	Using component packages, you avoid possible naming conflicts with components.	See “Using component packages” on page 237 .
Component inheritance	Using the <code>extends</code> attribute of the <code>cfcomponent</code> tag, you import another component’s methods and properties.	See “Using component inheritance” on page 239 .
Component introspection	Using component metadata, you can describe component functionality programmatically.	See “Using component metadata” on page 240 .

Building secure ColdFusion components

To restrict access to component methods, ColdFusion components use the following security features:

- 1 Web server basic authentication
For more information, see [“Using web server authentication” on page 234](#).
- 2 Application security
For more information, see [“Using ColdFusion application security” on page 235](#).
- 3 Role-based security
For more information, see [“Using role-based security” on page 236](#).
- 4 Programmatic security
For more information, see [“Using programmatic security” on page 237](#).

Using web server authentication

The majority of web servers allow directory access protection using basic authentication. When a client tries to access one of the resources under a protected directory and is not properly authenticated, the server automatically sends back an authentication challenge to the web browser. The web browser shows a login dialog box.

When you enter your authentication information, the web browser authenticates the information to the web server. If the authentication passes, the web browser caches the authentication data while the browser window is open and every subsequent request to the web server sends the same authentication data.

ColdFusion developers can use the authentication information for ColdFusion resources, such as ColdFusion pages or components, in the appropriate application.cfm file, as the following example shows:

```
<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>
```

Using ColdFusion application security

You can use the previous example with minor modification to include the login challenge in the application.cfm file as well. You can create an HTML form page that passes authentication information to ColdFusion, or you can return the access-denied 401 information back to the web browser.

The following example shows an authentication challenge by generating an HTML page with a login form. The login form sends two form fields, `j_username` and `j_password`, to ColdFusion, which are automatically detected by the `cflogin` tag.

```
<cflogin>
  <cfif IsDefined( "cflogin" )>
    <cfif cflogin.name eq "admin" and cflogin.password eq "p1">
      <cfset roles = "user,admin">
    <cfelseif cflogin.name eq "user" and cflogin.password eq "p2">
      <cfset roles = "user">
    </cfif>
  </cfif>
  <cfif IsDefined( "roles" )>
    <cfloginuser name="#cflogin.name#"
      password="#cflogin.password#"
      roles="#roles#">
  <cfelse>
    <!--- authentication failed - generate the login form --->
    <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
    <html><head><title>Application Log In</title></head>
    <body>
      <form action="" method="post">
        <pre>
          username: <input type="text" name="j_username">
          password: <input type="password" name="j_password">
          <input type="submit" value="log in">
        </pre>
```

```

        </form>
    </body>
</html>
    <cfabort>
</cfif>
</cflogin>

```

When you return a 401 access denied response, the browser automatically displays a login dialog box. When the user enters his or her login dialog, the authentication parameters are passed in the request header and are detected by the `cflogin` tag, as shown in the following example:

```

<cflogin>
    <cfif IsDefined( "cflogin" ) >
        <cfif cflogin.name eq "admin" and cflogin.password eq "p1">
            <cfset roles = "user,admin">
        <cfelseif cflogin.name eq "user" and cflogin.password eq "p2">
            <cfset roles = "user">
        </cfif>
    </cfif>
    <cfif IsDefined( "roles" ) >
        <cfloginuser name="#cflogin.name#" password="#cflogin.password#"
            roles="#roles#">
    <cfelse>
        <!-- authentication failed - send back 401 -->
        <cfsetting enablecfoutputonly="yes" showdebugoutput="no">
        <cfheader statuscode="401">
        <cfheader name="WWW-Authenticate" value="Basic realm=""MySecurity""">
        <cfoutput>Not authorized</cfoutput>
        <cfabort>
    </cfif>
</cflogin>

```

The security realm name can be used to bind multiple directories together. If `Application.cfm` files located in those directories use the same realm name, only a single login is required to access resources in those directories. However, each `Application.cfm` file can establish different roles for a user.

Using role-based security

Access to a particular method in component can be restricted using roles security. When a component method is restricted to one or more roles using the `roles` attribute of the `cffunction` tag, users must fall into one of the security roles, as shown in the following example:

```

<cffunction name="foo" roles="admin,moderator">
    .
    .
    .
</cffunction>

```

Use the `cfloginuser` tag to establish the security roles. The `cflogin` tag caches the authentication information. When a user tries to invoke a method that he or she is not authorized to invoke, an exception is returned. For more information, see [Chapter 16, “Securing Applications” on page 347](#).

Using programmatic security

In the component method definition, you can protect resources using the same CFML constructs as ColdFusion pages. For example, the `IsUserInRole` function determines whether the user is authenticated in a particular security role:

```
<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 component packages

Components invoked by ColdFusion pages do not need to be in the same directory as the client ColdFusion page or component, web page, or Macromedia Flash movie. In fact, components can reside in any folder under the web root directory or virtual directory mapping in the web server, in a directory under a ColdFusion mapping, or the custom tag roots.

Components stored in the same directory are members of a component package. Component packages help prevent naming conflicts and facilitate easy component deployment.

To invoke a packaged component method using the `cfinvoke` tag:

- 1 In your web root directory, create a folder named *appResources*.
- 2 In the *appResources* directory, create a folder named *components*.
- 3 Move `tellTime.cfc` and `utcTimeFormatted.cfm` to the *components* directory.
- 4 Create a new ColdFusion page and save it in your web root as `timeDisplay.cfm`.
- 5 Modify the page so that it appears as follows:

```
<h3>Time Display Page</h3>
<b>Server's Local Time:</b>
<cfinvoke component="appResources.components.tellTime"
  method="getLocalTime"><br>
<b>Calculated UTC Time:</b>
<cfinvoke component="appResources.components.tellTime"
  method="getUTCtime">
```

You use dot syntax to navigate directory structures. Prefix the directory name before the component name.

- 6 Save your work.

The following example shows a CFScript invocation:

```
<cfscript>
helloCFC = createObject("component", "appResources.components.catQuery");
helloCFC.getSaleItems();
</cfscript>
```

The following example shows an URL invocation:

```
http://localhost/appResources/components/catQuery.cfc?method=getSalesItems
```

Saving ColdFusion components

The following table contains the locations in which you can save component files and the available accessibility options from each location:

	Web root	ColdFusion mappings	Custom tag roots	Current directory
URL	Yes	Yes	No	Yes
Form	Yes	No	No	Yes
Flash Remoting	Yes	No	No	Yes
Web services	Yes	No	No	Yes
Local	Yes	Yes	Yes	Yes

Note: ColdFusion 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 service invocation.

Naming ColdFusion components

Establishing a descriptive naming convention is a good practice, especially if the components will be installed as a part of packaged application. Like the common Java naming convention, you can reserve the order of your domain name, continue with application name, and so on, as the following example shows:

```
com.mycompany.catalog.product.saw
```

When you refer to a component using the fully qualified name, ColdFusion looks for the component in the following order:

- ColdFusion attempts to resolve the physical path from the request, such as `/com/mycompany/catalog/product/saw.cfc`, to a component file located in directories under the web root or directories under ColdFusion mappings.
- Otherwise, ColdFusion attempts to resolve the physical path in the custom tag root, such as `{customTagRoot}/com/mycompany/catalog/product/saw.cfc`.

When a component is invoked using any of the interfaces mentioned previously, ColdFusion generates the key name in the component metadata structure in the following order:

- If a component file exists in a directory accessible by ColdFusion mappings, use `GetRealPath` function to evaluate the component physical path. The URI path string after `.cfc` and the leading slash is removed, and all slashes are replaced with dots.
- Otherwise, ColdFusion loops over the custom tag roots looking for the ancestor directory of the component. The physical path string after the root path and file extension are removed, and all slashes are replaced with dots
- Otherwise, ColdFusion uses the file name without the extension as the component name.

Using component inheritance

Component inheritance lets you import component methods and properties from one component into another component. In addition, inherited components also share any component methods or properties that they inherit from other components.

When using component inheritance, inheritance should define an *is a* relationship between components. For example, a component named `president.cfc` inherits the component methods of `manager.cfm`, which inherits its methods from `employee.cfc`. In other words, `president.cfc` *is a* `manager.cfc`. The `manager.cfc` *is a* `employee.cfc`. In turn, `president.cfc` *is a* `employee.cfc`.

To use component inheritance:

- 1 Open the `corpQuery.cfc` file, and modify the code so that it appears as follows:

```
<cfcomponent extends="appResources.components.tellTime">
  <cffunction name="getEmp" returnType="query">
    <cfargument name="lastName" required="yes">
      <cfquery name="empQuery" datasource="ExampleApps" dbtype="ODBC">
        SELECT LASTNAME, FIRSTNAME, EMAIL
        FROM tblEmployees
        WHERE LASTNAME LIKE '#arguments.lastName#'
      </cfquery>
      <cfif empQuery.recordcount LT 1>
        <cfthrow type="noQueryResult"
          message="No results were found. Please try again.">
      <cfelse>
        <cfreturn empQuery>
      </cfif>
    </cffunction>
    <cffunction name="getCat" returnType="query">
      <cfquery name="catQuery" datasource="ExampleApps" dbtype="ODBC">
        SELECT ItemName, ItemDescription, ItemCost
        FROM tblItems
      </cfquery>
      <cfif #getCat.recordcount# LT 1>
        <cfthrow type="noQueryResult"
          message="No results were found. Please try again.">
      <cfelse>
        <cfreturn catQuery>
      </cfif>
    </cffunction>
  </cfcomponent>
```

In the example, the `cfcomponent` tag's `extends` attribute points to the `tellTime` component.

- 2 Save your work.
- 3 Create a new ColdFusion page, and save it as `inherit.cfm` in your web-root directory.
- 4 Modify the code in the `inherit.cfm` file so that it appears as follows:

```
<cfinvoke component="corpQuery" method="getEmp" lastName="gilson">
<cfinvoke component="corpQuery" method="getLocalTime">
```
- 5 Save your work.

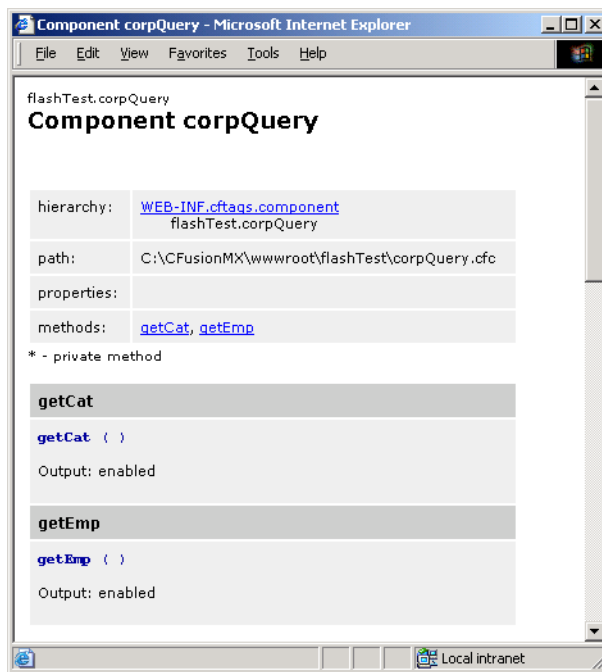
When you execute the `inherit.cfm` file, the `getLocalTime` component method executes like the `getEmp` component method.

Using component metadata

When you access a ColdFusion component directly with a web browser without specifying a component method, the following chain of events occurs:

- The request is redirected to `CFCEXplorer.cfc`, which is located in the `[webroot]\CFIDE\componentutils` directory.
- The `CFCEXplorer` component prompts users for the ColdFusion RDS password.
- The `CFCEXplorer` renders an HTML description. For example, when the `corpQuery` component is accessed directly by a web browser, it produces the following results:

The following figure shows the HTML description for the `corpQuery` component:

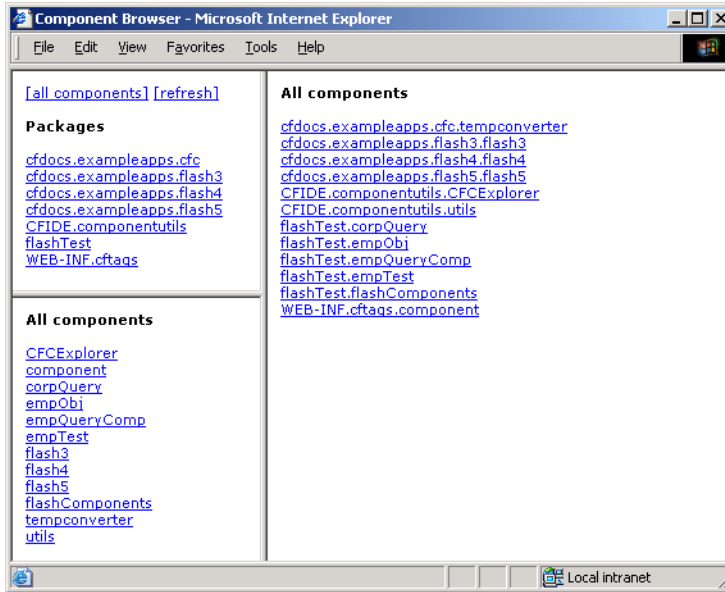


The description that displays in the web browser, components list the methods that you build. Development teams can use a component's automatically generated description as always up-to-date API reference information.

In addition, you can use the `cfcToMCDL` and `cfcToHTML` component methods of `utils.cfc`, which is located in the `[webroot]\CFIDE\componentutils` directory.

You can also browse the components available in ColdFusion using the Component Browser, which is located at `[webroot]\CFIDE\componentutils\componentdoc.cfm`.

The following figure shows the Component browser:



Note: To access the Component Browser in a virtual directory, you must add the virtual directory to the ColdFusion mappings.

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 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?](#) 244
- [Before you begin developing CFX tags in Java](#) 245
- [Writing a Java CFX tag](#) 247
- [ZipBrowser example](#) 251
- [Approaches to debugging Java CFX tags](#) 253
- [Developing CFX tags in C++](#) 256

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 provides several different techniques to create reusable code, including custom tags. For information on all of these techniques, see [Chapter 8, “Reusing Code in ColdFusion Pages” on page 157](#).

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) v 1.3.1 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, such as Dreamweaver MX, that provide an integrated environment for development, debugging, project management, and access to documentation.

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 the `lib` subdirectory of your ColdFusion installation directory. Consult your Java development tool documentation to determine how to configure the compiler classpath for your particular environment.

The lib directory created by the ColdFusion setup program serves two purposes:

- It contains the supporting classes required for developing and deploying Java CFX tags. This is the `com.allaire.cfx` package located in the `cfx.jar` archive.
- It supports a feature that reloads Java CFX tags located in the directory every time they are changed. Although this is not the default behavior for other Java classes, this behavior is very useful during an iterative development and testing cycle.

When you create new Java CFX tags, you should develop them in the `web_root/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 the ColdFusion embedded JVM. 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 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 Administrator’s online Help.

Writing a Java CFX tag

To create a Java CFX tag, create a class that implements the `CustomTag` 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_root/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\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 247](#).

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 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 Administrator:

- 1 On the ColdFusion Administrator Server tab, 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 Administrator (see [“Registering CFX tags” on page 257](#)).
- 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 Administrator:

- 1 On the ColdFusion Administrator Server tab, 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 `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 *CFML Reference*.

For a complete example Java CFX tag that uses `Request`, `Response`, and `Query` objects, see [“ZipBrowser example” on page 251](#).

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
<code>attributeExists</code>	Checks whether the attribute was passed to this tag.
<code>getAttribute</code>	Retrieves the value of the passed attribute.
<code>getIntAttribute</code>	Retrieves the value of the passed attribute as an integer.
<code>getAttributeList</code>	Retrieves a list of all attributes passed to the tag.
<code>getQuery</code>	Retrieves the query that was passed to this tag, if any.
<code>getSetting</code>	Retrieves the value of a global custom tag setting.
<code>debug</code>	Checks whether the tag contains the <code>debug</code> attribute.

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
<code>write</code>	Outputs text to the calling page.
<code>setVariable</code>	Sets a variable in the calling page.
<code>addQuery</code>	Adds a query to the calling page.
<code>writeDebug</code>	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
<code>getName</code>	Retrieves the name of the query.
<code>getRowCount</code>	Retrieves the number of rows in the query.
<code>getColumnNames</code>	Retrieves the names of the query columns.
<code>getData</code>	Retrieves a data element from the query.

Method	Description
addRows	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*.

Loading Java CFX classes

Each Java CFX class has its own associated `ClassLoader` that loads it and any dependent classes also located in the `web_root/WEB-INF/classes` directory. When Java CFX classes are reloaded after a change, a new `ClassLoader` is associated with the freshly loaded class. This special behavior is similar to the way Java servlets are handled by the web server and other servlet engines, and is required in order to implement automatic class reloading.

However, this behavior can cause subtle problems when you are attempting to perform casts on instances of classes loaded from a different `ClassLoader`. The cast fails even though the objects are apparently of the same type. This is because the object was created from a different `ClassLoader` and therefore is not technically the same type.

To solve this problem, only perform casts to class or interface types that are loaded using the standard Java classpath, that is, classes not located in the `classes` directory. This works because classes loaded from outside the `classes` directory are always loaded using the system `ClassLoader`, and therefore, have a consistent runtime type.

Automatic class reloading

You can determine how the server treats changed Java CFX class files by specifying the `reload` attribute when you use a CFX tag in your ColdFusion page. The following table describes the allowable values for the `reload` attribute:

Value	Description
Auto	Automatically reload Java CFX and dependent classes within the <code>classes</code> directory whenever the CFX class file changes. Does not reload if a dependent class file changes but the CFX class file does not change.
Always	Always reload Java CFX and dependent classes within the <code>classes</code> directory. Ensures a class reload even if a dependent class changes, but the CFX class file does not change.
Never	Never reload Java CFX classes. Load them once per server lifetime.

The default value is `reload="Auto"`. This is appropriate for most applications. Use `reload="Always"` during the development process, when you must ensure that you always have the latest class files, even when only a dependent class changed. Use `reload="Never"` to increase performance, by omitting the check for changed classes.

Note: The `reload` attribute applies only to class files located in the `classes` directory. The ColdFusion server loads classes located on the Java classpath once per server lifetime. You must stop and restart ColdFusion Server to reload these classes.

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 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—ColdFusion Server. 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 Server 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 Server 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="0n"` 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="0n">
```

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 *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\lib\cfx.jar` and `cf_root\runtime\lib\jrun.jar` 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:

- `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 ) 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. 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 NT only) Lets you add and delete Windows NT users.

On Windows NT, these tags are located in the `\cfusion\cfx\examples` directory. On UNIX, these tags are in `cf_root/coldfusion/cfx/examples`.

Setting up your C++ development environment

The following compilers 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
HPUX 11	HP aCC C++ 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 `\cfusion\cfx\include`. On UNIX this directory is `/opt/coldfusion/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++ class `CCFXRequest`. 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 *CFML Reference*.

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 on 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++ 6.0.
- 4 Select **Build > Start Debug > AttachProcess**.
- 5 Select `jruncvc.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 Administrator.

To register a C++ CFX tag:

- 1 On the ColdFusion Administrator Server tab, 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 On the ColdFusion Administrator Server tab, 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:

Designing and Optimizing a ColdFusion Application	261
Handling Errors	281
Using Persistent Data and Locking.....	315
Securing Applications	347
Developing Globalized Applications	373
Debugging and Troubleshooting Applications	389

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.cfm file. It also describes coding methods for optimizing application efficiency.

Contents

- [About applications.....](#) 262
- [Elements of a ColdFusion application.....](#) 262
- [Mapping an application.....](#) 265
- [Creating the Application.cfm page.....](#) 268
- [Optimizing ColdFusion applications](#) 272

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.

However, an application has a specific meaning in ColdFusion. A ColdFusion application consists of one or more ColdFusion pages that work together and share a common set of resources. In particular, the application shares an application name as specified in a `cfapplication` tag, and all pages in the application share variables in the Application scope. What appears to a user to be a single application, for example, a company's website, might consist of multiple ColdFusion applications.

While there are no definite rules as to 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 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
- Application-level settings and functions
- Reusable application elements
- Shared variables
- 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 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 <http://www.fusebox.org>.)

This chapter does not provide information on how to use or develop a specific application framework. However, it does discuss 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 [“Mapping an application” on page 265](#).

Note: For one example of an application framework, see “ColdFusion Methodologies for Content Management”, available at <http://www.macromedia.com/v1/handlers/index.cfm?ID=20750&method=full>.

Application-level settings and functions

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 provides a good place to define the application. It can contain the `cfapplication` tag that specifies the application name, and contains code that must be processed for all pages in the application. This page defines application-level settings, functions, and features.

Application-level features can include page processing settings, default variables, data sources, style settings, and other application-level constants, and application-specific custom error pages. When defined and set on the `Application.cfm` page, they are available on all pages in the application.

ColdFusion applications can have application-level variables that are not in the Application scope. For example, every page in an application might have a `currentPage` variable that identifies the page. The `Application.cfm` page can set this variable in the Variables scope, so each page gets a different, local value. Because every page in the application has the variable, it can be considered to be an application-level variable, even though it is not an Application scope variable.

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.

For more information on the `Application.cfm` and `OnRequestEnd.cfm` pages, see [“Creating the Application.cfm page” on page 268](#). For information on placing these pages in the application directory structure, see [“Mapping an application” on page 265](#).

Note: You can create a ColdFusion application without using `Application.cfm` or `OnRequestEnd.cfm` pages. 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.

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
- 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, “Reusing Code in ColdFusion Pages”](#) on page 157.

Shared variables

The following ColdFusion variable scopes maintain data that lasts beyond the scope of the current HTTP request:

Variable scope	Description
Session	Variables that are available for a single client browser for a single browser session in one application.
Client	Variables that are available for a single client browser over multiple browser sessions in one application.
Application	Variables that are available to all pages in an application for all clients.
Server	Variables that are available to all applications on a server and all clients.

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 315.

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”](#).

Mapping an application

When you design a ColdFusion application, you must map 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 you determine where to place your application pages and the `Application.cfm` and `OnRequestEnd.cfm` pages in a directory structure. For more information on how to define and use the `Application.cfm` page, see [“Creating the Application.cfm page” on page 268](#).

Processing the `Application.cfm` and `OnRequestEnd.cfm` pages

ColdFusion uses similar, but different, rules to locate and process the `Application.cfm` and `OnRequestEnd.cfm` pages.

Processing the `Application.cfm` page

When ColdFusion receives a request for an application page, it searches the page's directory for a file named `Application.cfm`. If one exists, the `Application.cfm` code is logically included at the beginning of that application page.

If the application page directory does not have an `Application.cfm` page, ColdFusion searches up the directory tree until it finds an `Application.cfm` page. If several directories in the directory tree have an `Application.cfm` page, ColdFusion uses the first page it finds. If the `Application.cfm` page is present in the directory tree (and has the required permissions set), you cannot prevent ColdFusion from including it.

ColdFusion processes only one `Application.cfm` page for each request. If a ColdFusion page has a `cfinclude` tag pointing to an additional ColdFusion page, ColdFusion does not search for an `Application.cfm` page when it includes the additional page.

If your application runs on a UNIX platform, which is case-sensitive, you must spell `Application.cfm` with an initial capital letter.

Processing the `OnRequestEnd.cfm` page

Just as the `Application.cfm` page runs before the code on an application page, an `OnRequestEnd.cfm` page runs, if it exists, after each application page in the same application.

The `OnRequestEnd.cfm` page must be 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.

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.

On UNIX systems, you must spell the `OnRequestEnd.cfm` file with the capital letters shown.

Defining the directory structure

Defining an application directory structure with an application-specific root directory has the following advantages:

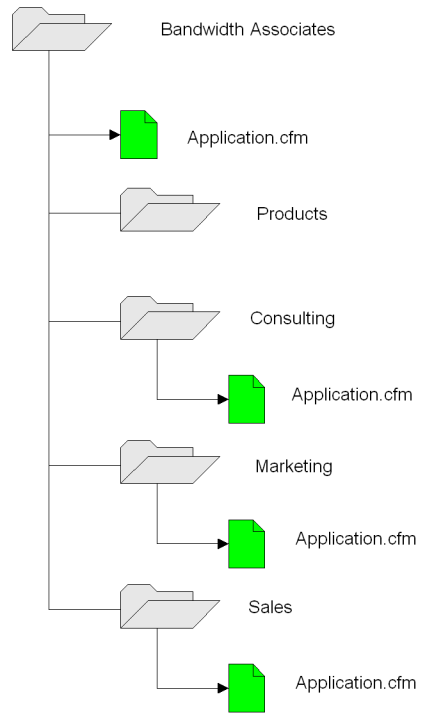
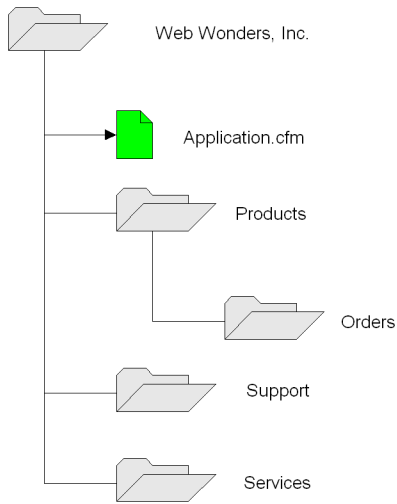
- **Development** The application is easier to develop and maintain, because the application page files are well-organized.
- **Portability** You can easily move the application to another server or another part of a server without changing any code in the application page files.
- **Application-level 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.cfm` page in the application root directory, or put different `Application.cfm` 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.cfm` pages with different application names. Alternatively, you can use multiple `Application.cfm` pages that specify the same application name, but have different common 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.cfm` 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.cfm` files to create individual ColdFusion applications at the departmental level. Only the Products application pages are processed using the settings in the root `Application.cfm` file. The Consulting, Marketing, and Sales directories each have their own `Application.cfm` file.



Creating the Application.cfm page

The Application.cfm page defines application-level settings and functions such as the following:

- Application name
- Client, application, and session variable management options
- Page processing settings
- Default variables, data sources, style settings, and other application-level constants
- Login processing
- Application-specific error handling

Naming the application

In ColdFusion, you define an application by giving it a name using the `cfapplication` tag. By using a specific application name in a `cfapplication` tag, you 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 you set for the name attribute in the `cfapplication` tag is limited to 64 characters.

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. For more information on using unnamed applications, see [Chapter 32, “Integrating J2EE and Java Elements in CFML Applications” on page 759](#).

Setting the client, application, and session variables options

You 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 `sessionManagement=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 Administrator.
- Specify a storage method for Client scope variables. This setting overrides the method set in the ColdFusion 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 315](#) and *CFML Reference*.

Defining page processing settings

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
<code>showDebugOutput</code>	Specifies whether to show debugging output. This setting cannot enable debugging if it is disabled in the ColdFusion Administrator. However, this option can ensure that debugging output is not displayed, even if the Administrator enables it.
<code>requestTimeout</code>	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 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.
<code>enableCFOutputOnly</code>	Disables output of text that is not included inside <code>cfoutput</code> 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.cfm`. For example, you might use it in multi-application environment to override the ColdFusion Administrator settings in one application.

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 347](#).

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.

You can also use the `Application.cfm` page to develop more sophisticated application-wide error-handling techniques, including error-handling methods that provide specific messages or use structured error-handling techniques.

For more information on error pages and error handling, see [Chapter 14, “Handling Errors”](#) on page 281.

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 347.

```
<!-- 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 readonly 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
<pre> <cfapplication name="Products" clientmanagement="Yes" clientstorage="myCompany" sessionmanagement="Yes"> </pre>	Names the application, enables Client and Session scope variables, and sets the client variable store to the myCompany data source.
<pre> <cfsetting showDebugOutput="No" > </pre>	Ensure that debugging output is not displayed, if the ColdFusion MX Administrator enables it.
<pre> <cferror type="request" template="requesterr.cfm" mailto="admin@company.com"> <cferror type="validation" template="validationerr.cfm"> </pre>	Specifies custom error handlers for request and validation errors encountered in the application. Specifies the mailing address for use in the request error handler.
<pre> <cfset app_is_initialized = False> . . . </pre>	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 315.

Code	Description
<pre><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></pre>	<p>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.</p>
<pre><cfset mainpage = "default.cfm"> <cfset current_page = "#cgi.path_info#?#cgi.query_string#"></pre>	<p>Sets two Variables scope variables that are used throughout the application. Creates the current_page variable dynamically; it's value varies from request to request.</p>
<pre><cfinclude template= "commonfiles/productudfs.cfm"></pre>	<p>Includes a library of user-defined functions that are used in most pages in the application.</p>

Optimizing ColdFusion applications

You can optimize your ColdFusion application in many ways. Much of optimizing ColdFusion 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 book 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 Administrator online Help 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 389](#).

For additional information on optimizing ColdFusion, see the Macromedia ColdFusion support center at <http://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 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 each URL to be 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 above 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 indicating 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 above 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 your `Application.cfm` 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" expireURL="HR*">
```

If you have a ColdFusion page that updates data 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 parts 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 in 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 case 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 Applicaiton 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
<pre><cfoutput> Welcome to our home page.
 The time is #TimeFormat(Now())#.
 Your lucky number is: #RandRange(1,1000)#
 <hr>
 </cfoutput></pre>	Displays the part of the page that must change each time.
<pre><cflock scope="application" timeout="20" type="readonly"> <cfset IsCached = IsDefined ("Application.ProductCache")> </cflock></pre>	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.
<pre><cfif not IsCached> <cfsavecontent variable="ProductCache"></pre>	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.
<pre><cfquery dataSource="ProductInfo" name="specialQuery"> SELECT ItemName, Item_Link, Description, BasePrice FROM SaleProducts </cfquery></pre>	Queries the database to get the necessary information
<pre><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></pre>	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.
<pre></cfsavecontent></pre>	Ends the cfsavecontent tag block.
<pre><cflock scope="Application" type="Exclusive" timeout=30> <cfset Application.productCache = productcache> </cflock></pre>	Inside an Exclusive cflock tag, saves the contents of the local variable ProductCache in the Application scope variable Application.productCache.
<pre></cfif></pre>	Ends the code that executes only if the Application.productCache variable does not exist.
<pre><cflock scope="application" timeout="20" type="readonly"> <cfoutput>#Application.ProductCache#</ cfoutput> </cflock></pre>	Inside a cflock tag, displays the contents of the Application.productCache variable.

Optimizing database use

Two important ColdFusion MX tools for optimizing your use of databases are the `cfstoredproc` tag and the `cfquery` tag cachedWithin attribute.

Note: 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.

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 = F00 dbVarName = @param2>
<!-- Close the cfstoredproc tag -->
</cfstoredproc>

<cfoutput>
    The output param value: '#foo#'  
</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
<pre> <cfstoredproc procedure = "foo_proc" dataSource = "MY_SYBASE_TEST" username = "sa" password = "" returnCode = "Yes"> </pre>	<p>Runs the stored procedure <code>foo_proc</code> on the <code>MY_SYBASE_TEST</code> data source. Populates the <code>cfstoredproc.statusCode</code> variable with the status code returned by stored procedure.</p>
<pre> <cfproccresult name = RS1> <cfproccresult name = RS3 resultSet = 3> </pre>	<p>Gets two record sets from the stored procedure: the first and third result sets it returns.</p>
<pre> <cfproccparam type = "IN" CFSQLType = CF_SQL_INTEGER value = "1" dbVarName = @param1> <cfproccparam type = "OUT" CFSQLType = CF_SQL_DATE variable = FOO dbVarName = @param2> </cfstoredproc> </pre>	<p>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 <code>FOO</code>.</p> <p>Ends the <code>cfstoredproc</code> tag body.</p>

Code	Description
<pre><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></pre>	<p>Displays the results of running the stored procedure:</p> <ul style="list-style-type: none"> • 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 `CompanyInfo` data source for one hour.

```
<cfquery datasource="CompanyInfo" 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, "Retrieving and Formatting Data"](#) on page 579.

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 <http://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

ColdFusion 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, “Retrieving and Formatting Data” on page 579](#) and [Chapter 27, “Building Dynamic Forms” on page 607](#). For information on debugging, see [Chapter 18, “Debugging and Troubleshooting Applications” on page 389](#).

Contents

- [About error handling in ColdFusion..... 282](#)
- [Understanding errors 283](#)
- [Error messages and the standard error format 289](#)
- [Determining error-handling strategies 291](#)
- [Specifying custom error messages with cferror 293](#)
- [Logging errors with the cflog tag..... 297](#)
- [Handling runtime exceptions with ColdFusion tags 299](#)

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 ColdFusion MX Administrator Server Settings page. For more information on specifying custom error pages in the Administrator, see the Administrator Help.

- Use the `cferror` tag to specify ColdFusion pages to handle specific types of errors.
- Log errors. ColdFusion logs certain errors by default. You can use the `cflog` tag to log other 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.

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 [“Handling errors in UDFs,” in Chapter 9](#).

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 389 . 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 <code>cfinclude</code> tags or custom tag calls that cannot find their targets.
Form field data validation	User data that does not meet the server-side form field validation rules in a form being submitted. You specify server-side form validation by using hidden HTML form fields. All other types of server-side validation, such as the <code>cfparam</code> tag generate runtime exceptions. For more information on validating form fields see Chapter 27, “Building Dynamic Forms” on page 607 .

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”](#)

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 the ColdFusion Server

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 <code>cfinclude</code> , <code>cfmodule</code> , and <code>cferror</code> tags are missing. (A <code>cferror</code> tag generates a <code>missingInclude</code> error only when an error of the type specified in the tag occurs.) The <code>MissingInclude</code> error type is a subcategory of <code>Template</code> error. If you do not specifically handle the <code>MissingInclude</code> error type, but do handle the <code>Template</code> error type, the <code>Template</code> error handler catches these errors. <code>MissingInclude</code> errors are caught at runtime.
Template errors	Template	General application page errors, including invalid tag and attribute names. Most <code>Template</code> 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 <code>cflock</code> critical section times out or fails at runtime.

Type	Type name	Description
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 *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 describes 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 server-side (hidden form field) data validation, and a `cferror` tag in the `Application.cfm` page specifies a Validation error handler, ColdFusion displays the specified error page. Otherwise, it displays the error information in a standard format that consists of a default header, a bulleted list describing the error(s), and a default footer. For more information on using hidden form field validation, see [Chapter 26, “Validating form field data types” on page 603](#).

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, ColdFusion handles the error as follows:
 - If a `cferror` tag on the `Application.cfm` page specifies an error handler for the exception type, ColdFusion displays the specified error page.
 - If 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 289](#).

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 <code>cftry</code> tag and the exception type is specified in a <code>cfcatch</code> tag.	Executes the code in the <code>cfcatch</code> tag. If the <code>cftry</code> block does not have a <code>cfcatch</code> tag for this error, tests for an appropriate <code>cferror</code> handler or site-wide error handler.
A <code>cferror</code> tag specifies an exception error handler for the exception type.	Uses the error page specified by the <code>cferror</code> 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 <code>cferror</code> tag specifies a Request error handler.	Uses the error page specified by the <code>cferror</code> 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, 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, such as the one shown in the following figure:

Error Occurred While Processing Request

Context validation error for tag cftry.

The start tag must have a matching end tag. This could be because CFCATCH is not the last tag nested in the CFTRY. CFCATCH must be the last tag inside a CFTRY.

The Error Occurred in
C:\CFusionMX\wwwroot\MYStuff\NeoDocs\cftryexample.cfm:
line 17

```
15 : <cfset EmpID=3>
16 : <cfparam name="errorCaught" default="">
17 : <cftry>
18 :     <cfquery name="test" datasource="CompanyInfo">
19 :         SELECT Dept_ID, FirstName, LastName, glog
```

Please Try The Following:

- [Check the beta forums](#), or [post a message](#).
- [Submit](#) a bug against Neo.
- Check the [CFML Reference Manual](#) to verify that you are using the correct syntax.
- Search the [Knowledge Base](#) to find a solution to your problem.

Browser Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 4.0; T312461)
Remote Address 127.0.0.1
Referer
Date/Time 06-Apr-02 04:23 PM
[Stack Trace \(click to expand\)](#)

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 297.)

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.

Section	Description
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: <ul style="list-style-type: none">• 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 287](#). 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 on the Application.cfm page to specify your own validation error handler. The section [“Example of a validation error page,” in Chapter 14](#) provides an example of such a page.

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 389](#) 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	<p>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.</p> <p>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.</p> <p>For more information, see “Handling runtime exceptions with ColdFusion tags” on page 299.</p>
cferror with exception-specific error handler pages	<p>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.</p> <p>For more information, see “Specifying custom error messages with cferror” on page 293.</p>
cferror with a Request error page	<p>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.cfm page to make it apply to all pages in an application.</p> <p>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.</p> <p>For example code, see “Example of a request error page” on page 296.</p>
Site-wide error handler page	<p>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.</p> <p>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.</p> <p>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.</p>

Specifying custom error messages with cferror

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 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 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.cfm file also. For more information on using the Application.cfm page, see [Chapter 13, “Designing and Optimizing a ColdFusion Application” on page 261](#).

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 284 .
Template	The ColdFusion page to display.
MailTo	(Optional) An e-mail address. The <code>cferror</code> tag sets the error page <code>error.mailTo</code> variable to this value. The error page can use the <code>error.mailTo</code> 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 = "servern@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	<ul style="list-style-type: none">• Cannot use CFML tags.• Can use HTML tags.• Can use the <code>Error.InvalidFields</code>, <code>Error.validationHeader</code>, and <code>Error.validationFooter</code> variables by enclosing them with pound sings (#).• Cannot use any other CFML variables.
Request	<ul style="list-style-type: none">• Cannot use CFML tags.• Can use HTML tags.• Can use nine CFML error variables, such as <code>Error.Diagnostics</code>, by enclosing them with pound sings.• Cannot use other CFML variables.
Exception	<ul style="list-style-type: none">• Can use full CFML syntax, including tags, functions, and variables.• Can use nine standard CFML Error variables and <code>cfcatch</code> variables. Use either <code>Error</code> or <code>cferror</code> as the prefix for both types of variables.• Can use other application-defined CFML variables.• To display any CFML variable, use the <code>cfoutput</code> tag.

The following table describes the variables available on error pages:

Error page type	Error variable	Description
Validation	<code>error.invalidFields</code>	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.
	<code>error.validationHeader</code>	Text for the header of the default validation message.
	<code>error.validationFooter</code>	Text for the footer of the default validation message.
Exception and Request	<code>error.browser</code>	Browser that was running when the error occurred.
	<code>error.dateTime</code>	Date and time when the error occurred.
	<code>error.diagnostics</code>	Detailed error diagnostics.
	<code>error.generatedContent</code>	Any content that ColdFusion generated in response to the request prior to the error.
	<code>error.HTTPReferer</code>	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.
	<code>error.mailTo</code>	E-mail address of the administrator who should be notified. This value is set in the <code>mailto</code> attribute of the <code>cferror</code> tag.
	<code>error.queryString</code>	URL query string of the client's request, if any.
	<code>error.remoteAddress</code>	IP address of the remote client.
	<code>error.template</code>	Page being executed when the error occurred.
Exception only	<code>error.messge</code>	Error message associated with the exception.
	<code>error.rootCause</code>	Java servlet exception reported by the JVM as the cause of the "root cause" of the exception. This variable is a Java object.
	<code>error.tagContext</code>	Array of structures structure 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 301
	<code>error.type</code>	Exception type. For more information, see "About ColdFusion exceptions" on page 284 .

Exception error pages can also use all of the exception variables listed in the section [“Exception information in cfcatch blocks” on page 301](#). 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 297](#).

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 *Administering ColdFusion MX*. It also provides the `cflog` tag which adds entries to ColdFusion logs.

ColdFusion automatically logs errors to the default logs in the following cases:

- If you use the default error handlers
- If a `cferror` handler of type Request handles the error

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#
      Diagnostcs: #error.diagnostics#">

</body>
</html>
```

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
<pre><cflog type="Error" file="myapp_errors" text="Exception error Exception type: #Error.type# Template: #Error.template#, Remote Address: #Error.remoteAddress#, Diagnostics: #Error.diagnostics#"></pre>	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:

```
"Error", "web-13", "12/19/01", "11:29:07", MYAPP, "Exception error --
  Exception type: coldfusion.runtime.CfErrorWrapper
  Template: /MYStuff/NeoDocs/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 by a cfapplication tag, normally in the 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 Administrator Site-wide Error Handler setting. 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
<code>cftry</code>	If any exceptions occur while processing the tag body, look for a <code>cfcatch</code> tag that handles the exception, and execute the code in the <code>cfcatch</code> tag body.
<code>cfcatch</code>	Execute code in the body of this tag if the exception caused by the code in the <code>cftry</code> tag body matches the exception type specified in this tag's attributes. Used in <code>cftry</code> tag bodies only.
<code>cfthrow</code>	Generate a user-specified exception.
<code>cfrethrow</code>	Exit the current <code>cfcatch</code> block and generates a new exception of the same type. Used only in <code>cfcatch</code> tag bodies.

Using `cftry` and `cfcatch` tags

The `cftry` tag allows you to 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 309](#).
- 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 the `cfcatch` structure.

Standard `cfcatch` variables

The following table describes the variables that are available in most `cfcatch` blocks:

Property variable	Description
<code>cfcatch.Detail</code>	<p>A detailed message from the CFML compiler. This message, which can contain HTML formatting, can help to determine which tag threw the exception.</p> <p>The <code>cfcatch.Detail</code> value is available in the CFScript <code>cfcatch</code> statement as the <code>exceptionVariable</code> parameter.</p>
<code>cfcatch.ErrorCode</code>	<p>The <code>cfthrow</code> tag can supply a value for this code through the <code>errorCode</code> attribute. For <code>Type="Database"</code>, <code>cfcatch.ErrorCode</code> has the same value as <code>cfcatch.SQLState</code>.</p> <p>Otherwise, the value of <code>cfcatch.ErrorCode</code> is the empty string.</p>
<code>cfcatch.ExtendedInfo</code>	<p>Custom error message information. This is returned only to <code>cfcatch</code> tags for which the <code>type</code> attribute is <code>Application</code> or a custom type.</p> <p>Otherwise, the value of <code>cfcatch.ExtendedInfo</code> is the empty string.</p>

Property variable	Description
<code>cfcatch.Message</code>	The exception's default diagnostic message, if one was provided. If no diagnostic message is available, this is an empty string. The <code>cfcatch.Message</code> value is included in the value of the CFScript catch statement <code>exceptionVariable</code> parameter.
<code>cfcatch.RootCause</code>	The Java servlet exception reported by the JVM as the cause of the "root cause" of the exception.
<code>cfcatch.TagContext</code>	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.
<code>cfcatch.Type</code>	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
<code>Column</code>	Obsolete (retained for backwards compatibility). Always 0.
<code>ID</code>	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 <code>cfmodule</code> .
<code>Line</code>	The line on the page in which the tag is located.
<code>Raw_Trace</code>	The raw Java stack trace for the error.
<code>Template</code>	The pathname of the application page that contains the tag.
<code>Type</code>	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
<code>cfcatch.NativeErrorCode</code>	<p>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 <code>cfcatch.NativeErrorCode</code> are driver-dependent.</p> <p>If no error code is provided, the value of <code>cfcatch.nativeErrorCode</code> is -1. The value is 0 for queries of queries.</p>
<code>cfcatch.SQLState</code>	<p>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.</p> <p>If the driver does not provide an SQLState value, the value of <code>cfcatch.SQLState</code> is -1.</p>
<code>cfcatch.Sql</code>	The SQL statement sent to the data source.
<code>cfcatch.queryError</code>	The error message as reported by the database driver.
<code>cfcatch.where</code>	If the query uses the <code>cfqueryparam</code> tag, query parameter name-value pairs.

Expression exceptions

The following variable is only available for Expression exceptions:

Property variable	Description
<code>cfcatch.ErrNumber</code>	An internal expression error number, valid only when <code>type="Expression"</code> .

Locking exceptions

The following additional information is available for exceptions related to errors that occur in `cflock` tags:

Property variable	Description
<code>cfcatch.lockName</code>	The name of the affected lock. This is set to "anonymous" if the lock name is unknown.
<code>cfcatch.lockOperation</code>	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
<code>cfcatch.missingFileName</code>	The name of the missing file.

Using `cftry`: an example

The following example shows the `cftry` and `cfcatch` tags. It uses the `CompanyInfo` data source used in many of the examples in this book 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="CompanyInfo">
        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>

```

Testing the code

Use the following procedure 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 back 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",,"C:\Neo\servers\default\logs\
MyAppPage.log initialized"
"Information","web-0","11/20/01","16:27:08",,"Page: /neo/MYStuff/NeoDocs/
cftryexample.cfm Error: MissingInclude"
"Information","web-1","11/20/01","16:27:32",,"Page: /neo/MYStuff/NeoDocs/
cftryexample.cfm Error: "
"Information","web-0","11/20/01","16:27:49",,"Page: /neo/MYStuff/NeoDocs/
cftryexample.cfm Error: Database"
"Information","web-1","11/20/01","16:28:21",,"Page: /neo/MYStuff/NeoDocs/
cftryexample.cfm Error: General Exception"
"Information","web-0","11/20/01","16:28:49",,"Page: /neo/MYStuff/NeoDocs/
cftryexample.cfm Error: "
```

Reviewing the code

The following table describes the code:

Code	Description
<pre><cfset EmpID=3> <cfparam name="errorCaught" default=""></pre>	<p>Initializes the employee ID to a valid value. An application would get the value from a form or other source.</p> <p>Sets the default errorCaught variable value to the empty string (to indicate no error was caught).</p> <p>There is no need to put these lines in a cftry block.</p>
<pre><cftry> <cfquery name="test" datasource="CompanyInfo"> SELECT Dept_ID, FirstName, LastName FROM Employee WHERE Emp_ID=#EmpID# </cfquery></pre>	<p>Starts the cftry block. Exceptions from here to the end of the block can be caught by cfcatch tags.</p> <p>Queries the CompanyInfo database to get the data for the employee identified by the EmpID variable.</p>

Code	Description
<pre> <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> </pre>	<p>Begins the HTML page. This section contains all the code that displays information if no errors occur.</p> <p>Includes the includeme.cfm page.</p> <p>Displays the user information record from the test query.</p>
<pre> <cfcatch type="MissingInclude"> <h1>Missing Include File</h1> <cfoutput> Message: #cfcatch.Message# Detail: #cfcatch.Detail# File name: #cfcatch.MissingFilename# </cfoutput> <cfset errorCaught = "MissingInclude"> </cfcatch> </pre>	<p>Handles exceptions thrown when a page specified by the cfinclude tag cannot be found.</p> <p>Displays cfcatch variables, including the ColdFusion basic error message, detail message, and the name of the file that could not be found.</p> <p>Sets the errorCaught variable to indicate the error type.</p>
<pre> <cfcatch type="Database"> <h1>Database Error</h1> <cfoutput> Message: #cfcatch.Message# Native error code: #cfcatch.NativeErrorCode# SQLState: #cfcatch.SQLState# Detail: #cfcatch.Detail# </cfoutput> <cfset errorCaught = "Database"> </cfcatch> </pre>	<p>Handles exceptions thrown when accessing a database.</p> <p>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.</p> <p>Sets the errorCaught variable to indicate the error type.</p>
<pre> <cfcatch type="Any"> <cfoutput> <hr> <h1>Other Error: #cfcatch.Type#</h1> Message: #cfcatch.message# Detail: #cfcatch.Detail# </cfoutput> <cfset errorCaught = "General Exception"> </cfcatch> </pre>	<p>Handles any other exceptions generated in the cftry block.</p> <p>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.</p> <p>Displays the ColdFusion basic and detailed error message.</p> <p>Sets the errorCaught variable to indicate the error type.</p>
<pre> </body> </html> </cftry> </pre>	<p>Ends the HTML page, then the cftry block.</p>

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
<code>type</code>	The type of error. It can be a custom type that has meaning only to your application, such as <code>InvalidProductCode</code> . You can also specify <code>Application</code> , the default type. You cannot use any of the predefined ColdFusion error types, such as <code>Database</code> or <code>MissingTemplate</code> .
<code>message</code>	A brief text message indicating the error.
<code>detail</code>	A more detailed text message describing the error.
<code>errorCode</code>	An error code that is meaningful to the application. This field is useful if the application uses numeric error codes.
<code>extendedInfo</code>	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. The ColdFusion Server 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 `cfthrow`

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 `rethrow` tag can only be used in a `cfcatch` tag body.

The `cfrethrow` tag syntax

The following pseudo-code 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 `rethrow` 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 `CompanyInfo` 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")>
    <cdump var="#getEmpsResult#">
</cfif>

```


Reviewing the code

The following table describes the code:

Code	Description
<pre><cftry> <cf_getEmps EmpName="Jones"></pre>	In a <code>cftry</code> block, calls the <code>cf_getEmps</code> custom tag (<code>getEmps.cfm</code>).
<pre><cfcatch type="myApp.getUser.noEmpName"> <h2>Oops</h2> <cfoutput>#cfcatch.Message#</cfoutput>
 </cfcatch> </cftry></pre>	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 <code>cfthrow</code> tag in the custom tag.
<pre><cfif isdefined("getEmpsResult")> <cfdump var="#getEmpsResult#"> </cfif></pre>	If the tag returns a result, uses the <code>cfdump</code> tag to display it. (A production application would not use <code>cfdump</code> .)

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", 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="CompanyInfo">
        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="CompanyInfoBackup">
```

```

        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
<pre><cfif NOT IsDefined("attributes.EmpName")> <cfthrow Type="myApp.getUser.noEmpName" message = "Last Name was not supplied to the cf_getEmps tag."> <cfexit method = "exittag"></pre>	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.
<pre><cfelse> <cftry></pre>	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.
<pre><cftry> <cfquery Name = "getUser" DataSource="CompanyInfo"> SELECT * FROM Employee WHERE LastName = '#attributes.EmpName#' </cfquery> <cfset caller.getEmpsResult = getuser></pre>	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.
<pre><cfcatch type= "Database"> <cfif (cfcatch.sqlstate IS "S100") OR (cfcatch.sqlstate IS "IM002")> <cftry> <cfquery Name = "getUser" DataSource= "CompanyInfoBackup" SELECT * FROM Employee WHERE LastName = '#attributes.EmpName#' </cfquery> <cfset caller.getEmpsResult = getuser></pre>	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.
<pre><cfcatch type = "Database" /></pre>	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.
<pre><cfcatch type = "Any"> <cfrethrow> </cfcatch> </cftry></pre>	If the second database query failed for any other reason, throws the error up to the next try block. Ends the innermost try block
<pre><cfelse> <cfrethrow> </cfif> </cfcatch></pre>	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.

Code	Description
<pre><cfcatch type = "Any"> <cfrethrow> </cfcatch> </cftry></pre>	<p>In the second try block, catches any errors other exceptions and rethrows them up to the outermost try block.</p> <p>Ends the second try block.</p>
<pre><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></pre>	<p>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 <code>cfcatch</code> tag handles only errors that are rethrown from the nested blocks.</p> <p>Exits the custom tag and returns to the calling page.</p> <p>Ends the catch block, try block, and initial <code>cfif</code> block.</p>

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, `MyAttrib`. 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 `CompanyInfo`.
- Insert `cfthrow` tags throwing custom exculpations in various places in the code and observe the effects.

CHAPTER 15

Using Persistent Data and Locking

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](#) 316
- [Managing the client state](#) 318
- [Configuring and using client variables](#) 323
- [Configuring and using session variables](#) 328
- [Configuring and using application variables](#) 333
- [Using server variables](#) 335
- [Locking code with cflock](#) 336
- [Examples of cflock](#) 343

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	<p>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 328.</p> <p>Useful for client-specific information, such as client preferences, that you want to store for a significant period of time.</p> <p>Data is stored as cookies, database entries, or Registry values. client variables can time out after an extended period.</p> <p>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.</p>
Session	<p>Contains variables that are available for a single client browser for a single browser session in an application.</p> <p>Useful for client-specific information, such as shopping cart contents, that you want to persist while the client is visiting your application.</p> <p>Data is stored in memory and times out after a period of inactivity or when the server shuts down.</p> <p>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 328</p> <p>You must use the Session scope prefix in the variable name.</p>
Application	<p>Contains variables that are available to all pages in an application for all clients.</p> <p>Useful for application-specific information, such as contact information, that can vary over time and should be stored in a variable.</p> <p>Data is stored in memory and times out after a period of inactivity or when the server shuts down.</p> <p>You must use the Application scope prefix in the variable name.</p>
Server	<p>Contains variables that are available to all applications in a server and all clients.</p> <p>Useful for information that applies to all pages on the server, such as an aggregate page-hit counter.</p> <p>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.</p> <p>You must use the Server scope prefix in the variable name.</p>

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 ensure data consistency, you must lock access to all code that changes variables in these scopes and all code that reads variables in these scopes with values that can change.

Additionally, you must be careful when using client variables in a server cluster, where an applications can run on multiple servers.

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.

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:

- Many clustering systems, including ClusterCats support “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, select the Purge Data for Clients that Remain Unvisited option in the ColdFusion MX Administrator Client Variables Add/Edit Client Store page on one server only.

For more information on using client and session variables in clustered systems, see [“Managing client identity information in a clustered environment” on page 322](#).

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 336](#).

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	<p>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.</p> <p>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.</p> <p>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.</p> <p>Client variables must be simple variables, such as numbers, dates, or strings. They cannot be arrays, structures, query objects, or other objects.</p> <p>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.</p> <p>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.</p> <p>You do not lock code that uses client variables.</p> <p>You can use client variables that are stored in cookies or a common database in clustered systems.</p>
Session	<p>Data is stored in memory so it is accessed quickly.</p> <p>Data is lost when the client browser is inactive for a time-out period. You specify the time-out in the ColdFusion Administrator and Application.cfm.</p> <p>As with client variables, data is available to a single client and application only.</p> <p>Variables can store any ColdFusion data type.</p> <p>You must prefix all variable names with the Session scope name.</p> <p>Lock code that uses session variables to ensure data integrity.</p> <p>You can use session variables in clustered systems only if the systems support “sticky” sessions, where a session is limited to a single server.</p>

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 320](#).

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 328](#).

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 do not require users to enable 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 the `setClientCookies` attribute of the `cfapplication` tag to `No`.

To use ColdFusion client or 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 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 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 must set the `cfapplication` tag `clientmanagement` attribute to `Yes` on every page. Because the `Application.cfm` file is included in all of the application's pages, you enable client management in the `cfapplication` tag, at the beginning of the `Application.cfm` file. For example, to enable client variables in an application named `SearchApp`, you 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 `clientStorage` attribute in the `cfapplication` tag.

You can specify the following values in the `clientStorage` attribute:

- Registry(default)
- Name of a data source configured in ColdFusion 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 `cfapplication` tag `clientstorage="Cookie"` attribute, 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:

- 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 non alphanumeric 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, 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, 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 the Application.cfm file

The `cfapplication` tag `clientStorage` attribute lets you override the default client variable storage application location. The following line tells ColdFusion to store the client variables in the `mydatasource` data source:

```
<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 [“Using WDDX,” in Chapter 30](#).

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	A combination of the CFID and CFToken values, in the form <code>CFID=IDNum&CFTOKEN=tokenNum</code> . This variable is useful if the client does not support cookies and you must pass the CFID and CFToken variables from page to page.
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 320](#).

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 of ColdFusion 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: To preserve data integrity, put code that uses session variables inside `cflock` tags. For information on using `cflock` tags see ["Locking code with cflock" on page 336](#).

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 of ColdFusion Administrator Server tab.

You can also set the time-out period for session variables inside a specific application (thereby overruling the Administrator default setting) by using the `cfapplication` tag `sessionTimeout` attribute. However, you cannot use the `cfapplication` tag to set a time-out value that is greater than the maximum session time-out value set on the Administrator Memory Variables page.

Your application can also manually end a session, for example, when a user logs out.

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. When you use ColdFusion session management, session variables are not available to JSP pages or Java servlets that you call from your ColdFusion pages.

J2EE session management uses a session-specific session identifier, `jsessionid`, which is created afresh at the start of each session. With J2EE session management, you can share session variables between ColdFusion pages and JSP pages or Java servlets that you call from the ColdFusion pages. 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.

Configuring and enabling session variables

To use session variables, you must enable them in both of the following places:

- ColdFusion MX Administrator
- The active `cfapplication` tag

ColdFusion Administrator and the `cfapplication` tag also provide facilities for configuring session variable behavior, including the variable time-out.

Selecting and enabling session variables in ColdFusion MS 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. The `cfapplication` tag can override this value. The default value for this time-out is 20 minutes.
- Specify a maximum session time-out. The `cfapplication` tag 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 `cfapplication` tag in your `Application.cfm` file. 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 320](#). 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 <code>CFID=IDNum&CFTOKEN=tokenNum</code> . 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: the string <code>jsessionid=</code> followed by the J2EE session ID.
Session.SessionID	A unique identifier for the session. ColdFusion session management: the application name and CFID and CFToken values. J2EE session management: the <code>jsessionid</code> 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 <http://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

If you use J2EE session management, the session and all session variables are deleted, when the user closes the browser.

If you use ColdFusion session management and do not explicitly terminate a session, for example when a user logs out, the session variables remain in ColdFusion server memory until the session time-out period elapses. If you store sensitive data, such as personally identifiable user information, in session variables, you should clear the data from the Session structure when the user logs out.

You can expire a session by using the `cfapplication` tag and setting the `sessiontimeout` attribute 0. The `cfapplication` tag must also reset all other attributes to the desired settings, as follows:

```
<cfapplication
  name="MyApp"
  clientmanagement="Yes"
  applicationtimeout="#CreateTimeSpan(0, 0, 0, 30)#"
  sessionmanagement="Yes"
  sessiontimeout="#CreateTimeSpan(0, 0, 0, 0)#" >
```

To save processing time, ColdFusion deletes expired session variables every 10 seconds. As a result, the session might continue to exist for up to 10 seconds after ColdFusion executes your code.

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: To preserve data integrity, put code that uses application variables inside `cflock` tags. For information on using `cflock` tags see [“Locking code with cflock” on page 336](#).

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 in the `cfapplication` tag for the current page.

Note: ColdFusion supports unnamed applications for compatibility with J2EE applications. For more information see [“Unnamed ColdFusion Application and Session scopes,” in Chapter 32](#).

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 `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 `cfapplication` tag 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 `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.

To preserve data integrity, you must put all 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 331](#) and [“Accessing and changing session variables” on page 331](#).

For information on locking write-once read-many application variables efficiently, see [“Locking application variables efficiently” on page 342](#)

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: To preserve data integrity, put code that uses server variables inside `cflock` tags. 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
<code>Server.ColdFusion.AppServer</code>	The name of the J2EE application server ColdFusion is using. For ColdFusion MX Server editions, which have an integrated application server, the name is <code>JRun4</code> .
<code>Server.ColdFusion.Expiration</code>	The date, in ODBC date format, on which the ColdFusion Server license expires. (A null string in all but trial versions of ColdFusion.)
<code>Server.ColdFusion.ProductLevel</code>	The server product level, such as Enterprise.
<code>Server.ColdFusion.ProductName</code>	The name of the product (ColdFusion Server).
<code>Server.ColdFusion.ProductVersion</code>	The version number for the server that is running, such as 6,0,0.
<code>Server.ColdFusion.Rootdir</code>	Directory under which ColdFusion is installed, such as <code>C:\cfusion</code> .
<code>Server.ColdFusion.SerialNumber</code>	The serial number assigned to this server installation.
<code>Server.ColdFusion.SupportedLocales</code>	The locales, such as English (US) and Spanish (Standard), supported by the server.
<code>Server.OS.AdditionalInformation</code>	Additional information provided by the operating system, such as the Service Pack number.
<code>Server.OS.arch</code>	The processor architecture, such as x86 for Intel Pentium processors.
<code>Server.OS.BuildNumber</code>	The specific operating system build, such as 1381
<code>Server.OS.Name</code>	The name of the operating system, such as Windows NT.
<code>Server.OS.Version</code>	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 Server 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.

While the ColdFusion Server 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, the ColdFusion Server 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 the ColdFusion Server 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 ensure data integrity.

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.totalTicketsSold` 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 343](#).

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.

However, although leaving code that uses write-once data unlocked can improve application performance, it also has risks. You must make sure that the variables are truly written only once. For example, you must make sure 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.

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 manages 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 be 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 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 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` block, 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:

```
<!-- Inititalize 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.variable1 = 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 turtlesnecks 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 turtlesnecks sold -->

<cflock scope="Application"
    timeout="10"
    type ="ReadOnly">
    <cfoutput>
        E-Turtlesneck is proud to say that we have sold
        #Application.number# turtlesnecks 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-Turtlesneck.
    Today you have chosen a turtlesneck 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 turtles 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 turtle neck
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>
<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

ColdFusion MX has two major security features: resource (file and directory-based) security and user (programmatic) security. This chapter provides an overview of ColdFusion security. It briefly describes how you use the ColdFusion MX Administrator to configure resource security, and discusses structuring an application to take advantage of resource security. 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 example, the chapter on LDAP discusses secure access to LDAP directories. Similarly, the section “[Enhancing security with cfqueryparam](#),” in [Chapter 20](#) describes a method for preventing inappropriate access to SQL databases. See the Security entries in the Index for a complete listing of such sections.

For detailed information on using Administrator-controlled security features, see *Administering ColdFusion MX*.

This chapter does not discuss general or web server security concepts and issues. 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](#) 348
- [About resource security](#) 349
- [About user security](#) 351
- [Implementing user security](#) 360

ColdFusion security features

ColdFusion provides scalable, granular security for building and deploying your ColdFusion applications. ColdFusion provides 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. This chapter does not these passwords. For more information see the Administrator Help.
- **Resource** The ColdFusion MX Administrator can limit access to ColdFusion resources, including selected tags and functions, data sources, files, and host addresses, based on the location of your ColdFusion pages. 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 role or ID and selectively determine what to do based on this information.

Note: You can also use the `cfencode` utility, located in the `cf_root/bin` directory, to obfuscate ColdFusion pages that you distribute. Although this technique cannot prevent determined hackers from determining the contents of your pages, it does prevent inspection of the pages.

About resource security

Resource security lets you secure access to ColdFusion resources based on the ColdFusion page location, by applying a set of access rules to all ColdFusion pages in a directory. The directory or directories to which a set of rules apply is called a **sandbox**, and resource security is also called sandbox security. The ColdFusion Administrator Security Settings page enables resource security; the Sandbox Security page configures access to resources. Resource security controls 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, cfexecute, cffile, cfftp, cfgridupdate, cfhttp, cfhttpparam, cfindex, cfinsert, cfinvoke, cfdap, cflog, cfmail, cfobject, cfobjectcache, cfquery, cfregistry, 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, DirectoryExists, ExpandPath, FileExists, GetBaseTemplatePath, GetDirectoryFromPath, GetFileFromPath, GetProfileString, GetTempDirectory, GetTemplatePath, SetProfileString
Files/Directories	Sets read, write, execute, and delete access to specified directories, directory trees, or files.
Server/Ports	Controls access to IP addresses and port numbers. You can specify host names or numeric addresses, and you can specify individual ports and port ranges.

By default, resource security rules apply to the specified directory and all its subdirectories. If you create a set of rules for a subdirectory of another sandbox, the subdirectory's rules override the parent directory's rules.

Resource 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 allow each application to access only its own data sources and files.

Resource security also lets you to structure and partition an application to reflect the access rights that are appropriate to different functional components. For example, if your application has both user functions and administrator functions, you could structure the application as follows:

- Administrator pages go in one directory with access rules that enable most activities.
- User pages go in another directory whose rules limit the files they can modify and the tags they can use.
- Pages required for both administrative and user functions go in a third directory with appropriate access rules.

For more information on configuring resource security, see *Administering ColdFusion MX*.

About user security

User security lets your application use security rules to determine what it displays. It has two elements:

- Authentication
- Authorization

Authentication ensures that a valid user is logged in, based on an ID and password provided by the user. ColdFusion 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:

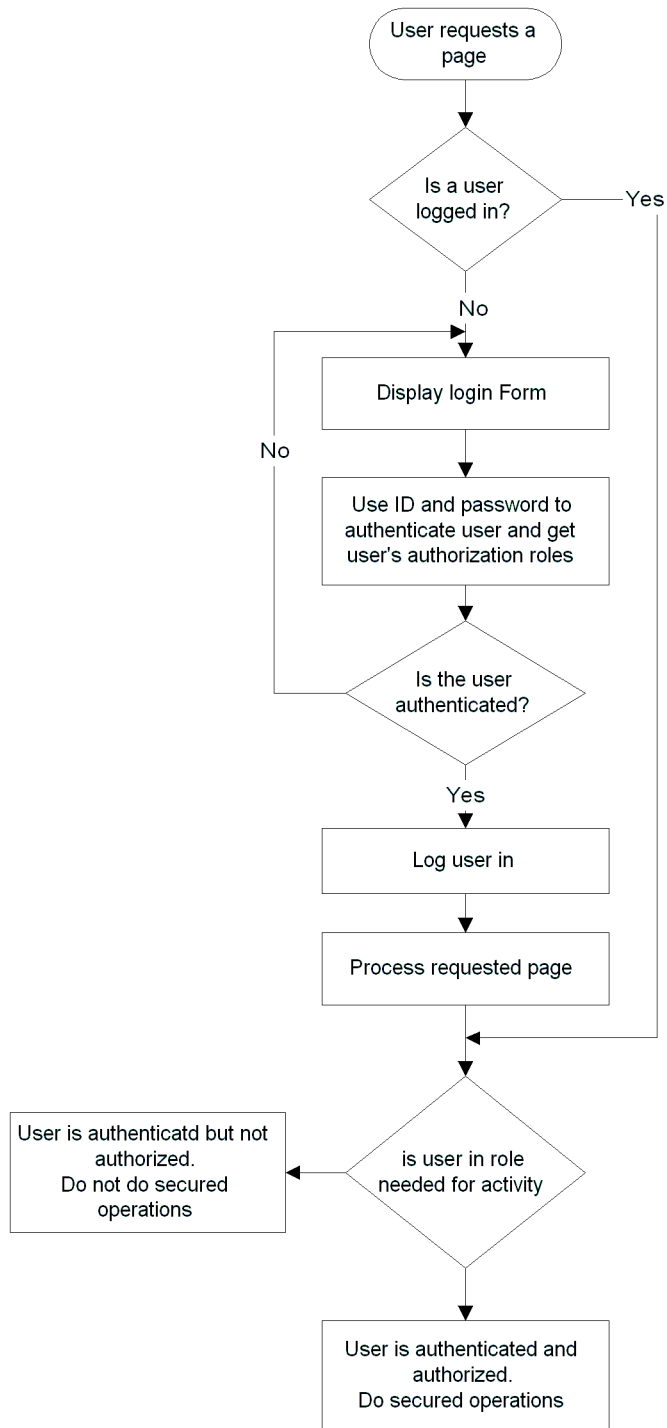
- Employees
- Human Resources
- Benefits
- Managers

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 at log 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.



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 there is no logged-in user. When using application-based security, you put code in the body of the <code>cflogin</code> to check the user-provided ID and password against a data source, LDAP directory, or other repository of login identification. The body includes a <code>cfloginuser</code> tag (or a ColdFusion page that contains a <code>cfloginuser</code> tag) to establish the authenticated user's identity in ColdFusion.
cfloginuser	Identifies the authenticated user to ColdFusion. Specifies the user's ID, password, and roles. Typically used inside <code>cflogin</code> tags. The <code>cfloginuser</code> tag requires three attributes, <code>name</code> , <code>password</code> , and <code>roles</code> , and does not have a body. The <code>roles</code> 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.
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 when the session ends. The <code>cflogout</code> tag does not take any attributes, and does not have a body.
cffunction	Used only in <code>cfcomponent</code> tags. If you include a <code>roles</code> 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 current logged in user.

About web server authentication and application authentication

ColdFusion supports two forms of user authentication:

- Web server basic authentication
- Application authentication

About web server basic authentication

All major web servers support basic authentication, also known as basic HTTP authentication. The web server requires the user to log in to access pages in a particular directory, as follows:

- 1 When the user first request a page in the secured directory, the web server presents the user with a login page
- 2 The user fills in the login page and submits it
- 3 The web server checks the user's login ID and password using its own user authentication mechanism.

- 4 If the user logs in successfully, the browser caches the authentication information and sends it with every subsequent page request from the user.
- 5 The web server processes the requested page and all future page requests from the browser that contain the cached login information, if it is valid for the requested page.

The application can perform additional authorization checks based on the information provided by the browser with each request. For example, the application can determine the user's roles based on the login ID and check the roles against specific roles required to access pages or sections of code within pages.

You can use basic web server authentication without using any ColdFusion security features. In this case, you only perform directory-based user authentication, and you configure all user security through the web server's interfaces. You do not use any of the ColdFusion security features, and typically do not perform role-based authorization.

You can also use basic web server authentication with ColdFusion security tags and functions to enforce user authorization. In this case you rely on the web server for user authentication, and your application does not have to display a login page. You then use the web server authentication information with the `cflogin` and `cfloginuser` tags to log the user into the ColdFusion user security system and use the `IsUserInRole` and `GetUserName` functions to ensure user authorization. For more information on this form of security, see [“A basic authentication security scenario” on page 356](#).

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 `GetUserName` functions to check the user's roles or identity before running a ColdFusion page or specific code on a page. For more information on application authentication, see [“An application authentication security scenario” on page 357](#).

Controlling ColdFusion login behavior

When you use the `cfloginuser` tag within a `cflogin` tag, ColdFusion stores a login token in a memory-only browser cookie. Therefore, to use the `cflogin` tag to check for an authenticated user, the user must enable memory-only cookies in the browser. The login cookie does not last after the user closes the browser.

You can use the `cfloginuser` tag without user cookies, but the login information remains in effect for only the current page. For more information on user logins without cookies, see [“Using ColdFusion security without cookies” on page 356](#).

The `cflogin` tag has three optional arguments that control the characteristics of a ColdFusion login, as follows:

Attribute	Use
<code>idleTimeout</code>	If no page requests occur during the <code>idleTimeout</code> period, the ColdFusion logs the user out. The default is 1800 seconds (30 minutes).
<code>applicationToken</code>	Limits the login validity to a specific application as specified by a ColdFusion page's <code>cfapplication</code> tag. The default value is the current application name.
<code>cookieDomain</code>	The Internet domain for which the ColdFusion security cookie is valid. By default, there are no domain limitations.

Logging a user out

After a user logs in, the ColdFusion user authorization and authentication information remains valid until any of the following happens:

- The login times out. This happens if the user does not request a new page for the `idleTimeout` period.
- The application uses a `cflogout` tag to log out the user, usually in response to the user clicking a logout link or button.
- The user closes the browser.

Specifying an applicationToken value

The login identification created by `cflogin` tag is valid only for pages within the directory that contains 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. In normal usage, there you do not need to specify a `applicationToken` value in the `cflogin` tag.

Limiting the valid internet domain

Use the `cookieDomain` attribute to limit the log-in capabilities to users from a specific domain or even a specific system. For example, to ensure that only users located in the `macromedia.com` domain can log in to your application, specify `cookieDomain=".macromedia.com"`. To specify a domain name, you start the name with a period.

The cflogin structure

The `cflogin` tag has a built-in `cflogin` structure that contains two variables: `cflogin.name` and `cflogin.password`. These variables contain the user ID and password in either of the following cases:

- The `cflogin` tag body is executing in response to a user logging in on the browser's basic login page.
- The `cflogin` tag body is executing in response to a user logging in on an application login form that contains input fields with the names `j_username` and `j_password`.

Therefore, the `cflogin` structure provides a consistent interface for determining the user's login ID and password independent of the technique you use for displaying the login form.

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 where you use ColdFusion login identification.

A basic authentication security scenario

An application that uses basic web server authentication might work as follows. The example in [“Basic authentication user security example” on page 360](#) 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 the user in. 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 runs the contents of the `Application.cfm` page before it runs the requested page. The `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 by a `cflogout` tag or the login has not timed out from inactivity.
- 4 Code in the `cflogin` tag body uses the user ID and password from browser login, contained in the `cflogin.name` and `cflogin.password` variables, as follows:
 - a First 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 The `cflogin` tag body code calls the `cfloginuser` tag with the user's ID, password, and roles to identify the user to ColdFusion.
- 5 Application.cfm completes processing and ColdFusion processes the requested application page.
- 6 The 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. For example, administrative pages
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.

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 362](#) implements this scenario.

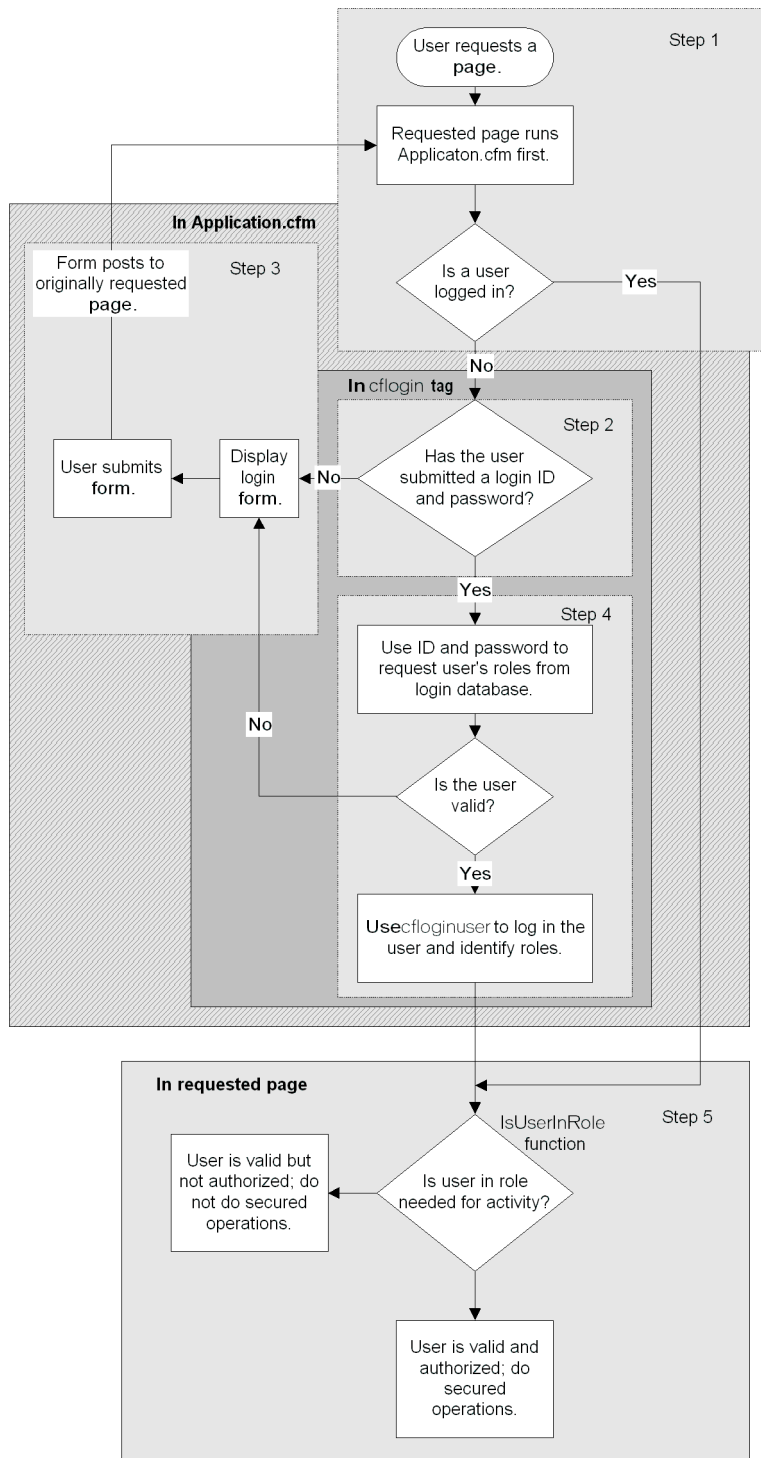
- 1 Whenever ColdFusion receives a request for a ColdFusion page, it runs the contents of the Application.cfm page before it runs the requested page. The Application.cfm page contains a `cflogin` tag. ColdFusion executes the `cflogin` tag body if the user is not logged in. A user is logged in if the `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 `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 `login` tag in Application.cfm 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 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 log-out 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 on the Application.cfm page.

Note: A log-out option is not always required, as the user is automatically logged out when the browser closes or is inactive for the time-out period. However, you can enhance security in cases where a system might be shared by providing a log-out facility. You must explicitly log out a user before a new user can log in using the same browser session.

While 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 containing 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.
- You must 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` to log the user into the ColdFusion session.

The following figure shows this flow of control. For simplicity, it omits the logout button.



Implementing user security

The following section provide several examples of ways to implement security using basic authentication and application authentication

Basic authentication user security example

The example in this section shows how you might implement user security using web-server basic authentication and two roles, user and administrator.

This example has two ColdFusion pages:

- The Application.cfm 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.

You can test the security behavior by adding your own pages to the same directory as the Application.cfm page.

Example: Application.cfm

The Application.cfm page consists of the following:

```
<cfapplication name="Orders">
<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>
```

Reviewing the code

The Application.cfm page executes before the code in each ColdFusion page in an application. For more information on the Application.cfm page and when it is executed, see [Chapter 13, “Designing and Optimizing a ColdFusion Application”](#) on page 261.

The following table describes the CFML code in Application.cfm and its function:

Code	Description
<code><cfapplication name="Orders"></code>	Identifies the application. The login information on this page only applies to this application.
<code><cflogin> <cfif IsDefined("cflogin")> <cfif cflogin.name eq "admin"> <cfset roles = "user,admin"> <cfelse> <cfset roles = "user"> </cfif></code>	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.
<code><cfloginuser name = "#cflogin.name#" password = "#cflogin.password#" roles = "#roles#" /></code>	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.
<code><cfelse> <!-- this should never happen ---> <h4>Authentication data is missing.</h4> Try to reload the page or contact the site administrator. <cfabort></code>	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.
<code></cfif> </cflogin></code>	Ends if/else block. Ends the cflogin tag body.

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.cfm 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 code:

```

<!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
<pre> <cfoutput> <h2>Welcome #GetAuthUser()#!</h2> </cfoutput> </pre>	Displays a welcome message that includes the user's login ID.
<pre> ALL Logged-in Users see this message.

 </pre>	Displays this message in all cases. The page does not display until a user is logged in.
<pre> <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> </pre>	<p>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.</p> <p>The user sees one message per role to which he or she belongs.</p>

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.cfm 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.cfm.
- 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.cfm page.

The example gets user information from the LoginInfo table of the CompanyInfo 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.

The following sections contain listings and descriptions of each of the pages.

Example: Application.cfm

The Application.cfm page consists of the following:

```
<cfapplication name="Orders" sessionmanagement="Yes">

<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="CompanyInfo">
                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=MyApp/index.cfm" method="Post">
            <input type="submit" Name="Logout" value="Logout">
        </form>
    </cfoutput>
</cfif>

```

Reviewing the code

The `Application.cfm` page executes before the code in each ColdFusion page in an application. For more information on the `Application.cfm` page and when it is executed, see [Chapter 13, “Designing and Optimizing a ColdFusion Application” on page 261](#). The following table describes the CFML code in `Application.cfm` and its function:

Code	Description
<pre> <cfapplication name="Orders" sessionmanagement="Yes"> </pre>	Identifies the application and enables Session scope variables.
<pre> <cfif IsDefined("Form.logout")> <cflogout> </cfif> </pre>	If the user just submitted the logout form, logs out the user. The following <code>cflogin</code> tag runs as a result.
<pre> <cflogin> <cfif NOT IsDefined("cflogin")> <cfinclude template="loginform.cfm"> <cfabort> </pre>	<p>Executes if there is no logged-in user.</p> <p>Tests to see if the user has submitted a login form. If not, uses <code>cfinclude</code> to display the form. Uses the built-in <code>cflogin</code> variable that contains the user name and password if it was submitted by the login form.</p> <p>The <code>cfabort</code> tag prevents processing of any code that follows on this page.</p>
<pre> <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> </pre>	<p>Executes if the user submitted a login form.</p> <p>Tests to make sure both name and password have data. If either variable is empty, displays a message, followed by the login form.</p> <p>The <code>cfabort</code> tag prevents processing of any code that follows on this page.</p>

Code	Description
<pre><cfelse> <cfquery name="loginQuery" dataSource="CompanyInfo"> SELECT UserID, Roles FROM LoginInfo WHERE UserID = '#cflogin.name#' AND Password = '#cflogin.password#' </cfquery></pre>	<p>Executes if the user submitted a login form and both fields contain data.</p> <p>Uses the cflogin structure's name and password entries to find the user record in the database and get the user's roles.</p>
<pre><cfif loginQuery.Roles NEQ ""> <cfloginuser name="#cflogin.name#" Password = "#cflogin.password#" roles="#loginQuery.Roles#"></pre>	<p>If the query returns data in the Roles field, logs in the user using the UserID and Roles fields from the database. In this application, every user must be in some role.</p>
<pre><cfelse> <cfoutput> <H2>Your login information is not valid.
 Please Try again</H2> </cfoutput> <cfinclude template="loginform.cfm"> <cfabort></pre>	<p>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.</p> <p>The cfabort tag prevents processing of any code that follows on this page.</p>
<pre> </cfif> </cfif> </cfif> </cflogin></pre>	<p>Ends loginquery.Roles test code.</p> <p>Ends form entry empty value test.</p> <p>Ends form entry existence test.</p> <p>Ends cflogin tag body.</p>
<pre><cfif GetAuthUser() NEQ ""> <cfoutput> <form action=MyApp/index.cfm" method="Post"> <input type="submit" Name="Logout" value="Logout"> </form> </cfoutput> </cfif></pre>	<p>If a user is logged in, displays the Logout button.</p> <p>If the user clicks the button, posts the form to the application's (theoretical) entry page, index.cfm.</p> <p>Application.cfm then logs out the user and displays the login form. If the user logs in again, ColdFusion displays index.cfm.</p>

Example: loginform.cfm

The loginform.cfm page consists of the following:

```
<cfset url="http://" & "#CGI.server_name#" & ":" & "#CGI.server_port#" &
  "#CGI.script_name#">
<cfif CGI.query_string IS NOT "">
  <cfset url=url & "?#CGI.query_string#">
</cfif>
<H2>Please Log In</H2>
<cfoutput>
  <form action="#url#" 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
<pre> <cfset url="http://" & "#CGI.server_name#" & ":" & "#CGI.server_port#" & "#CGI.script_name#"> <cfif CGI.query_string IS NOT ""> <cfset url=url & "?#CGI.query_string#"> </cfif> </pre>	<p>Constructs the URL to use in the form's action attribute from CGI variables. The CGI.script_name variable identifies the originally requested page, because that page's invocation of Application.cfm causes the login form to be displayed.</p> <p>Appends any URL query string used in requesting the page so the page can process the request.</p> <p>A more secure application would use the HTTPS protocol to encrypt the user name and password. To use HTTPS, you must obtain a security certificate and enable the protocol in your web server.</p>
<pre> <H2>Please Log In</H2> <cfoutput> <form action="#url#" 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> </pre>	<p>Displays the login form. The form requests a user ID and password and posts the user's input to the page specified by the url variable.</p> <p>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.</p>

Example: securitytest.cfm

The securitytest.cfm page shows how any application page can use ColdFusion user authorization features. Application.cfm 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 code:

```
<!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
<pre><cfoutput> <h2>Welcome #GetAuthUser()#!</h2> </cfoutput></pre>	Displays a welcome message that includes the user's login ID.
<pre>ALL Logged-in Users see this message.

</pre>	Displays this message in all cases. The page does not display until a user is logged in.
<pre><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></pre>	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 application-based security with a browser's login dialog

You do not have to create a login page to display a user login form; you can rely on the browser to display its standard login page. 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. The browser then displays its login page and returns the information to the `cflogin` tag's `cflogin` structure when the user clicks the login button.

For example, the following code tells the browser to display a login form if the user has not logged in, or if the user does not provide a name "user" and password "p1", or a name "admin" and password "p2":

```
<cflogin>
  <cfif IsDefined( "cflogin" )>
    <cfif cflogin.name eq "admin" and cflogin.password eq "p1">
      <cfset roles = "user,admin">
    <cfelseif cflogin.name eq "user" and cflogin.password eq "p2">
      <cfset roles = "user">
    </cfif>
  </cfif>

  <cfif IsDefined( "roles" )>
    <cfloginuser name="#cflogin.name#" password="#cflogin.password#"
      roles="#roles#">
  </cfif>
```

```

        <!-- User has not logged in or authentication failed - send 401 --->
        <cfsetting enablecfoutputonly="yes" showdebugoutput="no">
        <cfheader statusCode="401">
        <cfheader name="WWW-Authenticate" value="Basic realm=""MySecurity"">
        <cfoutput>Not authorized</cfoutput>
        <cfabort>

    </cfif>
</cflogin>

```

Using an LDAP Directory for security information

LDAP directories are often used to store security information. The following example `cflogin` tag checks an LDAP directory to authenticate the user and retrieve the users roles.

The most important thing to note in this example is that it queries the directory twice, first as the directory manager, then with the user's identity:

- The first query uses the identity of the directory manager as the `username` attribute. This query gets the distinguished name that corresponds to the user-supplied user ID. Using the directory manager's identity ensures that there will be a valid response for any user ID in the directory.
- The second query accesses the directory with the distinguished name from the first query as the `username` attribute, and the user-supplied password as the `password` attribute. This query succeeds, and thereby authenticates the user, only if the user's password allows that user to access the directory. In other words, the application uses the user's LDAP directory password as its own password.

The "Reviewing the code" section that follows describes the code's function in detail. For more information on using LDAP directories with ColdFusion, see [Chapter 23, "Managing LDAP Directories" on page 489](#).

```

<cflogin>
<!-- setting basic attributes --->
<cfset root = "o=macromedia.com">
<cfset server="ldap.macromedia.com">
<cfset port="389">

<!-- These attributes are used in the first search. --->
<!-- This filter will look in the objectclass for the user's ID. --->
<cfset filter = "(&(objectclass=*)(uid=#Form.UserID#))">
<!-- Need directory manager's cn and password to get the user's
    password from the directory --->
<cfset LDAP_username = "cn=directory manager">
<cfset LDAP_password = "password">

<!-- Search for the user's dn information. This is used later to
    authenticate the user.
    NOTE: Do this as the Directory Manager to ensure access to the
    information --->
<cftry>
    <cfldap action="QUERY"
        name="userSearch"
        attributes="uid,dn"

```

```

        start="#root#"
        scope="SUBTREE"
        server="#server#"
        port="#port#"
        filter="#filter#"
        username="#LDAP_username#"
        password="#LDAP_password#"
    >
    <cfcatch type="Any">
        <cfset UserSearchFailed = true>
    </cfcatch>
</cftry>
<!-- If user search failed or returns 0 rows abort -->
<cfif NOT userSearch.recordcount OR UserSearchFailed>
    <cfoutput>
        <script> alert("UID for #uid# not found"); </script>
    </cfoutput>
    <cfabort>
</cfif>

<!-- pass the user's DN and password to see if the user authenticates
and get the user's roles>

<cftry>
    <cfldap
        action="QUERY"
        name="auth"
        attributes="dn,roles"
        start="#root#"
        scope="SUBTREE"
        server="#server#"
        port="#port#"
        filter="#filter#"
        username="#userSearch.dn#"
        password="#Form.password#"
    >

    <cfcatch type="any">
        <cfif FindNoCase("Invalid credentials", cfcatch.detail)>
            <cfoutput><script>alert("User ID or Password invalid for user:
                #Form.userID#")</script>
            </cfoutput>
            <cfabort>
        <cfelse>
            <cfoutput><script>alert("Unknown error for user: #Form.userID#
                #cfcatch.detail#")</script>
            </cfoutput>
            <cfabort>
        </cfif>
    </cfcatch>
</cftry>

```

```

<!-- If the LDAP query returned a record, the user is valid. -->
<cfif auth.recordcount>
    <cfloginuser name="#Form.userID#" password="#Form.password#"
        roles="#auth.roles#">
</cfif>
</cflogin>

```

Reviewing the code

The following table describes the code and its function:

Code	Description
<pre> <cflogin> <cfset root = "o=macromedia.com"> <cfset server="ldap.macromedia.com"> <cfset port="389"> <cfset filter = "(&(objectclass=*) (uid=#Form.UserID#))"> <cfset LDAP_username = "cn=directory manager"> <cfset LDAP_password = "password"> </pre>	<p>Starts cflogin tag body. Sets several of the values used as attributes in the cfldap tags as variables. This ensures that the same value is used in both tags, and makes it easier to change the settings if needed.</p> <p>Sets the directory manager's user name and password for the first query.</p>
<pre> <cftry> <cfldap action="QUERY" name="userSearch" attributes="uid,dn" start="#root#" scope="SUBTREE" server="#server#" port="#port#" filter="#filter#" username="#LDAP_username#" password="#LDAP_password#" > </pre>	<p>In a cftry block, uses the directory manager's identity to get the distinguished name (dn) for the user. If the user ID is not in the directory, returns an empty record set.</p>
<pre> <cfcatch type="Any"> <cfset UserSearchFailed = true> </cfcatch> </cftry> </pre>	<p>Catches any exception. Sets a UserSearchFailed flag to True.</p> <p>Ends the cftry block.</p>
<pre> <cfif NOT userSearch.recordcount OR UserSearchFailed> <cfoutput> <script> alert("UID for #uid# not found"); </script> </cfoutput> <cfabort> </cfif> </pre>	<p>If the LDAP lookup did not return any results or the UserSearchFailed flag is True, displays an error message and ends processing of the page</p>
<pre> <cftry> <cfldap action="QUERY" name="auth" attributes="dn,roles" start="#root#" scope="SUBTREE" server="#server#" port="#port#" filter="#filter#" username="#userSearch.dn#" password="#Form.password#"> </pre>	<p>In a try block, uses the distinguished name from the previous query and the user-supplied password to access the directory and get the user's roles. If either the dn or password is invalid, the cfldap tag throws an error, which is caught in the cfcatch block.</p>

Code	Description
<pre> <cfcatch type="any"> <cfif FindNoCase("Invalid credentials", cfcatch.detail)> <cfoutput><script>alert("User ID or Password invalid for user: #Form.userID#")</script> </cfoutput> </cfif> <cfabort> </cfcatch> </cftry> </pre>	<p>Catches any exceptions.</p> <p>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 an error message indicating the problem. Otherwise, displays a general error message.</p> <p>If an error is caught, the <code>cfabort</code> tag ends processing of the request after displaying the error description.</p>
<pre> <cfif auth.recordcount> <cfloginuser name="#Form.userID#" password="#Form.password#" roles="#auth.roles#"> </cfif> </cflogin> </pre>	<p>If the second query returned a valid record, logs in the user and sets the roles to the values returned by the query.</p> <p>Ends the <code>cflogin</code> tag body.</p>

CHAPTER 17

Developing Globalized Applications

ColdFusion 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](#) 374
- [About character encodings](#)..... 377
- [Locales](#)..... 378
- [Processing a request in ColdFusion](#)..... 379
- [Tags and functions for globalizing](#)..... 382
- [Handling data in ColdFusion](#) 385

Introduction to globalization

Globalization lets you create application for all of your customers in all the languages that your 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 language other than English. For example, you can develop your application in Japanese so that the default character set 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 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 represent 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 output appropriate responses for that language.

Importance of globalization 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 1/2/2002 is interpreted as January 2, 2002 in the United States, but as February 1, 2002 in European countries.

Another reason 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 supports globalization

ColdFusion is implemented in Java. As a Java application, ColdFusion can leverage many of the inherent globalization features to be an effective web application server. For example, ColdFusion stores all strings internally using the Unicode character encoding. Because it uses Unicode, ColdFusion can represent any text data from any language.

In addition, ColdFusion 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 and locales

When you discuss globalization issues, two topics that you must consider are the character sets 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. For more information on character sets, see [“About character encodings” on page 377](#).

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 378](#).

About character encodings

An **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 bytes 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 that apply to character encodings:

- SBCS single-byte character set such as ASCII
- DBCS double-byte character set such as Shift-JIS
- MBCS multiple-byte character set

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	SBCS	8-bit encoding used by many Western European languages
Shift-JIS	DBCS	16-bit Japanese encoding
EUC-KR	DBCS	16-bit Korean encoding
UCS-2	DBCS	Two-byte Unicode encoding
UTF-8	MBCS	ASCII is 7-bit, European characters with diacriticals are two-byte and 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 the following URL:

<http://www.w3.org/International/O-charset.html>

The Unicode character encoding

ColdFusion uses the Java Unicode Standard for representing character data internally. The Unicode Standard Character encoding can represent many major languages, including ASCII, Latin-1, Shift-JIS, and others. Therefore, ColdFusion can input, store, process, and output text from all languages supported by Unicode.

By default, ColdFusion uses UTF-8 to represent text data sent to a browser. UTF-8 converts characters into a variable-length encoding. Most data is sent as a single byte, for ASCII, or as three bytes, for most other languages. One advantages of UTF-8 is that it can be recognized by systems designed to process single-byte ASCII character while being flexible enough to handle multiple-byte character representations.

While the default format of text data output by ColdFusion is UTF-8, you can set the output type of a ColdFusion page to any character set. For example, you can output text using the Japanese language Shift-JIS character set. For more information, see [“Determining the character set of server output” on page 380](#).

Locales

A **locale** identifies the exact language and cultural settings to use for a user. The locale controls how dates and currencies are formatted, how to display time, and how to display numeric data.

In ColdFusion, a locale is identified by one or more of the elements shown in the following table:

Element	Description
language	This is the basic locale identifier, such as English. This is identified by an ISO 639 two-letter language code.
regional variation	A country code: for example the (US) in English (US). This is identified by an ISO 3166 two-letter country code.
variant	Not commonly used, variants create special locales with additional requirements. A common variant example is the euro variant used by European countries that have adopted the Euro as the currency.

Setting the locale

By default, the ColdFusion locale defaults to the locale of the JVM, which typically defaults to that of the operating system. However, when processing information for a different locale, you must change this default. You can set the locale in the JVM at startup time, or you can use the `SetLocale()` function within a ColdFusion page.

The `SetLocale()` function determines the default display format of date, time, number, and currency values. ColdFusion supports 26 locales. For the complete list, see *CFML Reference*. You use the `GetLocale()` function to determine the current locale setting of ColdFusion. If you have not made a call to `SetLocale()`, `GetLocale()` returns the locale of the JVM.

Note: In previous 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.

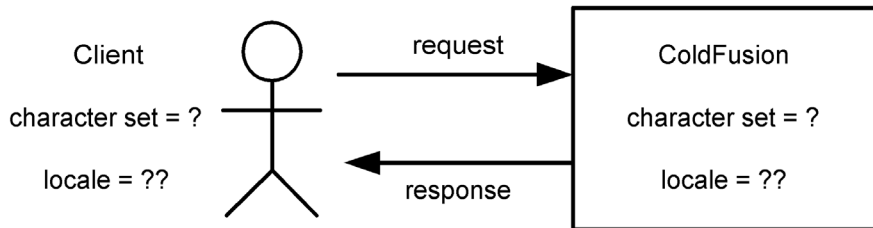
The following example code uses the `LSCurrencyFormat()` function to output the value 100,000 in monetary units for all the ColdFusion-supported locales. You can run this code to see how the locale affects the data returned to a browser.

```
<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 and reads its contents to parse it. A ColdFusion page can be encoded in a variety of ways, using different character sets and formats. 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 the dynamic content is processed by ColdFusion.

The default language of a website might be different than 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.

This section describes 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 set of a ColdFusion page

When a request for a ColdFusion page occurs, the ColdFusion server opens the page, processes the static (HTML) content, processes the dynamic content (CFML), 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 special a character at the beginning of a text stream that specifies the byte order (big/little endian) used by the page. The following table lists the common BOM values:

Encoding	BOM Signature
UTF-8	EE BB BF
UTF-16 Big Endian	FE FF
UTF-16 Little Endian	FF FE

To insert a BOM mark in a file, your editor must support BOM marks. Many IDEs support insertion of these character, including Macromedia Dreamweaver MX, however, ColdFusion Studio does not.

If your file does not contain a BOM, or if your IDE does not let you set one, you can use the `cfprocessingdirective` tag to set the character encoding of the page. However, 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 ColdFusion determines the page encoding:

- 1 Use the BOM if specified.

Macromedia recommends that you use BOM marks in your files.

- 2 Default to the JVM system encoding.

Typically, the JVM uses the same encoding as the operating system but you can override it.

- 3 Use the `pageEncoding` attribute of the `cfprocessingdirective` tag if specified.

If a BOM is detected in the file, it throws an error if `cfprocessingdirective` specifies an encoding different from the BOM.

If there are multiple occurrences of the `cfprocessingdirective` tag in the same ColdFusion page, the `pageEncoding` attribute must specify the same setting or else ColdFusion throws an error.

If you use the `cfprocessingdirective` tag, insert it as close to the top of the page as possible; for example, immediately after any `cfsetting` or `cfsilent` tag, but before any other logic.

The `cfprocessingdirective` tag specifies information to the ColdFusion compiler and is evaluated when ColdFusion compiles the page, not when it executes the page.

Therefore, you cannot embed the `cfprocessingdirective` tag within conditional logic. For example, the following code will not have any effect at execution time since the `cfprocessingdirective` tag will already have been evaluated:

```
<cfif dynEncoding is not "dynamic encoding is not possible">
  <cfprocessingdirective pageencoding=#dynEncoding# />
</cfif>
```

Determining the character set of server output

As part of servicing an HTTP request, ColdFusion must determine the character set of the data returned in the HTTP response. By default, ColdFusion returns character data using the Unicode UTF-8 format.

However, within a ColdFusion page you can override the default character encoding of the response using the `cfcontent` tag. Use the `type` attribute of `cfcontent` to specify the MIME type of the page output, including the character set, as follows:

```
<cfcontent type="text/html charset=EUC-JP">
```

ColdFusion pages (meaning .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 code 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.

Tags and functions for globalizing

ColdFusion supplies many tags and functions that you can use to develop globalized applications. This section describes these tags and functions.

Using tags for globalizing applications

The following table shows the tags that you use most often to globalize an application:

Tag	Attributes	Use
cfprocessingdirective	pageencoding	Specify the encoding of a ColdFusion page so ColdFusion can parse it. For an example, see “Determining the character set of a ColdFusion page” on page 379 .
cfcontent	type encoding	Specify the encoding of the results returned to the client browser. For an example, see “Determining the character set of server output” on page 380 .
cffile	encoding	Specify how to encode data written to or read from a file. For an example, see “Reading and writing file data” on page 387 .

Using functions for globalizing applications

ColdFusion contains functions that you use when globalizing an application. These functions include string functions as well as date, time, currency, and numeric functions.

String functions

ColdFusion provides the following functions to process string data:

Asc	Insert	LSParseNumber	REReplace
Chr	JavaCast	LTrim	REReplaceNoCase
CJustify	JSStringFormat	Mid	Reverse
Compare	LCase	MonthAsString	Right
CompareNoCase	Left	ParseDateTime	RJustify
DayOfWeekAsString	Len	REFind	RTrim
Encrypt	LJustify	REFindNoCase	SpanExcluding
FormatBaseN	ListValueCount	RemoveChars	SpanIncluding
Find	ListValueCountNoCase	RepeatString	ToBase64
FindNoCase	LSParseCurrency	Replace	Trim
FindOneOf	LSParseDateTime	ReplaceList	UCase
GetToken	LSParseEuroCurrency	ReplaceNoCase	Val
Hash			

These functions recognize the Unicode encodings so they operate correctly for all single and double-byte character sets.

Note: Applications developed for previous 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.

Date, time, currency, and numeric functions

CFML defines versions of the date, time, currency, and numeric functions that support different locales. The names of these functions are prefixed by `LS`. The following table lists the `LS` functions and several other functions used with date, time, currency, and numeric data:

Function	Function
<code>DateConvert</code>	<code>LSIsNumeric</code>
<code>GetHttpTimeString</code>	<code>LSNumberFormat</code>
<code>GetLocale</code>	<code>LSParseCurrency</code>
<code>GetTimeZoneInfo</code>	<code>LSParseDateTime</code>
<code>LSCurrencyFormat</code>	<code>LSParseEuroCurrency</code>
<code>LSDateFormat</code>	<code>LSParseNumber</code>
<code>LEuroCurrencyFormat</code>	<code>LSTimeFormat</code>
<code>LSIsCurrency</code>	<code>SetLocale</code>
<code>LSIsDate</code>	

You must precede calls to the `LS` functions with a call to the `SetLocale()` function in order to set the locale. If you do not, these functions default to using 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 formats for each locale supported by ColdFusion:

```
<!-- 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(), "mmm 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>
```

Handling data in ColdFusion

Many of the issues involved with globalizing applications deal with processing data from the various sources supported by ColdFusion, including the following:

- URL strings
- Forms
- Files
- Databases
- E-mail
- HTTP
- LDAP
- WDDX
- COM
- CORBA

This section describes how to handle data from each of these sources.

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). So 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 sets.

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 noted in the previous section, URL characters entered in using any character set 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 different character sets within a URL is to include a name/value pair within the URL that defines the character set of the URL. For example, the following URL uses a parameter called "encoding" to define the character set of the URL parameters:

```
http://company.com/prod_page.cfm?name=Stephen;ID=7645;encoding=Latin-1
```

Within the `product_name.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 set of URL parameters. The `setEncoding()` function takes two parameters; the first specifies a variable scope and the second specifies the character set 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>
```

Handling form data

The HTML `form` tag and the ColdFusion `cform` tag allows users to 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 though. 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 6 bytes.
- 2 The form returns the 6 bytes to ColdFusion as six characters. ColdFusion converts them to a representation using 2 bytes per input byte for a total of 12 bytes.
- 3 Outputting these characters results in corrupt information displayed to the user.

To work around this issue, ColdFusion supplies the `setEncoding()` function that you use when working with forms. You use this tag to specify the character set of input form text. The `setEncoding()` function takes two parameters; the first specifies the variable scope and the second specifies the character set 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.

For example, the following code 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>
```

Reading and writing file data

You use the `cffile` tag to write to and read from text files. By default, the `cffile` tag assumes that you are reading single byte character data. This causes a problem if you read a file that contains double-byte characters, whether it is a file you created using `cffile` to write to the file, or any file containing double-byte characters.

On a read, the `cffile` tag converts each byte into a two-byte representation. If the data is a single-byte representation, the read works fine. If the file contains double-byte characters, the read interprets each byte as a single character and corrupts the data.

To enable the `cffile` tag to correctly read and write double-byte characters, 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 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 the Unicode's UTF-8 and UTF-16 (UCS-2).

The database drivers supplied with ColdFusion 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 supports e-mail using the tags `cfmail` and `cfmailparam`. Because ColdFusion uses the Java mail package, which supports Unicode, you do not have to perform any special processing to handle e-mail.

HTTP

ColdFusion supports HTTP communication using the `cfhttp` and `cfhttpparam` tags and the `GetHttpRequestData` functions.

The `cfhttp` tag supports making HTTP requests using GET and POST. By default, the `cfhttp` tag uses the Unicode UTF-8 encoding for passing data. However, you can also insert the `cfhttpparam` tag to specify a MIME type.

LDAP

ColdFusion supports LDAP 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 supports `cfwddx` tag. ColdFusion stores WDDX data as UTF-8 encoding so it automatically supports double-byte character sets. You do not have to perform any special processing to handle double-byte characters with WDDX.

COM

ColdFusion supports COM through the `cfobject type="com"` tag. All strings data used in COM interfaces are constructed using wide characters (`wchars`) which support double-byte characters. You do not have to perform any special processing for interfacing with COM objects.

CORBA

ColdFusion supports CORBA through the `cfobject type="corba"` tag. The CORBA 2.0 interface definition language (IDL) basic type “String” used the Latin-1 character set 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 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

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` tag 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 Help.

Contents

- [Configuring debugging in the ColdFusion MX Administrator](#) 390
- [Using debugging information from browser pages](#) 393
- [Controlling debugging information in CFML](#) 402
- [Using the `cftrace` tag to trace execution](#) 404
- [Using the Code Compatibility Analyzer](#) 409
- [Troubleshooting common problems](#) 410

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 Debugging	<p>Enables debugging output. When this option is cleared, no debugging information is displayed, including all output of <code>cftrace</code> calls. (Cleared by default.)</p> <p>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 392.</p>
Select Debugging Output Format	<p>Determines how to display debugging output:</p> <ul style="list-style-type: none">• The <code>classic.cfm</code> template (the default) displays information as plain HTML text at the bottom of the page.• The <code>dockable.cfm</code> 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 400.
Report Execution Times	<p>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.</p>
Database Activity	<p>Displays debugging information about access to SQL data sources and stored procedures. (Selected by default.)</p>
Exception information	<p>Lists all ColdFusion exceptions raised in processing the request. (Selected by default.)</p>
Tracing information	<p>Displays an entry for each <code>cftrace</code> tag. When this option is cleared, the debugging output does not include tracing information, but the output page does include information for <code>cftrace</code> tags that specify <code>inline="Yes"</code>. (Selected by default.)</p> <p>For more information on using the <code>cftrace</code> tag, see “Using the cftrace tag to trace execution” on page 404.</p>

Option	Description
Variables	<p>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.)</p> <p>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,</p>
Enable Robust Exception Information	<p>Enables the display of the following information when ColdFusion displays the exception error page:</p> <ul style="list-style-type: none"> • 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 Performance Monitoring	<p>Allows the standard NT Performance Monitor application to display information about a running ColdFusion Application Server.</p>
Enable CFSTAT	<p>Enables you to use of the <code>cfstat</code> 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 <code>cfstat</code> utility, see <i>Administering ColdFusion MX</i>.</p>

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

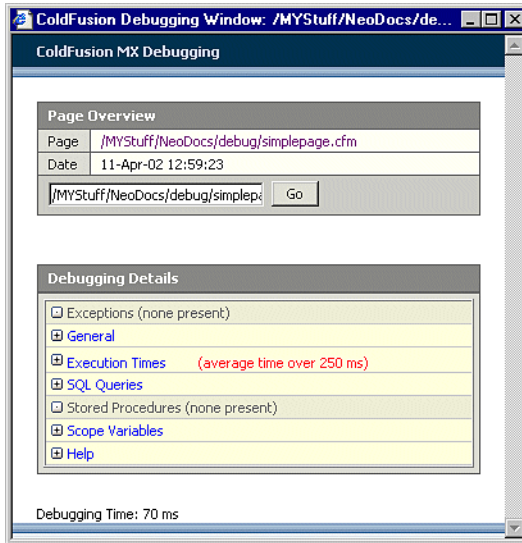
By default, when you enable debugging output, the output is visible only to local users (that is, via IP address 127.0.0.1). You can specify additional IP addresses whose users can see debugging output, or even disable output to local users. In the Administrator, use the Debugging IPs page to specify 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

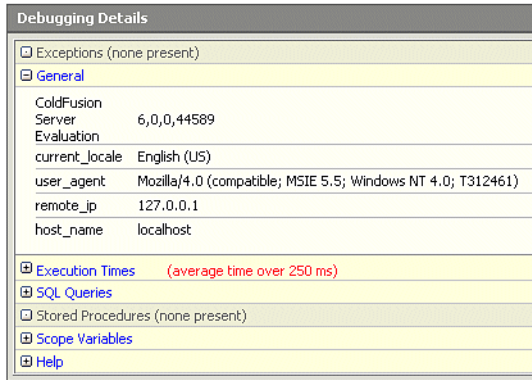
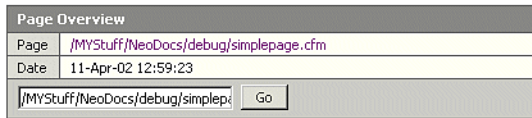
The ColdFusion debugging output that you configure in the Administrator displays whenever an HTML request completes. It represents the server conditions at the end of the request. For information on displaying debugging information while a request is processed, see [“Using the cftrace tag to trace execution” on page 404](#).

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 Server 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.cfm and OnRequestEnd.cfm pages, if used, and any CFML custom tags, pages included by the `cfinclude` tag, and any ColdFusion component (CFC) pages. You can display the execution time in two formats:


- Summary
- Tree

Note: Execution time 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 times 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 (average time over 250 ms)

Total Time	Avg Time	Count	Template
1512 ms	1512 ms	1	 C:\CFusion\MX\wwwroot\MYStuff\NeoDocs\debug\tryinclude.cfm
901 ms	451 ms	2	C:\CFusion\MX\wwwroot\MYStuff\NeoDocs\debug\mytag1.cfm
701 ms	701 ms	1	C:\CFusion\MX\wwwroot\MYStuff\NeoDocs\debug\includeme.cfm
602 ms	201 ms	3	C:\CFusion\MX\wwwroot\MYStuff\NeoDocs\debug\mytag2.cfm
10 ms	10 ms	1	C:\CFusion\MX\wwwroot\MYStuff\NeoDocs\debug\Application.cfm
70 ms			STARTUP, PARSING, COMPILING, LOADING, & SHUTDOWN

The following table describes the display fields:

Column	Description
Total Time	The total time required to process all instances of the page <i>and all pages that it uses</i> . 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.
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.

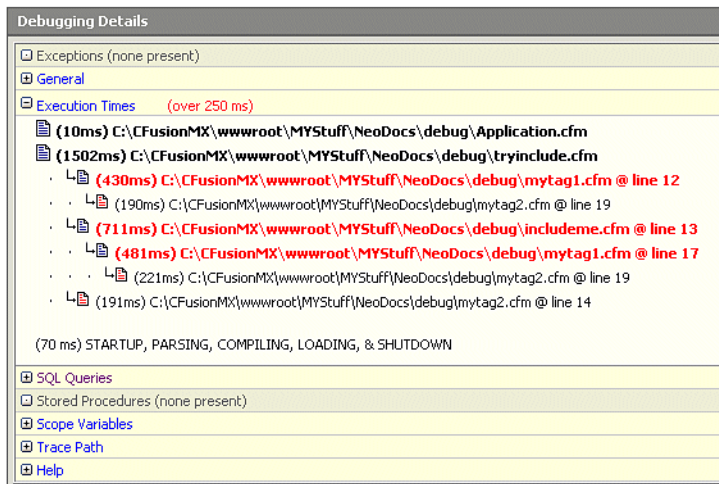
Any page with an average processing time that exceeds the highlight value that you set in the ColdFusion Administrator Debugging Settings page appears in red.

The last line of the output displays the time that ColdFusion required total time field describe the total time 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.

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 10 ms to process an Application.cfm page as part of the request.
- The requested page was tryinclude.cfm. It took 1502 ms to process this page and all pages required to execute it. The other pages took about 1330 (430 + 191) ms to process, so the code on tryincludme.cfm took about 170 ms.
- The tryinclude.cfm directly called or included three pages, mytag1.cfm, includeme.cfm, and mytag2.cfm.
- The includeme.cfm page directly called or included one page, mytag1.cfm.

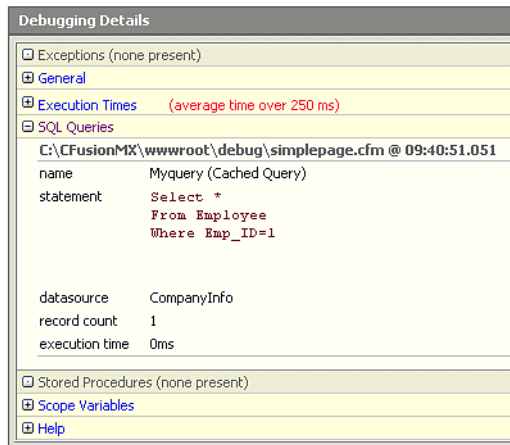
- The mytag1.cfm page directly called or included one page, mytag2.cfm
- The mytag2.cfm page takes about 200 ms to process.
- The mytag1.cfm page took about 450 ms to process. This time included the 200 ms for mytag2.cfm, so the code on mytag1.cfm took about 250 ms to process.
- The includeme.cfm page took about 700 ms to process. This time included about 480 ms for processing other pages, so the code on this page took about 220 ms.
- ColdFusion took 70 ms for processing that was not associated with a specific page.
- The total processing time was 1582 (10 + 1502 + 70) milliseconds.

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.
- Datasource 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

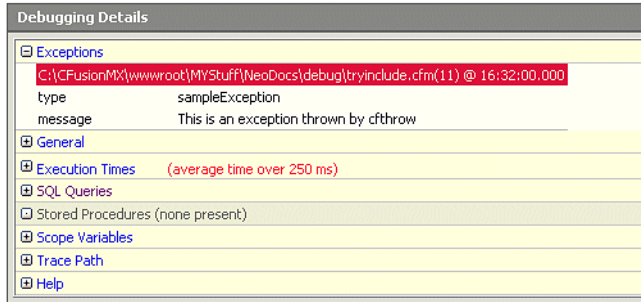
resultsets	
name	resultset
sql1	1

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 `cfproccparam` 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 `cfproccresult` 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 the ColdFusion Server.

Exceptions represent events that disrupt the normal flow of an application. You should catch and, whenever possible, recover from foreseeable exceptions in your application, as described in [Chapter 14, “Handling Errors” on page 281](#). 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:

Trace Points

- [16:52:55.055 C @ line: -1] [90 ms (1st trace)] - [custom tag] [RecordCount = 36554] *Checkpoint 3, in child tag*
- [16:52:56.056 C:\CFusionMX\wwwroot\debug\mytag1.cfm @ line: 26] [350 ms (260 ms)] - [custom tag] *Checkpoint 2 in recalculation algorithm*
- [16:52:56.056 C:\CFusionMX\wwwroot\debug\includeme.cfm @ line: 24] [611 ms (261 ms)] - [include file] [loopcount = 92347] *Shouldn't get here*
- [16:52:56.056 C @ line: -1] [851 ms (240 ms)] - [custom tag] [RecordCount = 36554] *Checkpoint 3, in child tag*

For more information on using the `cftrace` tag, see [“Using the `cftrace` tag to trace execution” on page 404](#).

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:

The screenshot shows the ColdFusion MX Debugging interface. On the left is a dockable pane with the following sections:

- Page Overview:** Page: /debug/tryinclude.cfm, Date: 11-Apr-02 17:07:41, URL: /debug/tryinclude.cfm, Go button.
- Debugging Details:**
 - Exceptions (none present)
 - General
 - Execution Times (average time over 250 ms)
 - SQL Queries
 - Stored Procedures (none present)
 - Scope Variables
 - CGI Variables
 - Client Variables
 - Cookies
 - Session Variables
 - urlopen sessionId=aa302580931018534136813
 - sessionId aa302580931018534136813
 - Trace Path
 - Help

At the bottom of the dockable pane, it says "Debugging Time: 20 ms".

On the right is the main debugging output area:

- Running ColdFusion MX
- The Time is now 05:07 PM
- You are visitor number 32917**
- This is the tryinclude file.
- This is the revised includeme file.
- This myTag1 file.
- This myTag2 file.
- This is a text
- [CFTRACE 17:07:40.040] [50 ms] [C @ line: -1] - [custom tag]**
- Checkpoint 3, in child tag*
- RecordCount** 36554
- This is a text
- [CFTRACE 17:07:41.041] [300 ms] [C @ line: -1] - [custom tag]**
- Checkpoint 2 in recalculation algorithm*
- This is a test Here are the results of the query FirstName: Ben
- Last Name: Frueh
- debug this page floating 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="CompanyInfo" 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. In the Administrator, the attribute controls debugging output *only* if the Debugging Settings page 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 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 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="CompanyInfo">
    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 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:

```

? [CFTRACE 13:21:11.011] [501 ms] [C:\CFusionMX\wwwroot\MYStuff\NeoDocs\tractest.cfm @
line: 14] - [UDF End] GetRecords UDF call has completed
MyStatus Success
  
```

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:\CFusionMX\wwwroot\MYStuff\NeoDocs\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:\CFusionMX\wwwroot\MYStuff\NeoDocs\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,” in Chapter 14.](#)

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:\CFusionMX\wwwroot\MYStuff\NeoDocs\tractest.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
<code>abort</code>	A Boolean value. If you specify <code>True</code> , ColdFusion stops processing the current request immediately after the tag. This attribute is the equivalent of placing a <code>cfabort</code> tag immediately after the <code>cftrace</code> tag. The default is <code>False</code> . If this attribute is <code>True</code> , the output of the <code>cftrace</code> call appears only in the <code>cftrace.log</code> file. The line in the file includes the text “[ABORTED]”.
<code>category</code>	<p>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.</p> <p>The <code>category</code> 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.</p> <p>You can include simple ColdFusion variables, but not arrays, structures, or objects, in the category text by enclosing the variable name in pound signs (#).</p>
<code>inline</code>	<p>A Boolean value. If you specify <code>True</code>, ColdFusion displays trace output in-line in the page. The default is <code>False</code>.</p> <p>The <code>inline</code> attribute lets you display the trace results at the place that the <code>cftrace</code> call is processed. This provides a visual cue directly in the ColdFusion page display.</p> <p>Trace output also appears in a section in the debugging information display.</p>
<code>text</code>	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 pound signs (#).
<code>type</code>	<p>A ColdFusion logging severity type. The inline trace display and <code>dockable.cfm</code> 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:</p> <ul style="list-style-type: none"> Information (default) Warning Error Fatal Information

Attribute	Purpose
var	<p>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 pound signs.</p> <p>Complex variables are displayed in inline output in <code>cfdump</code> format; the debugging display and log file report the number of elements in the complex variable, instead of any values.</p> <p>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.</p> <p>To display a function return value, put the function inside the message. Do not use the function in the <code>var</code> attribute, because the attribute cannot evaluate functions.</p>

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 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 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, pound 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 404](#).

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 pound signs (#).

```
<cdump 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 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 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:

Introduction to Databases and SQL	415
Accessing and Retrieving Data.....	433
Updating Your Database	445
Using Query of Queries	461
Managing LDAP Directories	489
Building a Search Interface.....	521
Using Verity Search Expressions	553

CHAPTER 19

Introduction to Databases and SQL

ColdFusion allows you to 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?](#) 416
- [Using SQL](#) 420
- [Writing queries using an editor](#) 428

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 works primarily with relational databases, such as Oracle, DB2, and SQL Server.

The following figure shows the basic layout of a database table:

column

row

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 used 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, you use the ColdFusion 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 *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 includes the `cftransaction` tag that allows you to 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. 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.

The 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 rows.

Tip: If you are using Dreamweaver MX, ColdFusion Studio, or 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 428](#).

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 describes 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

Operator	Description
+	Addition
-	Subtraction
/	Division
*	Multiplication

Case sensitivity with databases

ColdFusion 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 on 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 by a SELECT statement is in the form of a database table, ColdFusion 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 433](#).

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 quotes.

This is necessary because the table columns are defined to contain character strings.

Numeric data does not require the quotes.

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 updates and deletes.

Writing queries using an editor

Dreamweaver MX, ColdFusion Studio, and HomeSite+ provide a 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.

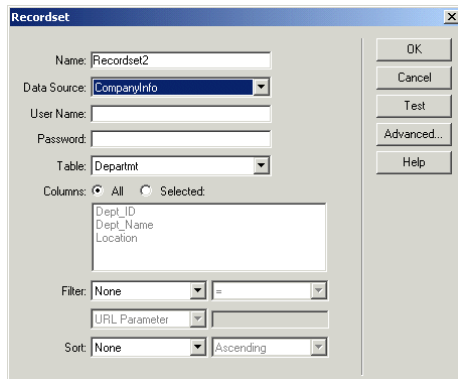
Writing queries using Dreamweaver MX

This section describes how to define a query using the Dreamweaver MX 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.

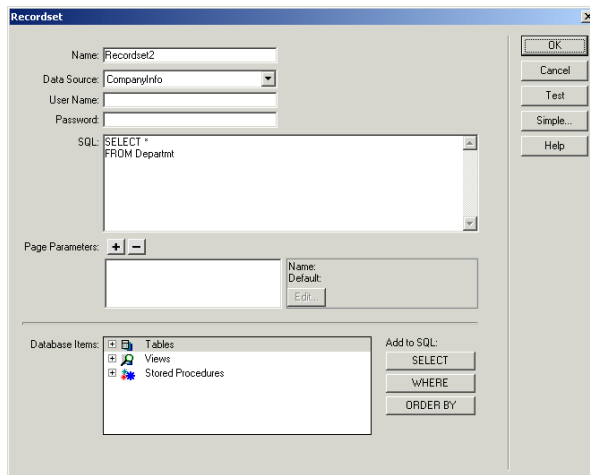
If you prefer to write your own SQL statements, or need to create more complex queries than the Simple Recordset dialog box allows, you can define record sets using the Advanced Recordset dialog box

Creating an advanced record set by writing SQL :

- 1 In the Dreamweaver MX Document window, open the page that will use the record set.
- 2 Choose **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 ColdFusion Studio and Macromedia HomeSite+

Macromedia HomeSite+ includes the combined features of HomeSite 5 and ColdFusion Studio 5, with additional support for new ColdFusion MX tags. Both HomeSite+ and ColdFusion Studio support 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.

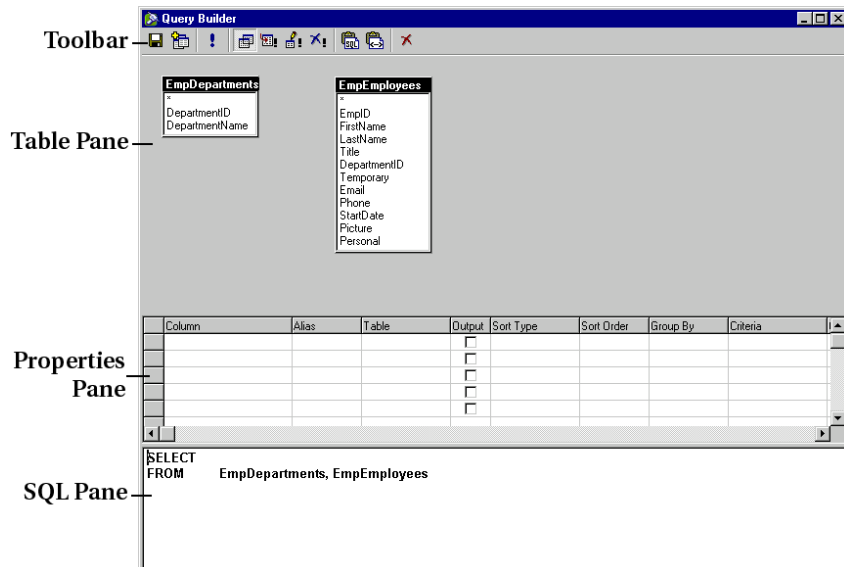
To open SQL Builder:

Do one of the following:

- Select **Tools > SQL Builder** from the HomeSite+ or ColdFusion Studio 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.

Contents

- Working with dynamic data..... 434
- Retrieving data 435
- Outputting query data..... 438
- Getting information about query results 441
- Enhancing security with `cfqueryparam` 443

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 web sites, 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 CompanyInfo data source. In Windows, this data source connects to a Microsoft Access database (company.mdb). In UNIX, this data source connects to a dBASE database. In subsequent chapters in this book, 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`, and `ColumnList` 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 = "CompanyInfo">
    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 461](#).

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.
- Later on the page, 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 with the `cfoutput` tag 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="CompanyInfo">
    SQL code...
</cfquery>
```

In this example, the query code tells ColdFusion to do the following:

- Connect to the `CompanyInfo` data source (the `company.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` later on the page.
- 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 461](#).
- 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 `CompanyInfo` data source that connects to the `company.mdb` database. This data source is installed by default. For information on adding or configuring a data source, see *Administering ColdFusion MX*.

To query the table:

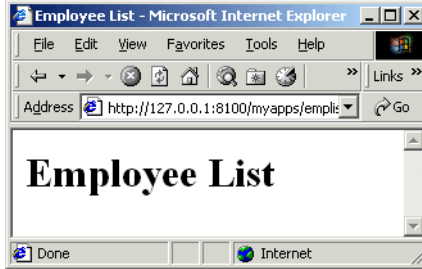
- 1 Create a ColdFusion page with the following content:

```
<html>
<head>
<title>Employee List</title>
</head>
<body>
<h1>Employee List</h1>
<cfquery name="EmpList" datasource="CompanyInfo">
    SELECT FirstName, LastName, Salary, Contract
    FROM Employee
</cfquery>
</body>
</html>
```

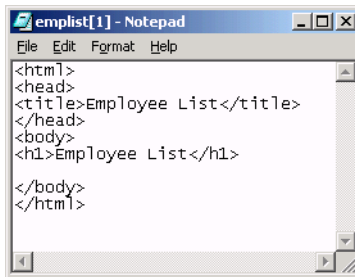
- 2 Save the page as `emplist.cfm` in the `myapps` directory under your `web_root` directory. For example, the default path on a Windows computer would be:


```
C:\CFusionMX\wwwroot\myapps\
```

- Enter the following URL in your web browser:
<http://127.0.0.1/myapps/emplist.cfm>
 Only the header appears, as the following figure shows:



- 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 CompanyInfo database. The following table describes the highlighted code and its function:

Code	Description
<code><cfquery name="EmpList" datasource="CompanyInfo"></code>	Queries the database specified in the CompanyInfo data source.
<code>SELECT FirstName, LastName, Salary, Contract FROM Employee</code>	Gets information from the FirstName, LastName, Salary, and Contract fields in the Employee table.
<code></cfquery></code>	Ends the cfquery block.

Outputting query data

After you define a query on a page, you can use the `cfoutput` tag with the `query` attribute to output data from the record set to a page. 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="CompanyInfo">
    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 [“Retrieving and Formatting Data” on page 579](#).

Reviewing the code

The results of the query appear on the page. The following table describes the highlighted code and its function:

Code	Description
<code><cfoutput query="EmpList"></code>	Displays information retrieved in the EmpList query.
<code>#EmpList.FirstName#, #EmpList.LastName#, #EmpList.Salary#, #EmpList.Contract#</code>	Displays the value of the FirstName, LastName, Salary, and Contract fields of each record, separated by commas and spaces.
<code>
</code>	Inserts a line break (go to the next line) after each record.
<code></cfoutput></code>	Ends the cfoutput block.

Query output notes and considerations

When outputting query results, keep the following guidelines in mind:

- A `cfquery` must precede the `cfoutput` that references its results. Both must be on the same page (unless you use the `cfinclude` tag; for more information, see [“Including pages with the cfinclude tag,” in Chapter 8](#)).
- It is a good idea to place queries at the top of the page, to simplify testing and debugging. However, some queries might not execute if certain conditions are not met.

- 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 quotes (").
- As with any variables that you reference for output, surround column names with pound signs (#) to tell ColdFusion to output the column's current values.
- Add a `
` 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.
CurrentRow	The current row of the query being processed by <code>cfoutput</code> .

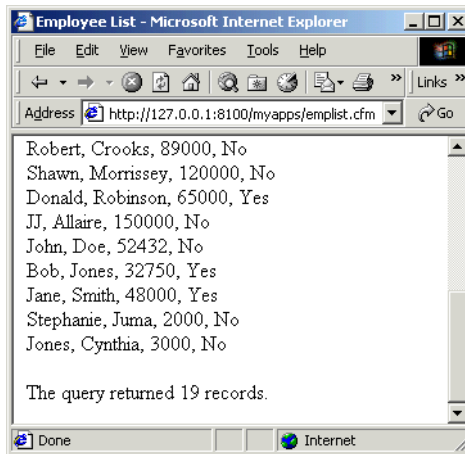
In your CFML code, you can use these variables as if they were columns in a database table.

To output the query record count 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="CompanyInfo">
    SELECT FirstName, LastName, Salary, Contract
    FROM Employee
</cfquery>
<cfoutput query="EmpList">
    #EmpList.FirstName#, #EmpList.LastName#, #EmpList.Salary#,
    #EmpList.Contract#<br>
</cfoutput>
<br>
<cfoutput>
    The query returned #EmpList.RecordCount# records.
</cfoutput>
</body>
</html>
```

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.

Note: The variable `cfquery.executionTime` contains the amount of time, in milliseconds, it took for the query to complete. Do not prefix the variable name with the query name.

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
<code><cfoutput></code>	Displays what follows.
The query returned	Displays the text "The query returned".
<code>#EmpList.RecordCount#</code>	Displays the number of records retrieved in the EmpList query.
records.	Displays the text "records..
<code></cfoutput></code>	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 pound 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 Database Management Systems (DBMSs) let you send multiple SQL statements in a single query. In many development environments—including ColdFusion, ASP, and CGI—URL or form variables in a dynamic query can append malicious SQL statements to existing queries. Be aware that there are potential security risks when you pass parameters in a query string.

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="CompanyInfo">
    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="CompanyInfo">
    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 CompanyInfo data source is an integer, you specify a cfsqltype of cf_sql_integer:

```
<cfquery name="EmpList" datasource="CompanyInfo">
    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 containing a string to a query, specify a `cfsqltype` of `cf_sql_char`, as in the following example:

```
<cfquery name = "getFirst" dataSource = "cfsnippets">
  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 quotes so that it appears as a single value to the database. Even if you pass a bad URL, it appears as follows:

```
WHERE LastName = 'Anwar 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

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](#) 446
- [Inserting data.....](#) 446
- [Updating data.....](#) 452
- [Deleting data](#) 459

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

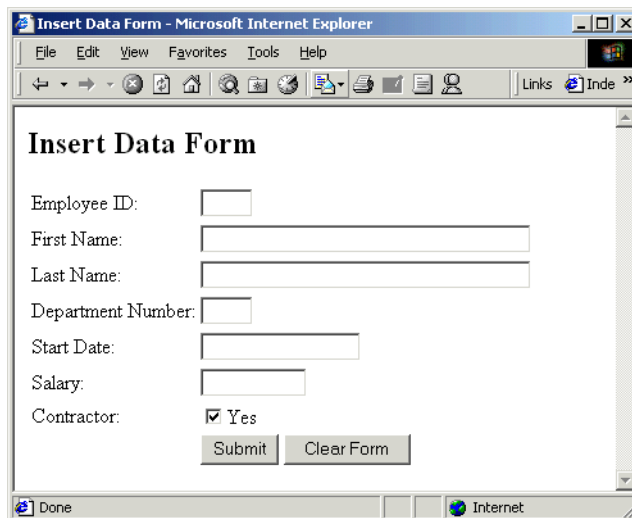
You usually use two application pages to insert data into a database:

- An insert form
- An insert action page

You can create an insert form with standard HTML form tags or with `cfform` tags (see [“Creating forms with the cfform tag” on page 608](#)). 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:



The screenshot shows a Microsoft Internet Explorer browser window titled "Insert Data Form - Microsoft Internet Explorer". The browser's address bar shows "Links" and "Inde". The main content area displays a form titled "Insert Data Form" with the following fields and controls:

- Employee ID:
- First Name:
- Last Name:
- Department Number:
- Start Date:
- Salary:
- Contractor: Yes
- Submit:
- Clear Form:

The browser's status bar at the bottom shows "Done" and "Internet".

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>&nbsp;</td>
<td><input type="Submit" value="Submit">&nbsp;<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 448](#).

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="CompanyInfo" 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 wish to compare views of the Employee table in the CompanyInfo 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
<pre><cfif not isdefined("Form.Contract")> <cfset Form.Contract = "No"> </cfif></pre>	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.
<pre><cfinsert datasource="CompanyInfo" tablename="Employee"></pre>	Creates a new row in the Employee table of the CompanyInfo database. Inserts data from the form into the database fields with the same names as the form fields.
<pre><cfoutput>You have added #Form.FirstName# #Form.LastName# to the employee database. </cfoutput></pre>	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, "Retrieving and Formatting Data" on page 579](#).

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 448](#).

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="CompanyInfo">
  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
<pre><cfquery name="AddEmployee" datasource="CompanyInfo"> INSERT INTO Employee VALUES (#{Form.Emp_ID#, '#{Form.FirstName#}', '#{Form.LastName#}', #{Form.Dept_ID#, '#{Form.StartDate#}', #{Form.Salary#, '#{Form.Contract#}') </cfquery></pre>	<p>Inserts a new row into the Employee table of the CompanyInfo database. Specifies each form field to be added.</p> <p>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.</p> <p>Because #From.Emp_ID#, #Form.Dept_ID#, and #Form.Salary# are numeric, they do not need to be enclosed in quotation marks.</p>

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="CompanyInfo">
    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="CompanyInfo">
  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>&nbsp;</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 455](#).

Reviewing the code

The following table describes the code and its function:

Code	Description
<pre><cfquery name="GetRecordToUpdate" datasource="CompanyInfo"> SELECT * FROM Employee WHERE Emp_ID = #URL.Emp_ID# </cfquery></pre>	Queries the CompanyInfo data source and returns records in which the employee ID matches what was entered in the URL that called this page.
<pre><cfoutput query="GetRecordToUpdate"> ... </cfoutput></pre>	Makes available as variables the results of the GetRecordToUpdate query in the form created in subsequent lines.
<pre><form action="update_action.cfm" method="Post"> ... </form></pre>	Creates a form whose variables will be processed on the update_action.cfm action page.
<pre><input type="Hidden" name="Emp_ID" value="#Emp_ID#">
</pre>	Uses a hidden input field to pass the Emp_ID (primary key) value to the action page.
<pre>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#">
</pre>	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.
<pre>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></pre>	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>

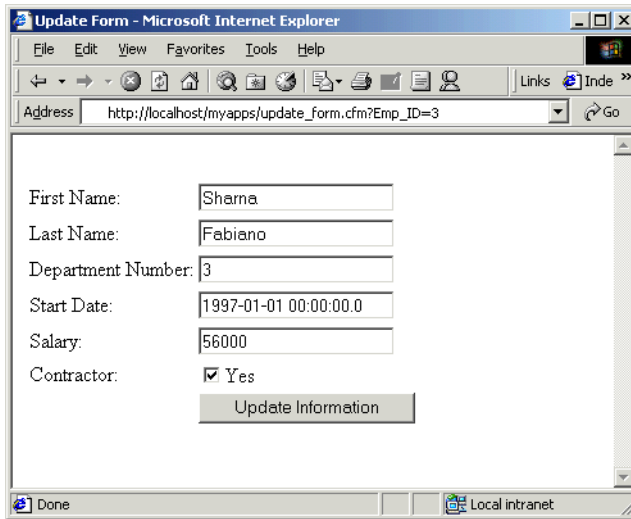
<cfupdate datasource="CompanyInfo"
  tablename="Employee">

<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:



The screenshot shows a web browser window titled "Update Form - Microsoft Internet Explorer". The address bar contains "http://localhost/myapps/update_form.cfm?Emp_ID=3". The form fields are as follows:

First Name:	Sharna
Last Name:	Fabiano
Department Number:	3
Start Date:	1997-01-01 00:00:00.0
Salary:	56000
Contractor:	<input checked="" type="checkbox"/> Yes

At the bottom of the form is a button labeled "Update Information".

- 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
<pre><cfif not isdefined("Form.Contract")> <cfset Form.contract = "No"> </cfif> <cfset form.contract = "Yes"> </cfif></pre>	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.
<pre><cfupdate datasource="CompanyInfo" tablename="Employee"></pre>	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.
<pre><cfoutput> You have updated the information for #Form.FirstName# #Form.LastName# in the employee database. </cfoutput></pre>	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 455](#).

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="CompanyInfo">
  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, type 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 with tenths of seconds, as follows:

Start Date:

This example uses the `left` function to trim the two final characters. The `CompanyInfo` data source connects to `company.mdb`.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
<pre><cfquery name="UpdateEmployee" datasource="CompanyInfo"> 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></pre>	<p>Updates the specified columns in the record in the <code>Employee</code> table of the <code>CompanyInfo</code> database that matches the primary key (<code>Emp_ID</code>).</p> <p>Because <code>#Form.Dept_ID#</code>, <code>#Form.Salary#</code>, and <code>#Form.Emp_ID#</code> are numeric, they do not need to be enclosed in quotation marks.</p> <p>Because of the way <code>cfquery</code> gets and displays dates from Access databases, you use the <code>left</code> function to trim the returned value.</p>

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="CompanyInfo">
  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
<pre><cfquery name="DeleteEmployee" datasource="CompanyInfo"> DELETE FROM Employee WHERE Emp_ID = #Form.Emp_ID# </cfquery></pre>	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.
<pre><cfoutput> You have deleted #Form.FirstName# #Form.LastName# from the employee database. </cfoutput></pre>	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](#) 462
- [About Query of Queries](#) 465
- [Query of Queries user guide](#) 474
- [BNF for Query of Queries](#)..... 486

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.

Referencing queries as objects

You can reference ColdFusion queries as objects by assigning a query to a variable, as follows:

```
<cfquery name = "query01"
  datasource = "myDNS"
  SELECT * FROM CUSTOMERS
</cfquery>
...
<cfset query02 = query01>
```

The query is not copied; both names point to the same record set data. Therefore, if you make changes to the table referenced in `query01`, the original query and the query object called `query02` both reflect these changes. If you perform a copy with an array, the array is copied.

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()`; query 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 --><cfset qInstruments = queryNew("name, instrument,
years_playing")>

<!-- 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>

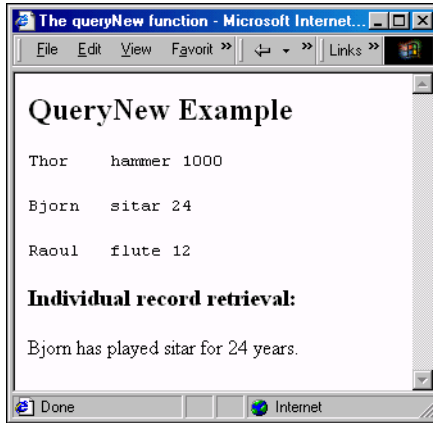
<h3>Individual record retrieval:</h3>
<cfoutput>
<p>#qInstruments.name[2]## has played #qInstruments.instrument[2]## for
#qInstruments.years_playing[2]## years.
</cfoutput>

</body>
</html>
```

- 2 Save the page as `queryNew.cfm` in the `myapps` directory under the `web_root` directory.

- 3 In your browser, enter the following URL to display the query results:
`http://127.0.0.1/myapps/queryNew.cfm`

The following figure shows how the output appears:



Note: When you create a record set, you can store in it complex objects, such as arrays and structures. However, you cannot use Query Of Queries on a record set that contains complex objects. For more information on Query of Queries, see [“About Query of Queries” on page 465](#).

About Query of Queries

After you have created a record set with a tag or function, you can query the record set in several 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 Server host machine.
- 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.

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.
You can write a normal query using a tag or function that creates a record set. This is sometimes called a **master query**. For more information, see [“Creating a record set” on page 462](#).
- 2 Write a **detail query**—a query that specifies `dbtype="query"` in its `cfquery` tag.
- 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 results 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:

```
<body>
<h1>Employee List</h1>
<!-- LastNameSearch (normally generated interactively) -->
<cfset LastNameSearch="Doe">

<!-- Master Query -->
<cfquery datasource="CompanyInfo" 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>
<br>

<p>Columns in the master query:</p>
<cfoutput>
  #master.columnlist#<br>
</cfoutput>
<br>

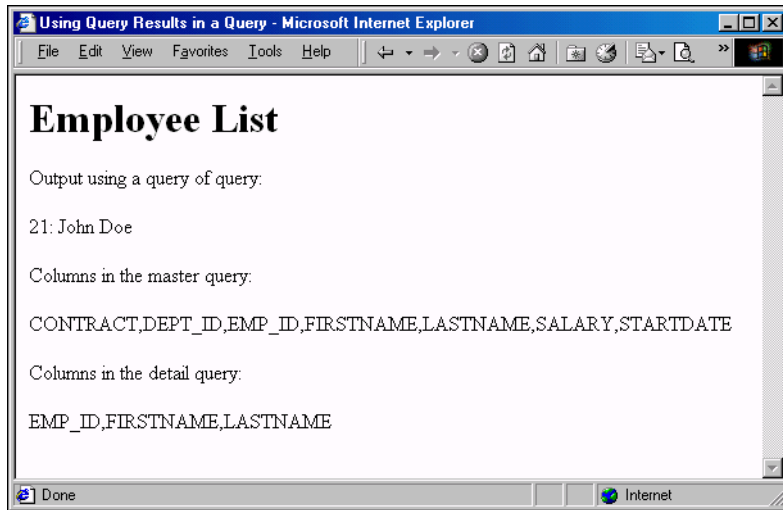
<p>Columns in the detail query:</p>
<cfoutput>
  #detail.columnlist#<br>
</cfoutput>
</body>
```

- 2 Save the page as `query_of_query.cfm` in the `myapps` directory under the `web_root`.

3 In your browser, enter the following URL to display the queryresults:

http://127.0.0.1/myapps/query_of_query.cfm

The following figure shows how the output appears:



Reviewing the code

The master query retrieves the entire Employee table from the CompanyInfo data source (the CompanyInfo database). 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
<pre>cfset LastNameSearch="Doe"</pre>	Sets the last name to use in the detail query. In a complete application, this information comes from user interaction.
<pre><cfquery datasource="CompanyInfo" name="master" cachedwithin=#CreateTimeSpan(0,1,0,0)#> SELECT * from Employee </cfquery></pre>	Queries the CompanyInfo 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.
<pre><cfquery dbtype="query" name="detail"> SELECT Emp_ID, FirstName, LastName FROM master WHERE LastName=<cfqueryparam value="#LastNameSearch#" cfsqltype="cf_sql_char" maxLength="20"></cfquery></pre>	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.

Code	Description
<pre><cfoutput query=detail> #Emp_ID#: #FirstName# #LastName#
 </cfoutput></pre>	Uses the detail query to display the list of employee IDs, first names, and last names.
<pre><cfoutput> #master.columnlist#
 </cfoutput></pre>	Lists all the columns returned by the master query.
<pre><cfoutput> #detail.columnlist#
 </cfoutput></pre>	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 441](#).

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 = "CompanyInfo">
  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>&nbsp;&nbsp;&nbsp;</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>

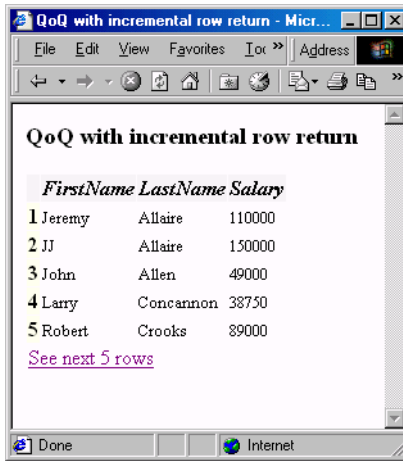
    <td valign = top>
        <font size = "-1">#Salary#</font>
    </td>
</tr>
</cfoutput>

<!-- 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 In your web browser, enter the following URL to display the query results:
http://127.0.0.1/myapps/qoq_next_row.cfm

The following figure shows how the output appears:



Using the cfdump tag with query results

As you debug your CFML code, you can use the `cfdump` tag to quickly present the value 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, Verity searches, 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 521.

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>
```

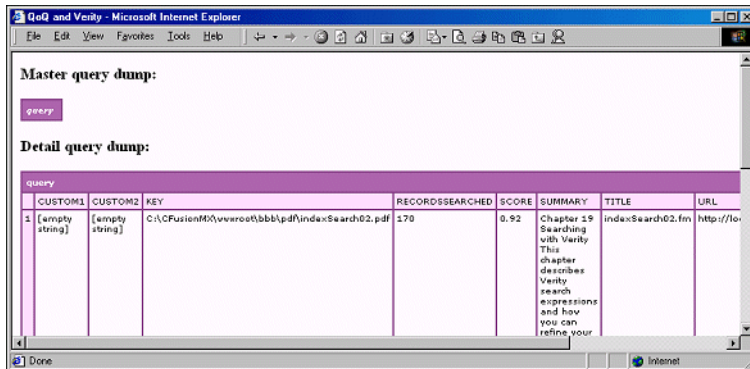
```
</h3>Detail query dump:</h3>
<cfdump var="#qoq#">
```

```
</body>
</html>
```

2 Save the page as qoq_verity.cfm in the myapps directory under the *web_root*.

3 In your web browser, enter the following URL to display the query results:
http://127.0.0.1/myapps/qoq_verity.cfm

The following figure shows how the output appears:



Note: This figure shows a collapsed master query output and an expanded detail query output. Click an output to expand or collapse it.

The first `cfdump` tag shows the master query, which retrieves all records. The second `cfdump` shows the Query of Queries results.

Tip: Adjust the score criterion of the detail query to reflect the contents of your collection.

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>

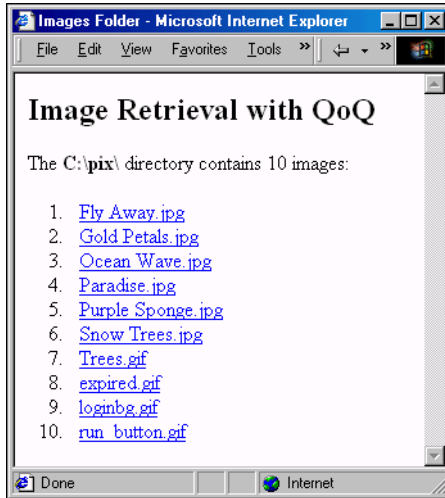
<!-- display output in a linked, ordered list -->
<cfoutput>
  <p>The <strong>#dir#</strong> directory contains #GetBoth.RecordCount#
    images:<br>
  <ol>
    <cfloop query="GetBoth">
      <li><a href="#dir/#Name#">#GetBoth.Name#</a><br>
    </cfloop>
  </ol>
</cfoutput>

</body>
</html>
```

- 2 Save the page as `qoq_cfdirectory.cfm` in the `myapps` directory under the `web_root`.

- 3 In your web browser, enter the following URL to display the query results:
http://127.0.0.1/myapps/qoq_cfdirectory.cfm

The following figure shows how the output appears:



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 tables. There are two main types of JOIN operations:

- **INNER JOIN** includes in the result set only records that are present in both tables
- **OUTER JOIN** includes in the result set all records in one of the tables.

ColdFusion does not support OUTER JOINS, nor does it support the INNER JOIN syntax, as the following example shows:

```
SELECT Dog_ID, Breed_ID,
FROM Dogs INNER JOIN Breed
ON Dogs.Dog_ID = Breed.Dog_ID;
```

ColdFusion supports INNER JOINS between two tables, as the following example shows. This operation is the most common type of join.

```
SELECT Dog_ID, Breed_ID
FROM Dogs, Breed
WHERE Dogs.Dog_ID = Breed.Dog_ID;
```

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

ColdFusion 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 match 'Sylvester', 'sylvester', 'SYLVESTER', and so on:

```
SELECT dog_name
FROM Dogs
WHERE LOWER(dog_name) LIKE 'Sylvester';
```

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 '\';
```

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.

Note: For more information, see *CFML Reference*.

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>
```

To generate output for the following examples, use the `cfdump` tag. For example, use `<cfdump var="#order_by#">` for the following `ORDER BY` example.

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')
```

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	SIZE	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			

BNF for Query of Queries

The Backus Naur Form (BNF) is a formal notation to describe programming syntax. The following is the BNF for Query of Queries:

```
Input ::= select_statement
```

```
select_statement ::= select_expression ( <ORDER> <BY> order_by_list )?
```

```
select_expression ::= ( <OPENPAREN> select_expression <CLOSEPAREN> |  
    select_specification ) ( <UNION> ( <ALL> )? select_expression )?
```

```
select_specification ::= <SELECT> ( <ALL> | <DISTINCT> )? select_list <FROM>  
    from_table_list ( <WHERE> cond_exp )? ( <GROUP> <BY> group_by_list )?  
    ( <HAVING> cond_exp )?
```

```
order_by_list ::= order_by_column ( <COMMA> order_by_column )*
```

```
order_by_column ::= ( <IDENTIFIER> | <INTEGER_LITERAL> ) ( <ASC> | <DESC> )?
```

```
group_by_list ::= column_ref ( <COMMA> column_ref )*
```

```
from_table_list ::= <IDENTIFIER> ( <COMMA> <IDENTIFIER> )*
```

```
select_list ::= select_column ( <COMMA> select_column )*
```

```
select_column ::= <ASTERISK>  
    | <IDENTIFIER> <DOT> ( <ASTERISK> | <IDENTIFIER> ( alias )? )  
    | expression ( alias )?
```

```
alias ::= ( <AS> )? <IDENTIFIER>
```

```
cond_exp ::= cond_term ( <OR> cond_exp )?
```

```
cond_term ::= cond_factor ( <AND> cond_term )?
```

```
cond_factor ::= ( <NOT> )? cond_test
```

```
cond_test ::= cond_primary ( <IS> ( <NOT> )? ( <TRUE> | <FALSE> | <UNKNOWN> ) )?
```

```
cond_primary ::= simple_cond  
    | <OPENPAREN> cond_exp <CLOSEPAREN>
```

```
simple_cond ::= like_cond  
    | null_cond  
    | between_cond  
    | in_cond  
    | comparison_cond
```

```
null_cond ::= row_constructor <IS> ( <NOT> )? <NULL>
```

```
comparison_cond ::= row_constructor comparison_operator row_constructor
```

```
between_cond ::= row_constructor ( <NOT> )? <BETWEEN> row_constructor  
    <AND> row_constructor
```



```

in_cond ::= row_constructor ( <NOT> )? <IN> <OPENPAREN> ( expression_list )
        <CLOSEPAREN>

row_constructor ::= expression

comparison_operator ::= <LESSEQUAL>
| <GREATEREQUAL>
| <NOTEQUAL>
| <NOTEQUAL2>
| <EQUAL>
| <LESS>
| <GREATER>

like_cond ::= string_exp ( <NOT> )? <LIKE> string_exp

expression_list ::= expression ( <COMMA> expression )?

expression ::= <STRING_LITERAL>
| <OPENPAREN> <STRING_LITERAL> <CLOSEPAREN>
| numeric_exp

numeric_exp ::= numeric_term ( ( <PLUS> | <MINUS> ) numeric_exp )?

numeric_term ::= numeric_factor ( ( <ASTERISK> | <SLASH> ) numeric_term )?

numeric_factor ::= ( <PLUS> | <MINUS> )? numeric_primary

numeric_primary ::= <INTEGER_LITERAL>
| <FLOATING_POINT_LITERAL>
| aggregate_func
| column_ref
| <OPENPAREN> numeric_exp <CLOSEPAREN>

aggregate_func ::= <COUNT> <OPENPAREN> count_param <CLOSEPAREN>
| ( <AVG> | <SUM> | <MIN> | <MAX> ) <OPENPAREN> ( <ALL> | <DISTINCT> )?
  numeric_exp <CLOSEPAREN>

count_param ::= <ASTERISK>
| ( <ALL> | <DISTINCT> )? numeric_exp

string_exp ::= <STRING_LITERAL>
| column_ref
| <OPENPAREN> string_exp <CLOSEPAREN>

column_ref ::= <IDENTIFIER> ( <DOT> <IDENTIFIER> )?

```


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](#) 490
- [The LDAP information structure](#) 492
- [Using LDAP with ColdFusion](#)..... 495
- [Querying an LDAP directory](#) 496
- [Updating an LDAP directory](#)..... 503
- [Advanced topics](#)..... 514

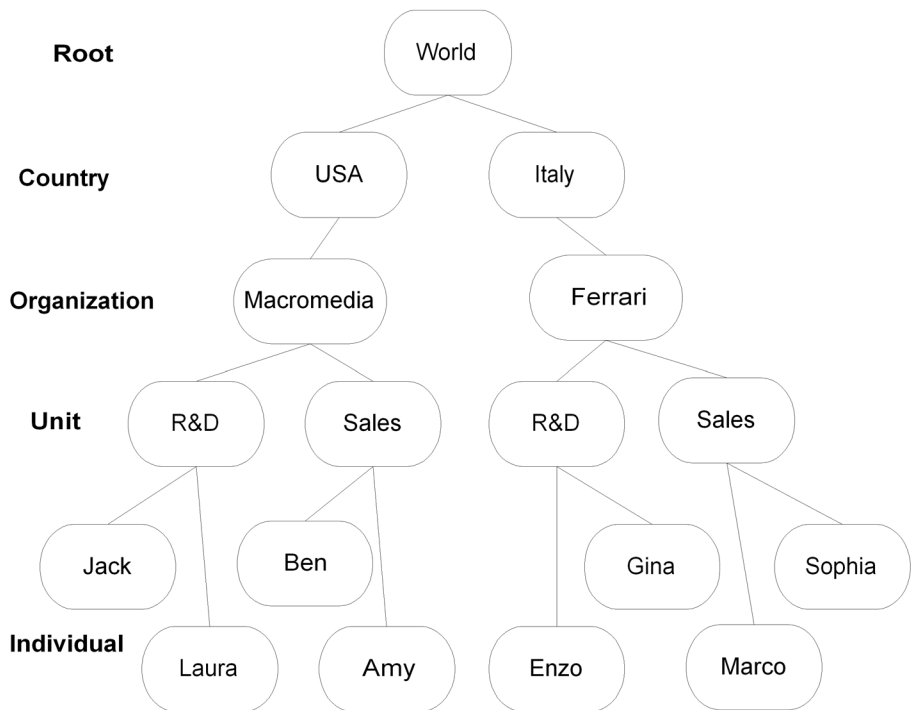
About LDAP

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.

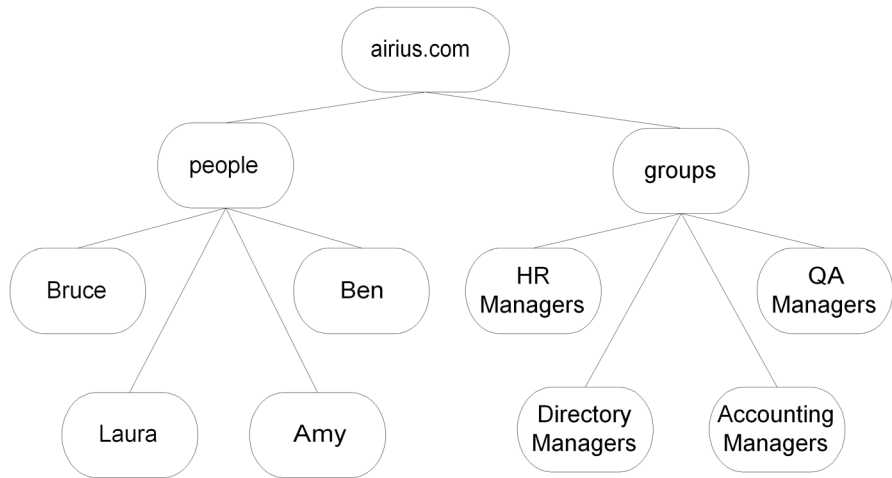
The following example shows a simple directory structure:



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 by no means limited to these types of applications. For example, you can store ColdFusion Server 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
- 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 493](#)).

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

At the time this chapter was written, Netscape provided a list of standard Attribute names on its website, at:

<http://developer.netscape.com/docs/manuals/directory/schema2/41/contents.htm>

For more information, see “Attribute type” on page 494.

Distinguished name (DN)

An entry’s **distinguished name** uniquely identifies it in the directory. A DN is made up of **relative distinguished names** (RDN)s. 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 *CFML Reference*. ColdFusion always returns DN’s 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 `cnldap 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 `cnldap 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 *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.
>=	(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".

Operator	Example	Matches
	((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>cflldap Query Example</title> </head>

<h3>cflldap 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="cflldap.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 ""))>
  <cflldap
    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=20
  >

<!-- 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
<pre><form action="cfldap.cfm" method="POST"> <input type="text" name="name">

 <input type="submit" value="Search"> </form></pre>	Uses a form to get the name or name fragment to search for.
<pre><cfif (isdefined("form.name") AND (form.name IS NOT ""))></pre>	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
<pre> <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=20 > </pre>	<p>Connects anonymously to LDAP server ldap.airius.com, query the directory, and return the results to a query object named results.</p> <p>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.</p> <p>Requests records for entries that contain the location (l) attribute value "Santa Clara" and the entered text in the common name attribute.</p> <p>Gets the common name, surname, organizational unit, e-mail address, and telephone number for each entry.</p> <p>Sorts the results first by organization name, then by surname. Sorts in the default sorting order.</p> <p>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.</p>
<pre> <table border=0 cellpadding=2 cellspacing=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> </pre>	<p>Starts a table to display the output</p> <p>Displays the number of records returned.</p> <p>Displays the common name, department, e-mail address, and telephone number of each entry.</p> <p>Displays only the first entry in the list of organizational unit values. (For more information, see the description that follows this table for more information.)</p>

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 503](#)
- [“Deleting a directory entry” on page 509](#)
- [“Updating a directory entry” on page 510](#)

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 512](#) and [“Changing a directory entry’s DN” on page 513](#) 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
<pre> <cfset myServer="ldap.myco.com"> <cfset myUserName="cn=Directory Manager"> <cfset myPassword="password"> </pre>	<p>Initializes the LDAP connection information variables. Uses variables for all connection information so that any changes have to be made in only one place.</p>
<pre> <cfparam name="fullNameValue" default=""> <cfparam name="surnameValue" default=""> <cfparam name="emailValue" default=""> <cfparam name="phoneValue" default=""> <cfparam name="uidValue" default=""> </pre>	<p>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 redisplay the page.</p>
<pre> <cfif isdefined("Form.action") AND Trim(Form.uid) IS NOT ""> </pre>	<p>Ensures that the user entered a User ID in the form.</p>
<pre> <cfif Form.action is "add"> </pre>	<p>If the user clicks Add, processes the code that follows.</p>

Code	Description
<pre> <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> </pre>	<p>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.</p>
<pre> <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)#"> </pre>	<p>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).</p> <p>The Trim function removes leading or trailing spaces from the user data.</p>
<pre> <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> </pre>	<p>Adds the new entry to the directory.</p>

Code	Description
<pre> <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> </pre>	<p>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.</p>
<pre> <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"> </pre>	<p>Queries the directory and gets the common name, surname, e-mail address, telephone number, and user ID from the matching entries.</p> <p>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.</p> <p>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.</p>
<pre> <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> </pre>	<p>Display the query results in a table.</p>

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 503.

- 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:

```
&nbsp;
<input type="Submit"
  name="action"
  value="Retrieve"
  tabindex="7">
&nbsp;
<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
<pre><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]></pre>	<p>If the user clicks Retrieve, queries the directory and gets the information for the specified User ID.</p> <p>Sets the form field's Value attribute to the corresponding query column.</p> <p>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.</p>
<pre><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></pre>	<p>The user clicks delete, deletes the entry with the specified User ID and informs the user that the entry was deleted.</p>
<pre>&nbsp; <input type="Submit" name="action" value="Retrieve" tabindex="7"> &nbsp; <input type="Submit" name="action" value="Delete" tabindex="8"></td></pre>	<p>Displays submit buttons for the Retrieve and Delete actions.</p>

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 503](#).
- 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:

```
&nbsp;
<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
<pre><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></pre>	<p>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.</p> <p>Displays a message to indicate that the entry was updated.</p> <p>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.</p>
<pre>&nbsp; <input type="Submit" name="action" value="Update" tabindex="9"></td></pre>	<p>Defines the Submit button for the update action.</p>

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	<pre>dn = "entry dn" action = "modify" modifyType = "add" attributes = "attribname=attribValue[...]"</pre>
Delete attribute from entry	<pre>dn = "entry dn" action = "modify" modifyType = "delete" attributes = "attribName[...]"</pre>

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 `sealso` LDAP attributes:

```
attributes="description=Senior Technical Writer;sealso=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 514](#).

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=#new_UID#"
  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 [“Indexing `cfldap` query results,” in Chapter 24.](#)

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 461.](#)

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
<pre><cfldap name="EntryList" server="ldap.mycorp.com" action="query" attributes="subschemaSubentry" scope="base" start=""></pre>	Gets the value of the <code>subschemaSubentry</code> attribute from the root of the directory server. The value is the DN of the schema.
<pre><cfldap name="EntryList2" server="ldap.mycorp.com" action="query" attributes="objectclasses, attributetypes" scope="base" filter="objectclass=*" start=#entryList.subschemaSubentry#></pre>	Uses the schema DN to get the <code>objectclasses</code> and <code>attributetypes</code> attributes from the schema.
<pre><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></pre>	<p>Displays the object class name, superior class, required attributes, and optional attributes for each object class in a table.</p> <p>The schema contains the definitions of all object classes in a comma delimited list, so the code uses a list type <code>cfloop</code> tag.</p> <p>The <code>thisElement</code> variable contains the object class definition. Trim off any leading or trailing spaces, then use the class definition field keywords in <code>Find</code> functions to get the starting locations of the required fields, including the Object class ID. (The ID is not displayed.)</p> <p>Gets the length of the <code>thisElement</code> string for use in later calculations.</p>

Code	Description
<pre> <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> </pre>	<p>Displays the field values. Uses the Mid function to extract individual field values from the thisElement string.</p> <p>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".</p> <p>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.</p>

Code	Description
<pre> <h2>Attribute Types</h2> <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> </pre>	<p>Does the same types of calculations for the attribute types as for the object classes.</p> <p>The attribute type field can contain the text ", alias for...". This text includes a comma, which also delimits attribute entries. Use the <code>ReplaceNoCase</code> function to replace any comma that precedes the word "alias" with an HTML <code>
</code> tag.</p> <p>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.</p>

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.

The LDAP server sends a certificate that is securely “signed” by a trusted authority and identifies (authenticates) the sender. The ColdFusion server uses the certificate to ensure that the server is valid. The ColdFusion server does not send the LDAP server a certificate, and you must use the `cfldap` tag `username` and `password` attributes to authenticate yourself to the LDAP server.

To use security, first ensure that the LDAP server supports SSL v3 security.

Specify the `cfldap` tag `secure` attribute as follows:

```
secure = "cfssl_basic"
```

For example:

```
<cfldap action="modify"
  modifyType="add"
  attributes="cn=Lizzie"
  dn="uid=lborder, 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 347](#).

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.

This chapter describes how to build a Verity search interface with which users can perform powerful searches on your application. It also describes how to index your documents and data sources so that users can search them.

Contents

- [About Verity](#) 522
- [Creating a search tool for ColdFusion applications](#) 528
- [Using the cfsearch tag](#) 542
- [Working with record sets](#) 545

About Verity

To efficiently search through paragraphs of text or files of varying types, you need full-text search capabilities. ColdFusion 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 523)
- Record sets returned from `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 much 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 containing documents for subject-based searching.
- 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. This additional capability requires only that the full path to the collection be specified in the index and search commands.
- 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 only be concerned 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 containing 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.

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.

To support multiple WYSIWYG document types, Verity bundles the KeyView Filter Kit. The KeyView Filter Kit includes document filters that support the indexing and viewing of more than 45 native document formats. Numerous popular document suites and formats are supported, including Microsoft Office 95, 97, and 2000, Corel WordPerfect, Microsoft Word, Microsoft Excel, Lotus AMI Pro, and Lotus 1-2-3.

The Verity KeyView filters support the following formats:

Word processing/text formats

- Applix Words (v4.2, 4.3, 4.4, 4.41)
- ASCII Text (All versions)
- ANSI Text (All versions)
- Folio Flat File (v3.1)
- HTML (Verity Zone Filter)
- Lotus AmiPro (v2.3)
- Lotus Ami Professional Write Plus (All versions)
- Lotus Word Pro (v96, 97, R9)
- Maker Interchange Format (MIF) v5.5
- Microsoft RTF (All versions)
- Microsoft Word (v2, 6, 95, 97, 2000)
- Microsoft Word Mac (v4, 5, 6, 98)
- Microsoft Word PC (v4.,5, 6)
- Microsoft Works (v1.0, 2.0, 3.0, 4.0)

- Microsoft Write (v1.0, 2.0, 3.0)
- PDF (Verity PDF Filter)
- Text files (Verity Text Filter)
- Unicode Text (All versions)
- WordPerfect (v5.x, 6, 7, 8)
- WordPerfect Mac (v2, 3)
- XyWrite (v4.12)

Spreadsheet formats

- Applix Spreadsheets (v4.3, 4.4)
- Corel QuattroPro (v7, 8)
- Lotus 1-2-3 (v2, 3, 4, 5, 96, 97, R9)
- Microsoft Excel (v3, 4, 5, 96, 97, 2000)
- Microsoft Excel Mac (98)
- Microsoft Works spreadsheet (v1.0, 2.0, 3.0, 4.0)

Presentation formats

- Applix Presents (v4.3, 4.4)
- Corel Presentations (v7.0, 8.0)
- Lotus Freelance (v96, 97, R9)
- Microsoft PowerPoint (v4.0, 95, 97, 2000)
- Microsoft PowerPoint Mac (98)

Picture formats

- AMI Draw Graphics (SDW)
- Applix Graphics v4.3, 4.4
- Fax Systems (TIFF CCITT) Groups 3 & 4
- Computer Graphics Metafile (CGM)
- Corel Draw CDR (TIFF Header)
- DCX Fax
- Encapsulated PostScript (EPS)
- Enhanced Metafile (EMF)
- JPEG File Interchange Format
- Lotus Pic (PIC)
- Mac PICT (raster content)
- MacPaint (MAC)
- Microsoft Excel Charts
- Microsoft Windows Animated Cursor
- Microsoft Windows Bitmap (BMP)
- Microsoft Windows Cursor/Icon
- Microsoft Windows Metafile (WMF)
- PC PaintBrush (PCX)
- Portable Network Graphics (PNG)

- Sun Raster SGI RGB
- Truevision Targa
- TIFF
- WordPerfect Graphics (WPG) v1, 2

Multimedia formats

- Audio Interchange File Format (AIFF)
- Microsoft Sound (WAV)
- MIDI (MID)
- MPEG 1 Video (MPG)
- MPEG 2 Audio
- NeXT/Sun Audio (AU)
- QuickTime Movie v2.0
- Video for Windows v2.1

Support for international languages

ColdFusion supports Verity Locales in European and Asian languages. For European languages, ColdFusion uses LinguistX™ technology from Inxight; for Asian languages, ColdFusion uses ICU (IBM® Classes for Unicode) technology. For more information about installing Verity Locales, see *Installing ColdFusion MX*.

The default language for Verity collections is English. To index data in another supported language, select it from the drop-down list when you create a collection with the ColdFusion Administrator. In CFML, the `cfcollection`, `cfindex`, and `cfsearch` tags have an optional `language` attribute that you use to specify the language of the collection that you are searching. If you do not specify a language in these tags, ColdFusion checks the `neo-verity.xml` file for the collection's language. If this is defined, ColdFusion uses that language.

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:\cfusionmx\verity\collections"
  language = "simplified_chinese">
```

The following table lists the languages names and attributes that ColdFusion supports:

Language	Language attribute	Localization technology
Arabic	arabic	ICU
Chinese (simplified)	simplified_chinese	ICU
Chinese (traditional)	traditional_chinese	ICU
Czech	czech	ICU
Danish	danish	LinguistX
Dutch	dutch	LinguistX
English	english	LinguistX
Finnish	finnish	LinguistX
French	french	LinguistX
German	german	LinguistX
Greek	greek	ICU
Hebrew	hebrew	ICU
Hungarian	hungarian	ICU
Italian	italian	LinguistX
Japanese	japanese	ICU
Korean	korean	ICU
Norwegian	norwegian	LinguistX
Norwegian (Bokmal)	bokmal	LinguistX
Norwegian (Nynorsk)	nynorsk	LinguistX

Language	Language attribute	Localization technology
Polish	polish	ICU
Portuguese	portuguese	LinguistX
Russian	russian	ICU
Spanish	spanish	LinguistX
Swedish	swedish	LinguistX
Turkish	turkish	ICU

You can register collections in the Administrator or by creating a collection with the `cfcollection` tag. If you register a given collection with ColdFusion and you specify a `language` attribute, then you do not have to specify the `language` attribute when using `cfindex` and `cfsearch` for that collection. If you do not register a given collection with ColdFusion, the language defaults to English, unless you specify it in the `language` attribute for the `cfindex` and `cfsearch` tags for that 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 Administrator to create and index the collection. Also, ColdFusion Studio has a Verity Wizard that generates ColdFusion pages that index the collection and design a search interface. The following table summarizes the steps and available methods for creating the search tool:

Step	CFML	ColdFusion MX	
		Administrator	Verity Wizard
Creating a collection	Yes	Yes	No
Indexing a collection	Yes	Yes	Yes
Designing a search interface	Yes	No	Yes

This chapter presents the non-code methods for developing a search tool, followed by code examples that perform the same task. If you have ColdFusion Studio and access to the ColdFusion Administrator, the fastest development method is as follows:

- 1 Create the collection with the ColdFusion Administrator.
- 2 Use the Verity Wizard to index the collection and design a search interface.

Creating a collection with the ColdFusion MX Administrator

Use the following procedure to quickly create a collection with the ColdFusion 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:\cfusionmx\verity\collections\
By default, ColdFusion stores collections in `\cf_root\verity\collections\` in Windows and in `/cf_root/verity/collections` in UNIX.
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.
- 5 Click Create Collection.

The name and full path of the new collection appears in the list of Connected Verity Collections:

Connected Local Verity Collections						
Actions	Alias Name	Mapped	Online	External	Language	Path
    	DemoDocs	NO	YES	NO	english	C:\CFusionMX\verity\collections\

Note: You can map a collection currently available on your network or local disk by creating a local reference (an alias) for that collection. In this procedure, enter the collection alias as the collection name, and enter a UNC (Universal Naming Convention) path to the folder for the collection.

You have successfully created a collection, DemoDocs, that currently has no data. A collection becomes populated with data when you index it. For more information, see the next section, [“About indexing a collection” on page 530](#).

About indexing a collection

A new collection is an empty shell that must be indexed before you search it. The indexing procedure also populates the collection with data contained in the collection’s files. Similar to creating a collection, you can index a collection either in the ColdFusion Administrator or programmatically.

Note: You can index and search against collections created outside of ColdFusion by using the external attribute of `cfindex` and `cfsearch`.

Use the following guidelines to determine which method to use:

Use the Administrator	Use the <code>cfindex</code> tag
To index document files	To index ColdFusion query results
When the collection does not require frequent updates	When the collection requires frequent updates
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

The `cfcollection` tag has the following `action` attribute values that can fix or improve your index:

- **repair** Repairs the internal index files of a collection. This might take a few minutes for large collections.
- **optimize** Optimizes a collection. Use this if you notice that your searches on a collection take longer than 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 reindex 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 *Administering ColdFusion MX*.

Indexing and building a search interface with the Verity Wizard

If you have ColdFusion Studio, you can use the Verity Wizard to generate a basic search and index interface. Use the following procedure to quickly create a search application for a collection. This procedure assumes the following:

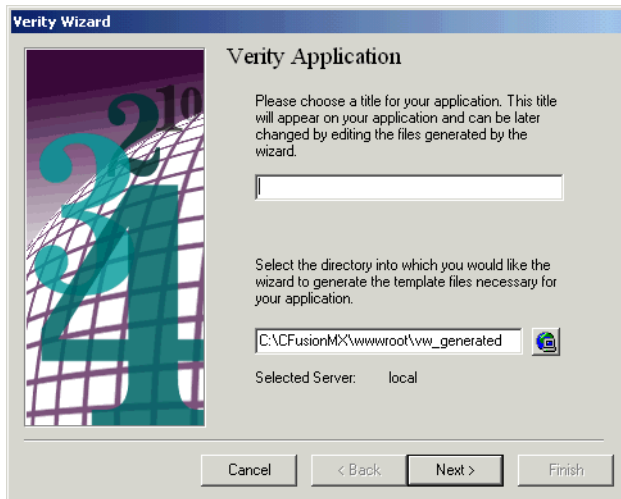
- There is an empty Verity collection to hold the indexed data. For details on how to use the ColdFusion Administrator to create a collection, see [“Creating a collection with the ColdFusion MX Administrator” on page 528](#).

- A directory contains files of several types, such as text, word processing, spreadsheet, and HTML. If this directory is within your *web_root*, then you can view the files from the web browser.
- Some of these files contain a search target word(s).
- There is an available directory to hold the four ColdFusion pages that the wizard generates.

To build a search interface using the Verity Wizard:

- 1 In ColdFusion Studio, select **File > New**.
- 2 In the New Document window, click the CFML tab.
- 3 Double-click the Verity Wizard.

The Verity Application window appears:

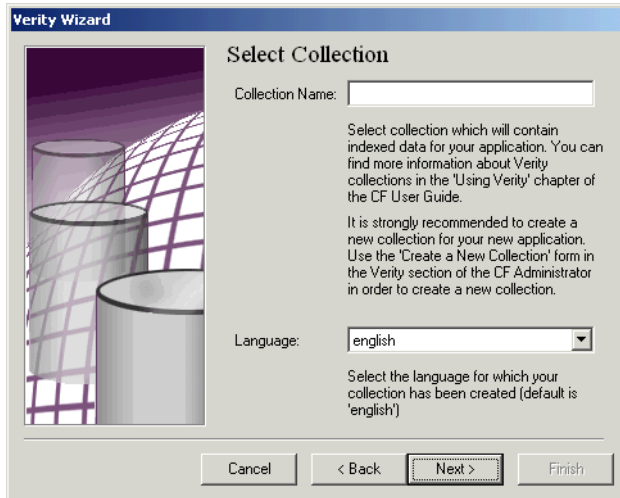


- 4 Enter the following information:

Field	Description	Example
Title	Appears at the top of each generated ColdFusion page.	Search CF Documentation
Directory	Contains the generated ColdFusion pages. The directory should be under your web_root so that you can view ColdFusion pages in the web browser.	web_root\vw_generated

- Click Next.

The Select Collection window appears:

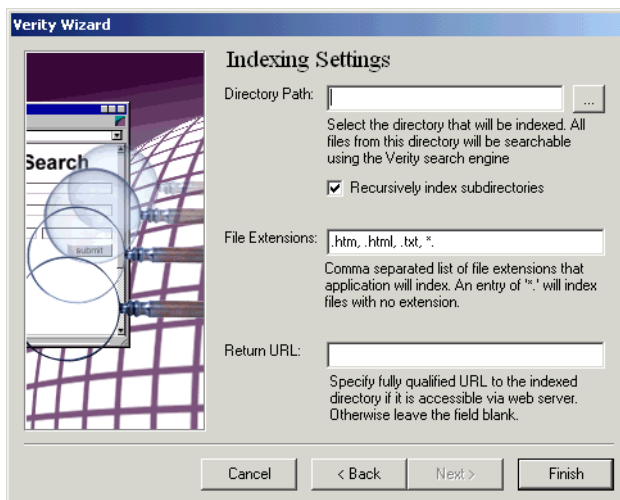


- Enter the following information:

Field	Description	Example
Collection Name	The name of the collection you created in the ColdFusion Administrator (or by using the <code>cfcollection</code> tag).	DemoDocs
Language	The language used to create the collection (English is the default).	english

- Click Next.

The Indexing Settings window appears:

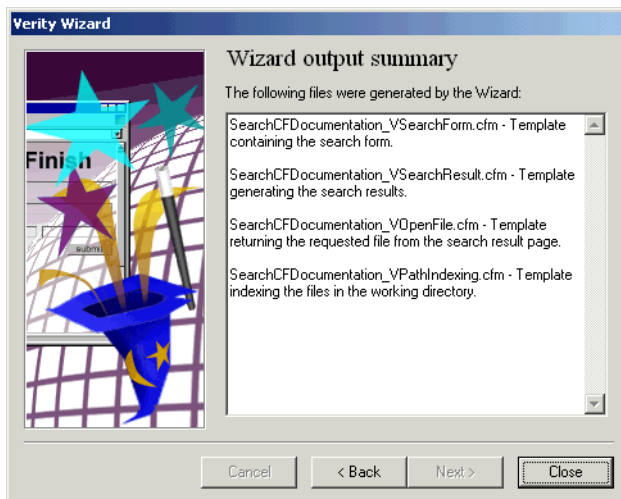


8 Enter the following information:

Field	Description	Example
Directory Path	The directory that contains the documents to be indexed.	C:\CFusionMX\wwwroot\cfdocs
Recursively Index Subdirectories	(Optional) Extends the indexing operation to all directories below the selected path.	enabled (default)
File Extensions	The type(s) of files to index. Use a comma to separate multiple file types.	.htm, .html, .xml
Return URL	(Optional) If your documents are beneath the web_root, enter a URL that corresponds to the Directory Path.	http://127.0.0.1:8500/cfdocs/

9 Click Finish.

The wizard generates four ColdFusion pages to the directory you specified in step 4, and displays an output summary:



Note: The file names are in the format pagetitle_Vpagename.cfm, where pagetitle is the value you specified in step 4 and pagename is SearchForm, SearchResult, OpenFile, or PathIndexing.

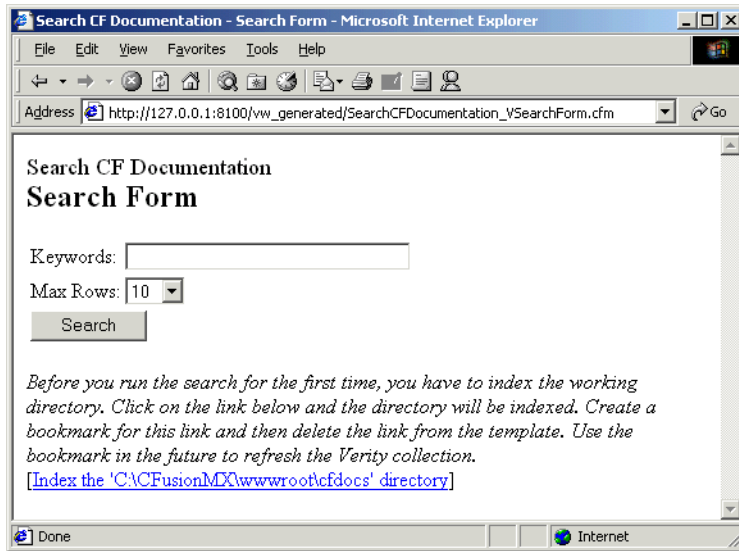
10 Click Close.

The wizard closes and the files open in ColdFusion Studio (you can adjust its size to display all file tabs).

- 11 Browse the SearchForm page in ColdFusion Studio.

Alternatively, you can use the web browser; if you do so, enter an HTTP URL that corresponds to your SearchForm, such as:

http://127.0.0.1:8500/vw_generated/SearchCFDocumentation_VSearchForm.cfm:



- 12 Click the Index link at the bottom of the page.

A confirmation message appears when indexing successfully completes.

- 13 Click the web browser's back button to return to the search form.

- 14 Enter your search term(s); for example, Verity AND data source.

Tip: For more information on the Verity search syntax, see ["Using Verity Search Expressions"](#) on page 553.

- 15 Click Search.

In ColdFusion Studio 4.x, the following compilation error might display:

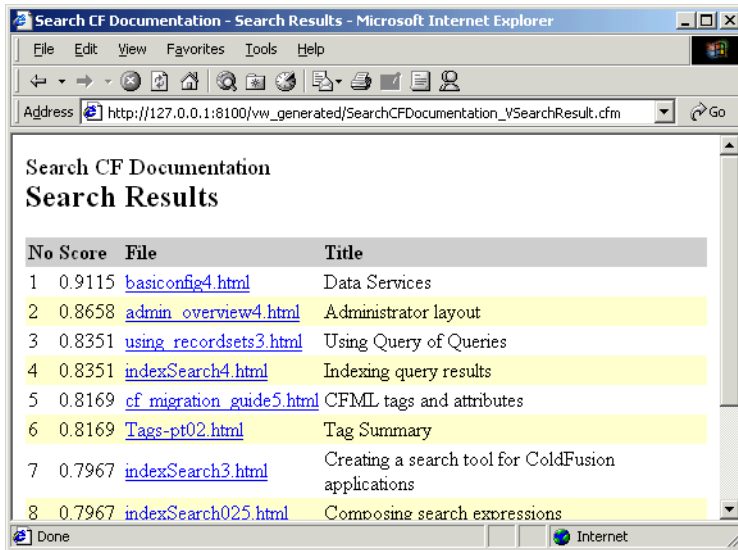
Invalid parser construct found on line 46 at position 49. ColdFusion was looking at the following text:'

To correct this error, do the following:

- a In ColdFusion Studio, open the SearchResult page in Edit mode; for example, WizardDocDemo_VSearchResult.cfm.
- b In line 46, delete the pound signs that precede the hexadecimal color codes. The correct code is:

```
<TR bgcolor="#If(CurrentRow Mod 2, DE('FFFFFF'), DE('FFFCF'))#">
```
- c Save the file.
- d Browse the SearchForm page and enter the search target.

Your search results appear:



If you entered a Return URL value and your documents are beneath your web_root (as in this procedure), you can click the link to open them.

You now have Verity search capability for your ColdFusion application. You can edit the generated ColdFusion pages or copy the generated code into the current pages to better integrate with your application.

You can create a search interface without using the Verity Wizard. The remainder of this chapter describes how to write CFML code that is functionally identical to the pages generated by the wizard. You can write the code using your text editor and preview it in the web browser.

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:

- **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 *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** (Optional) The language used to create the collection (English, by default).

You can create a collection by directly assigning a value to the `name` attribute of the `cfcollection` tag, as shown in the following code:

```
<cfcollection action = "create"
  collection = "a_new_collection"
  path = "c:\CFusionMX\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="Repair">Repair<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:\cfusionmx\verity\collections\">
      <p>The collection #Form.CollectionName# is created.
    </cfcase>

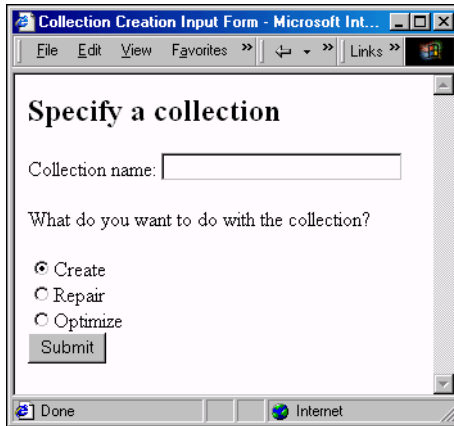
    <cfcase value="Repair">
      <cfcollection action="Repair"
        collection=#Form.CollectionName#>
      <p>The collection #Form.CollectionName# is repaired.
    </cfcase>

    <cfcase value="Optimize">
      <cfcollection action="Optimize"
        collection=#Form.CollectionName#>
      <p>The collection #Form.CollectionName# is optimized.
    </cfcase>

    <cfcase value="Delete">
      <cfcollection action="Delete"
        collection=#Form.CollectionName#>
      <p>Collection deleted.
    </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://127.0.0.1/myapps/collection_create_form.cfm

The following figure shows how the output appears:



- 4 Enter a collection name; for example, CodeColl.
- 5 Verify that Create is selected and submit the form.
- 6 (Optional) In the ColdFusion Administrator, reload the Verity Collections page. The name and full path of the new collection appears in the list of Connected 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 538](#).

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. When using this tag, the following attributes correspond to values entered in the ColdFusion MX Administrator:

- **collection** The name of the collection. If you are indexing an external collection (external = "Yes"), you must also specify the fully qualified path for the collection.
- **action** (Optional) Can be update (the default action), delete, purge, or refresh.
- **extensions** (Optional) The delimited list of file extensions that ColdFusion uses to index files if type="Path".
- **key** (Optional) The path containing the files you are indexing if type="path".
- **URLpath** (Optional) The URL path for files if type="file" and type="path". When the collection is searched with `cfsearch`, the pathname is automatically prefixed to filenames and returned as the url attribute.
- **recurse** (Optional) Yes or No. Yes specifies, if type = "Path", that directories below the path specified in the key attribute are included in the indexing operation.
- **language** (Optional) The language of the collection. English is the default.

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>

  <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 will 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>

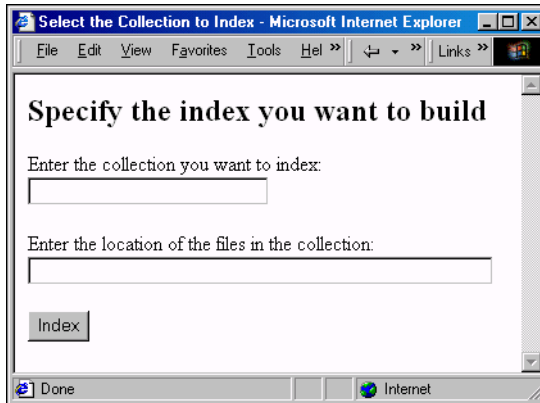
<cfindex collection="#Form.IndexColl#"
  action="refresh"
  extensions=".htm, .html, .xls, .txt, .mif, .doc"
  key="#Form.IndexDir#"
  type="path"
  urlpath="#Form.IndexDir#"
  recurse="Yes"
  language="English">

<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://127.0.0.1/myapps/collection_index_form.cfm

The following figure shows how the output appears:



- 4 Enter a collection name; for example, CodeColl.
- 5 Enter a file location; for example, C:\CFusionMX\wwwroot\vw_files.
- 6 Click Index.

A confirmation message appears upon successful completion.

Note: For information about using the `cfindex` tag with a database to index a collection, see [“Using database-directed indexing” on page 551](#).

Indexing a collection with the ColdFusion Administrator

As an alternative to programmatically indexing a collection and to using the Verity Wizard, use the following procedure to quickly index a collection with the ColdFusion Administrator.

To use ColdFusion Administrator to index a collection:

- 1 In the list of Connected Verity Collections, select a collection name; for example, CodeColl.
- 2 Click Index to open the index page.
- 3 For File Extensions, enter the type(s) 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 [“Support for international languages” on page 526](#).

8 Click Submit Changes.

The indexing process. 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.

Using the cfsearch tag

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. The following table compares the two tags:

<code>cfquery</code>	<code>cfsearch</code>
Searches a data source	Searches a collection
Requires name attribute	Requires 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 <code>cfoutput</code> to display query results	Uses <code>cfoutput</code> 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:

- `name` The name of the search query.
- `collection` The name of the collection(s) being searched. Use a fully qualified path for an external collection. Separate multiple collections with a comma; for example, `collection = "sprocket_docs,CodeColl"`.
- `criteria` The search target (can be dynamic).

Each `cfsearch` returns variables that provide the following information about the search:

- `RecordCount` The total number of records returned by the search.
- `CurrentRow` The current row of the record set being processed by `cfoutput`.
- `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.

Note: To use `cfsearch` to search a Verity K2 Server collection, the `collection` attribute must be the collection's unique alias name as defined in the `k2server.ini` and the `external` attribute must be "No" (the default). For more detail, see *Administering ColdFusion MX*.

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 a search target word(s) in this form, which passes this 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#">

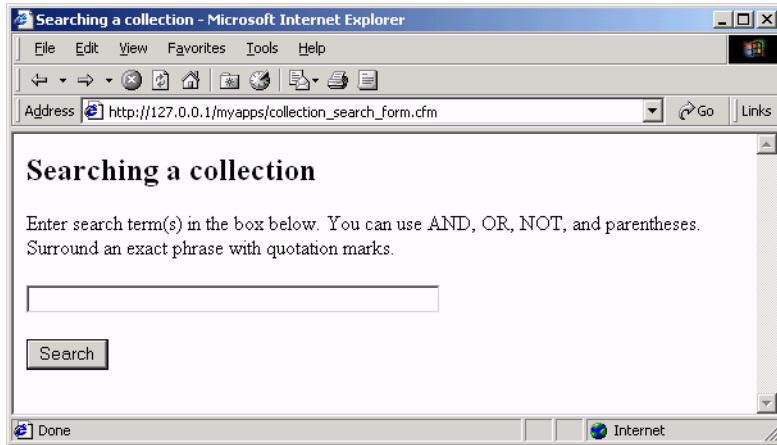
<h2>Search Results</h2>
<cfoutput>
Your search returned #codecoll_results.RecordCount# file(s).
</cfoutput>

<cfoutput query="codecoll_results">
    <p>
        File: <a href="#URL#">#Key#</a><br>
        Document Title (if any): #Title#<br>
        Score: #Score#<br>
        Summary: #Summary#</p>
</cfoutput>
</body>
</html>

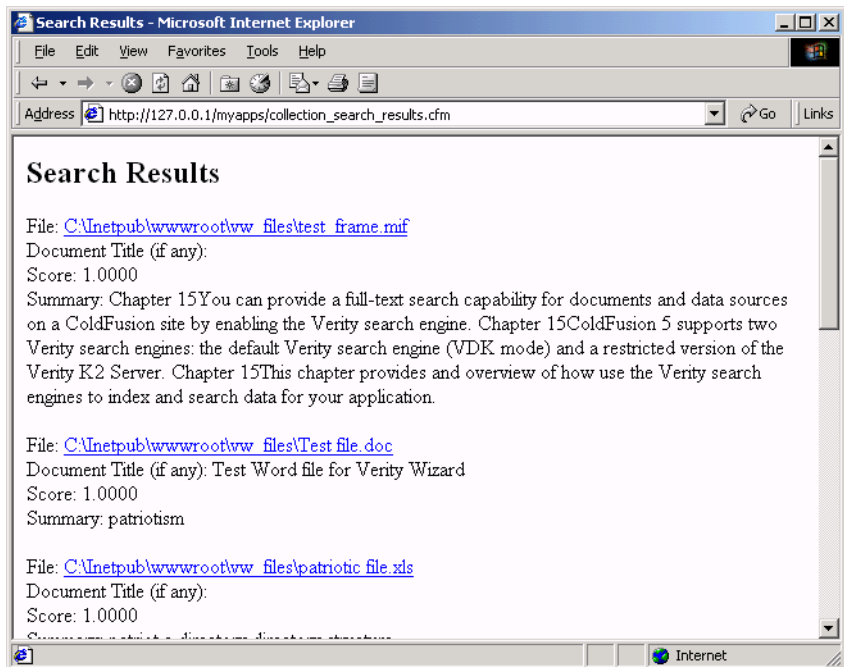
```

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.
The following figure shows how the output appears:



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 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 553](#).

Working with record sets

The `cfquery`, `cfldap`, and `cfpop` tags return the results of a database query in a record set. In some cases, you might want to search the record set. This section describes the reasons and procedures for indexing the results of database, LDAP, and pop queries. It also describes how a database can direct the indexing process, using different values for the `type` attribute of the `cfindex` tag.

Indexing database record sets

The following are the steps to perform a Verity search on record sets:

- 1 Write a query to generate a record set.
- 2 Index the record set.
- 3 Search the record set.

Performing searches against a Verity collection rather than using `cfquery` provides faster access, because the Verity collection indexes the database. Use this technique instead of `cfquery` in the following cases:

- You want to index textual data. You can search Verity collections containing textual data much more efficiently with a Verity search than with a SQL query.
- 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 your users to run queries but not update database tables.

Indexing the record set from a ColdFusion query involves an extra step not required when you index documents. You must code the query and output parameters, and then use the `cfindex` tag to index the record set from a `cfquery`, `cfldap`, or `cfpop` query.

You write a `cfquery` that retrieves the data to index, then you pass this information to a `cfindex` tag, which populates the collection. The `cfindex` tag contains the following attributes that correspond to the data source:

The <code>cfindex</code> attribute	Description
<code>key</code>	Primary key of the data source table
<code>title</code>	Specifies a query column name
<code>body</code>	Column(s) that you want to search for the index

Using the `cfindex` tag on large custom query data can cause a “Java out of memory error” or lead to excessive disk use on your computer. Because ColdFusion reads custom queries into memory, if the query size is larger than your physical memory, then paging of physical memory to disk occurs. The size of physical memory used is the smaller of the actual physical memory on your computer and the Java Virtual Machine (JVM) maximum memory parameter. You can specify the JVM parameter in the Administrator or in the configuration file `cfsuionmx/runtime/bin/jvm.config` by the argument `[-Xmx512m]`.

The following procedure assumes that you have a Verity collection named CodeColl. For more information, see [“Creating a collection with the cfcollection tag” on page 535](#). The following procedure uses the CompanyInfo data source that is installed with ColdFusion.

To index a ColdFusion query:

- 1 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="CompanyInfo">
  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">

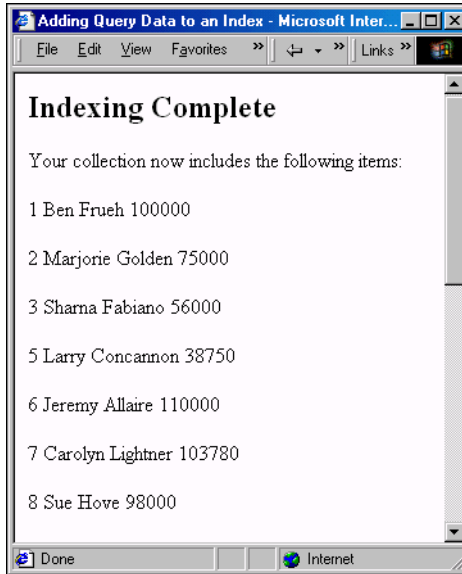
<h2>Indexing Complete</h2>

<!-- 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>
```

- 2 Save the file as collection_db_index.cfm in the myapps directory under the web root directory.

3 Open the file in the web browser to index the collection.

The resulting record set appears:



Using the `cfindex` tag for indexing tabular data is similar to indexing documents, with the following exceptions:

- You set the `type` attribute to `custom` when indexing tabular data.
- You refer to column names from the `cfquery` in the `body` attribute.

To search and display database records:

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#">

<!-- 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>

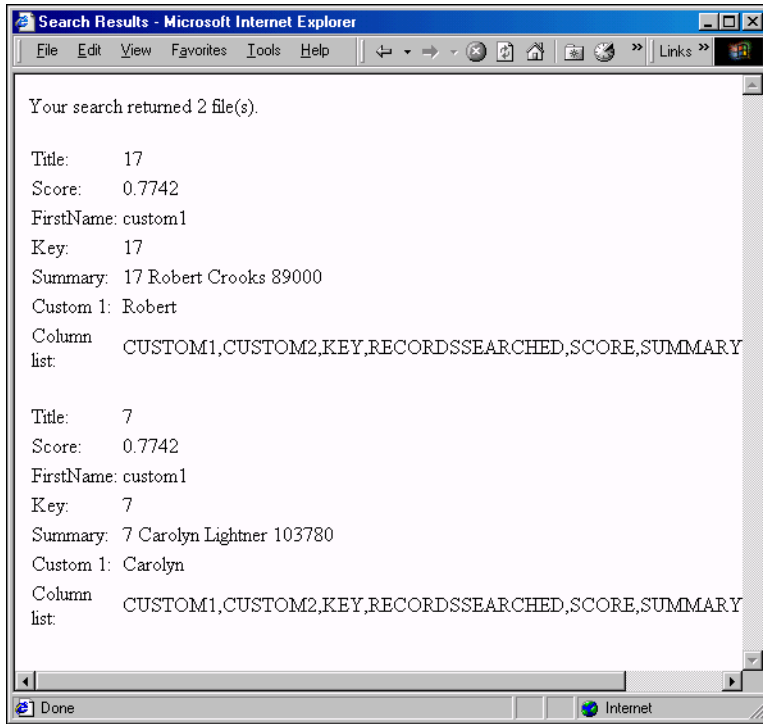
</cfoutput>

</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; for example, search the CodeColl collection for `lightner` or `crooks`.

The following figure shows how the output appears:



Indexing cfldap query results

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 allow users to 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, so re-index the collection before processing a search request.

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="(|(0=a*) (0=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*">

<!-- 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 containing 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="update"
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">

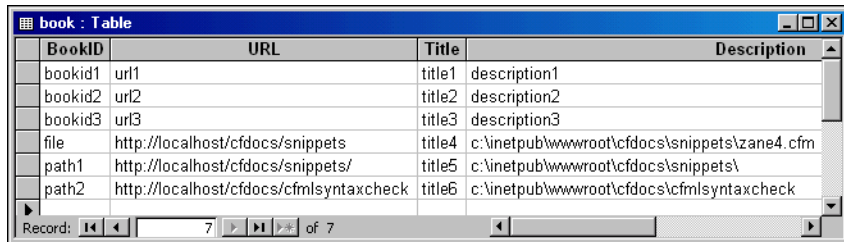
<!-- Output search record set -->
<cfoutput query="s_messages">
    #key#, #title# <br>
</cfoutput>

```

Using database-directed indexing

You can use the `cfindex` tag with a database that contains information on how to construct, or populate, the index. The `cfindex` tag has a `type` attribute, which can have `custom`, `file`, or `path` as its value. When `type=custom`, ColdFusion populates a collection with the contents of the record set. When `type=file` or `type=custom`, the record set becomes the input to perform any action—as defined by the `action` attribute—that uses the `key` attribute as input for filenames or filepaths.

The following figure shows a database that you can use to populate a collection:



BookID	URL	Title	Description
bookid1	url1	title1	description1
bookid2	url2	title2	description2
bookid3	url3	title3	description3
file	http://localhost/cfdocs/snippets	title4	c:\inetpub\wwwroot\cfdocs\snippets\zane4.cfm
path1	http://localhost/cfdocs/snippets/	title5	c:\inetpub\wwwroot\cfdocs\snippets\
path2	http://localhost/cfdocs/cfm-syntaxcheck	title6	c:\inetpub\wwwroot\cfdocs\cfmsyntaxcheck

The following code shows how to populate a collection named snippets with files that are specified in the description column of the database:

```
<html>
<head>
  <title>Database-directed index population</title>
</head>

<body>

<cfquery name="bookquery"
  datasource="book">
  SELECT * FROM book where bookid='file'
</cfquery>

<cfoutput query="bookquery">
  #url#,#description# <br>

<cfindex collection="snippets" action="update" type="file" query="bookquery"
  key="description" urlpath="url">

</cfoutput>
</body>
</html>
```

Use the following code to search the snippets collection and display the results:

```
<cfsearch name="mySearch" collection="snippets" criteria="*.*">
<cfdump var="#mySearch#">
```

The following code shows how to populate the snippets collection with paths that are specified in the description column of the database:

```
<html>
<cfquery name="bookquery"
  datasource="book">
  SELECT * FROM book where bookid='path1' or bookid='path2'
</cfquery>

<cfoutput query="bookquery">
  #url#,#description# <br>

<cfindex collection="snippets" action="update" type="path" query="bookquery"
  key="description" urlpath="url" >

</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](#) 554
- [Using simple queries](#) 555
- [Using explicit queries](#) 558
- [Composing search expressions](#) 562
- [Refining your searches with zones and fields](#) 573

About Verity query types

When you search a Verity collection, you can use either a simple or explicit query. The following table compares the two types:

Query type	Content	Use of operators and modifiers	CFML example
Simple	One or more words	Uses STEM operator and MANY modifier, by default	<pre><cfsearch name = "band_search" collection="bbb" type = "simple" criteria="film"></pre>
Explicit	Words, operators, modifiers	Must be specified	<pre><cfsearch name = "my_search" collection="bbb" type = "explicit" criteria="<WILDCARD>'sl[iau]m'"></pre>

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 555](#).

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 563](#).

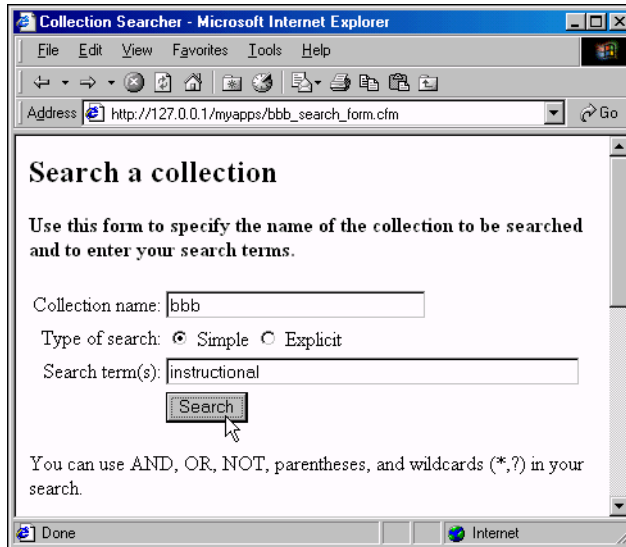
Stemming in simple queries

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.

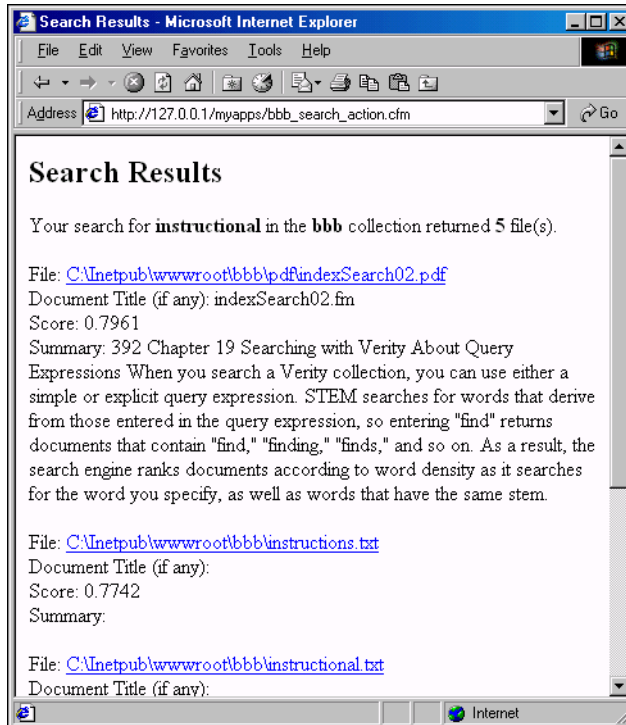
The STEM operator works on words, not word fragments. A search for instrument returns documents containing instrument, instruments, instrumental, and instrumentation, whereas a search for instru does not. (A wildcard search for instru* returns documents with these words, and also those with instruct, 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 572](#).

The following figure shows a basic search interface performing a single word search:



The results of this search show the effects of the STEM operator and MANY modifier:



In CFML, enter your search term(s) 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 558](#).
- Use the WORD operator. For more information, see [“Operators” on page 563](#).

In CFML, you can prevent stemming by enclosing the double-quoted search term with single quotes, as follows:

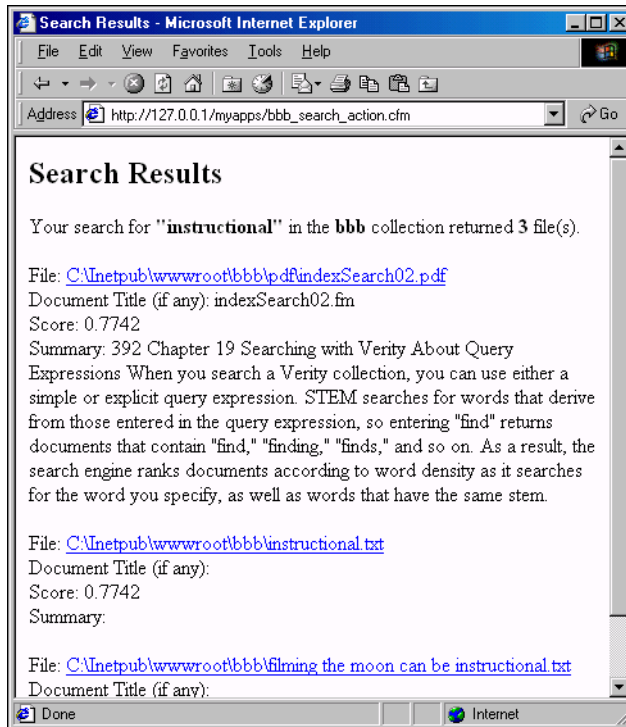
```
<cfsearch name="search_name"  
  collection="bbb"  
  type="simple"  
  criteria="'instructional'">
```

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 enclose the search term in double quotation marks, Verity does not use the STEM operator. For example, a search for “instructional”—enclosed in quotation marks—does not return files that contain `instruct`, `instructs`, `instructions`, and so on (unless the files also contain `instructional`). As the following figure shows, this search retrieves fewer files than a search without quotation marks:



Using AND, OR, and NOT

Verity has many powerful operators and modifiers available for searching (for more information, see “[Operators and modifiers](#)” on page 563). However, users might only use the most basic operators—AND 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
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

Wildcard	Description	Example	Search result
{ }	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> 's[^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 backquotes, 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 backquotes:

```
`*n``t`
```

You can use paired backquotes 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.

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 the user types, use the `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 568). 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, 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:\\cfusionmx\\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 containing 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 less 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.	<ul style="list-style-type: none">• 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.	See the text immediately after this table for examples.
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 574](#).

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.

Operator	Description	Example
NEAR/N	<p>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.</p> <p>You can specify multiple search terms using multiple instances of NEAR/N as long as the value of N is the same.</p>	<p>commute <NEAR/10> bicycle <NEAR/10> train <NEAR/10> retrieves documents that contain stemmed variations of these words within 10 words of each other.</p>
PARAGRAPH	<p>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.</p>	<p><PARAGRAPH> (mission, goal, statement) retrieves documents that contain these terms within a paragraph.</p>
PHRASE	<p>Selects documents that include a phrase you specify. A phrase is a grouping of two or more words that occur in a specific order.</p>	<p><PHRASE> (mission, oak) returns documents that contain the phrase mission oak.</p>
SENTENCE	<p>Selects documents that include all of the words you specify within the same sentence.</p>	<p><SENTENCE> (jazz, musician) returns documents that contain these words in the same sentence.</p>
IN	<p>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.</p>	<p>Chang <IN> author searches document zones named author for the word Chang.</p>

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 573](#).

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.	<code><CASE>Java OR <CASE>java</code> 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: <ul style="list-style-type: none">• WORD• WILDCARD• STEM• PHRASE• SENTENCE• PARAGRAPH	<code><PARAGRAPH><MANY>javascript <AND> vbscript.</code> You cannot use the MANY modifier with the following operators: <ul style="list-style-type: none">• AND• OR• ACCRUE• Relational operators
NOT	Excludes documents that contain the specified word or phrase. Use only with the AND and OR operators.	<code>Java <AND> programming <NOT> coffee</code> 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: <ul style="list-style-type: none">• PARAGRAPH• SENTENCE• NEAR/N Place the ORDER modifier before any operator.	<code><ORDER><PARAGRAPH> ("server", "Java")</code> 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, then 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 521](#). 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`, `URL`, and `CUSTOM` 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`

To explore how to use document fields to refine a search, consider the following database table, named `Calls`. This table has four fields and three records, as the following table shows:

<code>call_ID</code>	<code>Problem_Description</code>	<code>Short_Description</code>	<code>Product</code>
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 Administrator`).

The following table shows the relationship between the database column and `cfindex` attribute:

Database column	The <code>cfindex</code> attribute	Comment
<code>call_ID</code>	<code>key</code>	The primary key of a database table is often a key attribute.
<code>Problem_Description</code>	<code>body</code>	This column is the information to be indexed.
<code>Short_Description</code>	<code>title</code>	A short description is conceptually equivalent to a title, as in a running title of a journal article.
<code>Product</code>	<code>custom1</code>	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>
```

In a browser, the following retrieved results appear:

KEY	TITLE	CUSTOM1
3	File Open Problem	HomeSite
1	Bold Problem	HomeSite

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 cform 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:

Retrieving and Formatting Data	579
Building Dynamic Forms	607
Charting and Graphing Data	645
Using the Flash Remoting Service	673

CHAPTER 26

Retrieving and Formatting Data

This chapter explains how to use HTML forms to control the data displayed by a dynamic web page. It also describes how to populate an HTML table with query results and how to use ColdFusion functions to format and manipulate data.

Contents

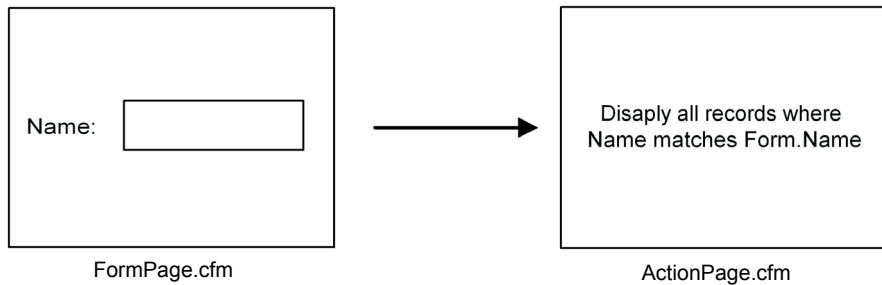
- Using forms to specify the data to retrieve 580
- Working with action pages..... 585
- Working with queries and data 589
- Returning results to the user 593
- Dynamically populating list boxes 597
- Creating dynamic check boxes and multiple-selection list boxes..... 599
- Validating form field data types 603

Using forms to specify the data to retrieve

In the examples in previous chapters, you retrieved all of the records from a database table using a SQL query. However, there are many instances when you want to retrieve data based on certain criteria. For example, you might want to retrieve records for everyone in a particular department, everyone in a particular town whose last name is Smith, or books by a certain author.

You can use forms in ColdFusion applications to allow users to specify what data they retrieve in a query. When you submit a form, you pass the variables to an associated page, called an **action page**, where some type of processing takes place.

The following figure shows a form, defined by `FormPage.cfm`, and its associated action page, `ActionPage.cfm`:



Note: Because forms are standard HTML, the syntax and examples that follow provide you with just enough detail to begin using ColdFusion. For information on using ColdFusion forms defined by the `cfform` tag, see [Chapter 27, “Building Dynamic Forms” on page 607](#).

HTML form tag syntax

Use the following syntax for the HTML form tag:

```
<form action="actionpage.cfm" method="post">  
  ...  
</form>
```

Attribute	Description
<code>action</code>	Specifies an action page to which you pass form variables for processing.
<code>method</code>	Specifies how the variables are submitted from the browser to the action page on the server. All ColdFusion forms must be submitted with an attribute setting of <code>method="post"</code> .

You can override the server request timeout (set on the ColdFusion Administrator Server Settings page) by adding a `RequestTimeout` parameter to the action page URL. Requests that take longer than the specified time are terminated. The following example specifies a request time-out of two minutes:

```
<form name="getReportCriteria"  
  action="runReport.cfm?RequestTimeout=120" method="post">
```

Form controls

Within the form, you describe the form controls needed to gather and submit user input. There are a variety of form controls types available. You select form control input types based on the type input you want to user to provide.

The following figure shows an example form containing different form controls:

The figure shows a form with the following elements:

- Text boxes:** Three input fields labeled "First Name:", "Last Name:", and "Salary:".
- Select box:** A dropdown menu labeled "City" with "Arlington" selected.
- Radio buttons:** Three radio buttons under the label "Department:" with options "Training", "Sales", and "Marketing".
- Check box:** A checkbox labeled "Contractor?" which is checked, with the text "Yes" next to it.
- Reset button:** A button labeled "Clear Form".
- Submit button:** A button labeled "Submit".

The following table shows the format of form control tags:

Control	Code
Text control	<code><input type="Text" name="ControlName" size="Value" maxlength="Value"></code>
Radio buttons	<code><input type="Radio" name="ControlName" value="Value1">DisplayName1 <input type="Radio" name="ControlName" value="Value2">DisplayName2 <input type="Radio" name="ControlName" value="Value3">DisplayName3</code>
List box	<code><select name="ControlName"> <option value="Value1">DisplayName1 <option value="Value2">DisplayName2 <option value="Value3">DisplayName3 </select></code>
Check box	<code><input type="Checkbox" name="ControlName" value="Yes No">Yes</code>
Reset button	<code><input type="Reset" name="ControlName" value="DisplayName"></code>
Submit button	<code><input type="Submit" name="ControlName" value="DisplayName"></code>

Use the following procedure to create the form in the previous figure.

To create a form:

- 1 Create a ColdFusion page with the following content:

```
<html>
<head>
<title>Input form</title>
</head>
<body>
<!-- define the action page in the form tag. The form variables will
      pass to this page when the form is submitted -->

<form action="actionpage.cfm" method="post">

<!-- text box -->
<p>
First Name: <input type="Text" name="FirstName" size="20" maxlength="35"><br>
Last Name: <input type="Text" name="LastName" size="20" maxlength="35"><br>
Salary: <input type="Text" name="Salary" size="10" maxlength="10">
</p>

<!-- list box -->
<p>
City
<select name="City">
  <option value="Arlington">Arlington
  <option value="Boston">Boston
  <option value="Cambridge">Cambridge
  <option value="Minneapolis">Minneapolis
  <option value="Seattle">Seattle
</select>
</p>

<!-- radio buttons -->
<p>
Department:<br>
<input type="radio" name="Department" value="Training">Training<br>
<input type="radio" name="Department" value="Sales">Sales<br>
<input type="radio" name="Department"
      value="Marketing">Marketing<br>
</p>

<!-- check box -->
<p>
Contractor? <input type="checkbox" name="Contractor"
      value="Yes" checked>Yes
</p>

<!-- reset button -->
<input type="Reset" name="ResetForm" value="Clear Form">
<!-- submit button -->
<input type="Submit" name="SubmitForm" value="Submit">

</form>
</body>
</html>
```


- 2 Save the page as formpage.cfm within the myapps directory under your web root directory.
- 3 View the form in a browser.
The form appears in the browser.

Do not click the Submit button yet. Remember that you need an action page in order to submit values; you create one later in this chapter in [“Creating action pages” on page 586](#).

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
<code><form action="actionpage.cfm" method="post"></code>	Gathers the information from this form using the Post method, and do something with it on the page actionpage.cfm.
<code><input type="Text" name="FirstName" size="20" maxlength="35"></code>	Creates a text box called FirstName where users can enter their first name. Makes it 20 characters wide, but allows input of up to 35 characters.
<code><input type="Text" name="LastName" size="20" maxlength="35"></code>	Creates a text box called LastName where users can enter their first name. Makes it 20 characters wide, but allows input of up to 35 characters.
<code><input type="Text" name="Salary" size="10" maxlength="10"></code>	Creates a text box called Salary where users can enter a salary to look for. Makes it 10 characters wide, and allows input of up to 10 characters.
<code><select name="City"> <option value="Arlington"> Arlington <option value="Boston">Boston <option value="Cambridge"> Cambridge <option value="Minneapolis"> Minneapolis <option value="Seattle">Seattle </select></code>	Creates a drop-down list box named City and populate it with the values “Arlington,” “Boston,” “Cambridge,” “Minneapolis,” and “Seattle.”
<code><input type="checkbox" name="Contractor" value="Yes" checked>Yes</code>	Creates a check box that allows users to specify whether they want to list employees who are contractors. Box selected by default.
<code><input type="Reset" name="ResetForm" value="Clear Form"></code>	Creates a reset button to allow users to clear the form. Puts the text Clear Form on the button.
<code><input type="Submit" name="SubmitForm" value="Submit"></code>	Creates a submit button to send the values that users enter to the action page for processing. Puts the text Submit on the button.

Form notes and considerations

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 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.
- All the data that you collect on a form is automatically passed as form variables to the associated action page.
- Check boxes, radio buttons, and multiple select boxes do not pass 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 if a variable exists on the action page, see [“Testing for a variable's existence” on page 587](#).
- You can dynamically populate drop-down lists using query data. For more information, see [“Dynamically populating list boxes” on page 597](#).

Working with action pages

A ColdFusion action page is just 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.

Because form variables extend beyond the local page—their scope is the action page—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.

Dynamically generating 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="CompanyInfo">
    SELECT  FirstName, LastName, Contract
    FROM    Employee
</cfquery>
```

But 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="CompanyInfo">
    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="CompanyInfo">
  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 433.

Creating action pages

Use the following procedure to create an action page for the page formpage.cfm 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="CompanyInfo">
  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 within the myapps directory.

- 3 View formpage.cfm 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 `actionpage.cfm` to fix the error, see [“Testing for a variable's existence” on page 587](#).

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
<pre><cfquery name="GetEmployees" datasource="CompanyInfo"></pre>	Queries the data source <code>CompanyInfo</code> and names the query <code>GetEmployees</code> .
<pre>SELECT FirstName, LastName, Salary FROM Employee WHERE LastName=<cfqueryparam value="#Form.LastName#" CFSQLType="CF_SQL_VARCHAR"></pre>	Retrieves the <code>FirstName</code> , <code>LastName</code> , and <code>Salary</code> fields from the <code>Employee</code> table, but only if the value of the <code>LastName</code> field matches what the user entered in the <code>LastName</code> text box in the form on <code>formpage.cfm</code> .
<pre><cfoutput query="GetEmployees"></pre>	Displays results of the <code>GetEmployees</code> query.
<pre>#FirstName# #LastName# #Salary#
</pre>	Displays the value of the <code>FirstName</code> , <code>LastName</code> , and <code>Salary</code> 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 <code>SELECT</code> statement, followed by a line break, until you run out of records.
<pre></cfoutput></pre>	Closes the <code>cfoutput</code> block.
<pre>
 <cfoutput>Contractor: #Form.Contractor# </cfoutput></pre>	Displays a blank line followed by the text <code>Contractor:</code> and the value of the form <code>Contractor</code> 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 if 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 provides a server-side mechanism for requiring users to enter data in fields.

To require entry in an input field, use a hidden field that has a `name` attribute composed of the field name and the suffix `"_required."` For example, to require that the user enter a value in the `FirstName` field, use the following syntax:

```
<input type="hidden" name="FirstName_required">
```

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 using the `value` attribute of the hidden field. For example, if you want the error message to read "You must enter your first name." use the following syntax:

```
<input type="hidden"
      name="FirstName_required"
      value="You must enter your first name.">
```

Form variable notes and considerations

When using form variables, keep the following guidelines in mind:

- A form variable's scope is the action page.
- Prefix form variables with "Form." when referencing them on the action page.
- Surround variable values with pound signs (#) for output.
- Variables for check boxes, radio buttons, and multiple select list boxes only get passed to the action page if you select an option. Text boxes, passwords, and textareas 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.

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 page `actionpage.cfm` in your editor.
- 2 Modify the page so that it appears as follows:

```
<html>
<head>
<title>Retrieving Employee Data Based on Criteia from Form</title>
</head>

<body>
<cfquery name="GetEmployees" datasource="CompanyInfo">
    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` within the `myapps` directory.
- 4 View `formpage.cfm` in your browser.
- 5 Enter `Smith` in the Last Name text box and submit the form.
- 6 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
<code><table></code>	Puts data into a table.
<code><tr> <th>First Name</th> <th>Last Name</th> <th>Salary</th> </tr></code>	In the first row of the table, includes three columns, with the headings: First Name, Last Name, and Salary.
<code><cfoutput query="GetEmployees"></code>	Gets ready to display the results of the <code>GetEmployees</code> query.
<code><tr> <td>#FirstName#</td> <td>#LastName#</td> <td>#Salary#</td> </tr></code>	Creates a new row in the table, with three columns. For a record, puts the value of the <code>FirstName</code> field, the value of the <code>LastName</code> field, and the value of the <code>Salary</code> field.
<code></cfoutput></code>	Keeps getting records that matches the criteria, and displays each row in a new table row until you run out of records.
<code></table></code>	End of table.

Formatting individual data items

You can format individual data items. For example, you can format the `Salary` field as a monetary value. To format the `Salary` using the dollar format, you use the CFML expression `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 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 521](#).

To build a more flexible search interface:

- 1 Open the ColdFusion page `actionpage.cfm` 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="CompanyInfo">
    SELECT Department.Dept_Name,
           Employee.FirstName,
           Employee.LastName,
           Employee.StartDate,
           Employee.Salary
    FROM Department, Employee
    WHERE Department.Dept_ID = Employee.Dept_ID
    <cfif IsDefined("Form.Department")>
        AND Department.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 `formpage.cfm` 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
<pre>SELECT Department.Dept_Name, Employee.FirstName, Employee.LastName, Employee.StartDate, Employee.Salary FROM Department, Employee WHERE Department.Dept_ID = Employee.Dept_ID</pre>	Retrieves the fields listed from the <code>Department</code> and <code>Employee</code> tables, joining the tables based on the <code>Dept_ID</code> field in each table.
<pre><cfif IsDefined("FORM.Department")> AND Department.Dept_Name = <cfqueryparam value="#Form.Department#" CFSQLType="CF_SQL_VARCHAR"> </cfif></pre>	If the user specified a department on the form, only retrieves records where the department name is the same as the one the user specified. You must use pound signs in the SQL <code>AND</code> statement to identify <code>Form.Department</code> as a ColdFusion variable, but not in the <code>IsDefined</code> function.
<pre><cfif Form.LastName IS NOT ""> AND Employee.LastName = <cfqueryparam value="#Form.LastName#" CFSQLType="CF_SQL_VARCHAR"> </cfif></pre>	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 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 where 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.

For more information on query variables, including `RecordCount`, see [Chapter 20, "Accessing and Retrieving Data"](#) on page 433.

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>  
</cfif>
```

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 page `actionpage.cfm`.
- 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="CompanyInfo">  
    SELECT Deptmt.Dept_Name,  
           Employee.FirstName,  
           Employee.LastName,  
           Employee.StartDate,  
           Employee.Salary  
    FROM Deptmt, Employee  
    WHERE Deptmt.Dept_ID = Employee.Dept_ID  
<cfif isdefined("Form.Department")>  
    AND Deptmt.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 output long-running requests to the browser before a ColdFusion page is fully processed. This allows you to 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 `cfflush` 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 generates since 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` to incrementally flush data as it becomes available. This format is particularly useful when a query responds slowly with large amounts of data.

Dynamically populating list boxes

In “Form controls” on page 581, you hard-coded a 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 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 file `formpage.cfm` in ColdFusion Studio.
- 2 Modify the file so that it appears as follows:

```
<html>
<head>
<title>Input form</title>
</head>
<body>
<cfquery name="GetDepartments" datasource="CompanyInfo">
SELECT DISTINCT Location
FROM Departmt
</cfquery>

<!-- Define the action page in the form tag.
The form variables will pass to this page
when the form is submitted -->

<form action="actionpage.cfm" method="post">

<!-- text box -->
<p>
First Name: <input type="Text" name="FirstName" size="20" maxlength="35"><br>
Last Name: <input type="Text" name="LastName" size="20" maxlength="35"><br>
Salary: <input type="Text" name="Salary" size="10" maxlength="10">
</p>

<!-- list box -->
City

<select name="City">
<cfoutput query="GetDepartments">
<option value="#GetDepartments.Location#">
#GetDepartments.Location#
</option>
</cfoutput>
</select>

<!-- radio buttons -->
<p>
Department:<br>
<input type="radio" name="Department" value="Training">Training<br>
```

```

<input type="radio" name="Department" value="Sales">Sales<br>
<input type="radio" name="Department" value="Marketing">Marketing<br>
<input type="radio" name="Department" value="HR">HR<br>
</p>

<!-- check box -->
<p>
Contractor? <input type="checkbox" name="Contractor" value="Yes" checked>Yes
</p>

<!-- reset button -->
<input type="reset" name="ResetForm" value="Clear Form">

<!-- submit button -->
<input type="submit" name="SubmitForm" value="Submit">
</form>
</body>
</html>

```

3 Save the page as formpage.cfm.

4 View formpage.cfm 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
<pre> <cfquery name="GetDepartments" datasource="CompanyInfo"> SELECT DISTINCT Location FROM Departmt </cfquery> </pre>	Get the locations of all departments in the Departmt table. The DISTINCT clause eliminates duplicate location names from the returned query results.
<pre> <select name="City"> <cfoutput query="GetDepartments"> <option value="#GetDepartments.Location#"> #GetDepartments.Location# </option> </cfoutput> </select> </pre>	Populate the City selection list from the Location column of the GetDepartments query. The control has one option for each row returned by the query.

Creating dynamic check boxes and multiple-selection list boxes

When an HTML form contains either a list of check boxes with the same name or a multiple-selection list box (that is, where 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 inputs.

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 correct this problem, make the form fields required, use Dynamic SQL, or use `cfparam` to establish a default value for the form field.

Check boxes

When you put a series of check boxes with the same name in an HTML 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:

```
<input type="checkbox"
      name="SelectedDepts"
      value="1">
  Training<br>

<input type="checkbox"
      name="SelectedDepts"
      value="2">
  Marketing<br>

<input type="checkbox"
      name="SelectedDepts"
      value="3">
  HR<br>

<input 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 SelectedDept form field is "2,4" and you use the SelectedDepts 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 containing string values (instead of numeric), you must modify the checkbox and cfquery syntax.

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:

```
<input type="checkbox"
    name="SelectedDepts"
    value="Training">
    Training<br>
```

```
<input type="checkbox"
    name="SelectedDepts"
    value="Marketing">
    Marketing<br>
```

```
<input type="checkbox"
    name="SelectedDepts"
    value="HR">
    HR<br>
```

```
<input 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,"")#)
```

Note: In SQL, all strings must be surrounded in single quotes. The ListQualify function returns a list with the specified qualifying character (here, a single quote) 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 Department
      WHERE Dept_Name IN ('Marketing','Sales')
```

Multiple selection lists

ColdFusion treats the result when a user selects multiple choices from a list box (HTML input type `select` with attribute `multiple`) just like 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 use multiple selection lists to search a database in the same way that you use check boxes.

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):

Select one or more companies to get more information on:

```
<select name="SelectDepts" multiple>
  <option value="1">Training
  <option value="2">Marketing
  <option value="3">HR
  <option value="4">Sales
</select>
```

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 Department
      WHERE Dept_ID IN (#form.SelectDepts#)
```

the following statement is sent to the database:

```
SELECT *
      FROM Department
      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):

```
<select name="SelectDepts" multiple>
  <option value="Training">Training
  <option value="Marketing">Marketing
  <option value="HR">HR
  <option value="Sales">Sales
</select>
```

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')
```

Validating form field data types

One limitation of standard HTML forms is that you cannot validate that users input the type or range of data you expect. ColdFusion enables you to do several types of data validation by adding hidden fields to forms.

The following table describes the hidden field suffixes that you can use to do validation:

Field suffix	Value attribute	Description
<code>_integer</code>	Custom error message	Verifies that the user entered a number. If the user enters a floating point value, it is rounded to an integer.
<code>_float</code>	Custom error message	Verifies that the user entered a number. Does not do any rounding of floating point values.
<code>_range</code>	MIN=MinValue MAX=MaxValue	Verifies that the numeric value entered is within the specified boundaries. You can specify one or both of the boundaries separated by a space.
<code>_date</code>	Custom error message	Verifies that the user entered a date and converts the date into the proper ODBC date format. Will accept most common date forms; for example, 9/1/98; Sept. 9, 1998.
<code>_time</code>	Custom error message	Verifies that the user correctly entered a time and converts the time to the proper ODBC time format.
<code>_eurodate</code>	Custom error message	Verifies that the user entered a date in a standard European date format and converts into the proper ODBC date format.

Note: Adding a validation rule to a field does not make it a required field. You need to add a separate `_required` hidden field if you want to ensure user entry.

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 illustrates another concept that might seem surprising. You can use the same ColdFusion page as both a form page and its action page. Because the only action is to display the values of the two variables that you enter, the action is on the same page as the form.

Using a single page for both the form and action provides the opportunity to show the use of the `IsDefined` function to check that data exists. This way, the form does not show any results until you submit the input.

To validate the data that users enter in the insert form:

- 1 Create a new page with the following text:

```
<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_required"
    value="You must enter a start date.">
  <input type="hidden"
    name="StartDate_date"
    value="Enter a valid date as the start date.">
  <input type="hidden"
    name="Salary_required"
    value="You must enter a salary.">
  <input type="hidden"
    name="Salary_float"
    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>

```

- 2 Save the file as `datatest.cfm`.
- 3 View the file in your browser, omit a field or enter invalid data, and click the Submit button.

When the user submits the form, ColdFusion scans the form fields to find any validation rules you specified. The rules are then used to analyze the user's input. If any of the input rules are violated, ColdFusion sends an error message to the user that explains the problem. The user then 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 dollar signs, these characters are automatically deleted from fields with `_integer`, `_float`, or `_range` rules before the form field is validated and the data is passed to the form's action page.

Reviewing the code

The following table describes the code and its function:

Code	Description
<pre><form action="datatest.cfm" method="post"></pre>	Gather the information from this form using the Post method, and do something with it on the page dataform.cfm, which is this page.
<pre><input type="hidden" name="StartDate_required" value="You must enter a start date."> <input type="hidden" name="StartDate_date" value="Enter a valid date as the start date."></pre>	Require input into the StartDate input field. If there is no input, display the error information "You must enter a start date." Require the input to be in a valid date format. If the input is not valid, display the error information "Enter a valid date as the start date."
<pre><input type="hidden" name="Salary_required" value="You must enter a salary."> <input type="hidden" name="Salary_float" value="The salary must be a number."></pre>	Require input into the Salary input field. If there is no input, display the error information "You must enter a salary." Require the input to be in a valid number. If it is not valid, display the error information "The salary must be a number."
<pre>Start Date: <input type="text" name="StartDate" size="16" maxLength="16">
</pre>	Create a text box called StartDate in which users can enter their starting date. Make it exactly 16 characters wide.
<pre>Salary: <input type="text" name="Salary" size="10" maxLength="10">
</pre>	Create a text box called Salary in which users can enter their salary. Make it exactly ten characters wide.
<pre><cfif isdefined("Form.StartDate")> <cfoutput> Start Date is: #DateFormat(Form.StartDate)#
 Salary is: #DollarFormat(Form.Salary)# </cfoutput> </cfif></pre>	Output 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. Use the DateFormat function to display the start date in the default date format. Use the DollarFormat function to display the salary with a dollar sign and commas.

CHAPTER 27

Building Dynamic Forms

This chapter describes how to use the `cform` tag to enrich your HTML forms with sophisticated graphical controls, including several Java applet-based controls. You can use these controls without writing a line of Java code.

Contents

- [Creating forms with the `cform` tag](#) 608
- [Building tree controls with `ctree`](#) 611
- [Building drop-down list boxes](#) 619
- [Building text input boxes](#) 620
- [Building slider bar controls](#) 621
- [Creating data grids with `cgrid`](#) 622
- [Embedding Java applets](#) 633
- [Input validation with `cform` controls](#) 637
- [Input validation with JavaScript](#) 642

Creating forms with the cfform tag

You already learned how to use HTML forms to gather user input (see [Chapter 26, “Retrieving and Formatting Data” on page 579](#)). This chapter shows you how to use the `cfform` tag to create dynamic forms in CFML. In addition to standard HTML form controls, the `cfform` tag allows you to create forms that contain the following controls:

- Text boxes in which you can specify the appearance, such as fonts and colors
- Text inputs that allow you to validate the data entered into the control
- Predefined ColdFusion Java applet based controls, including trees, sliders, and grids
- Custom Java applets that act as form elements

Most `cfform` controls offer input validation attributes that you can use to validate user entry, selection, or interaction. This means you do not have to write separate CFML code specifically for input validation, as you do in HTML forms.

Using HTML and cfform

ColdFusion dynamically generates HTML forms from `cfform` tags and passes to the browser any HTML code that it finds in the form. As a result, you can also do the following:

- You can use the `passthrough` attribute of the `cfform`, `cfinput`, and `cfselect` tags to enter any HTML attributes that are not explicitly allowed in these tags. The attribute values are passed through to the HTML generated by these form tags.
- You can replace your existing HTML form tags with `cfform` and your forms will work fine.
- ColdFusion passes to the action page of the `cfform` the variable `Form.fieldnames` which contains the names of the form fields submitted from the form.

The cfform controls

The following table describes the ColdFusion controls that you use in forms created using `cfform`. You can use these tags only inside a `cfform` tag.

Control	Description	For more information
<code>cfgrid</code>	Java applet-based control that creates a 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 cfgrid” on page 622.
<code>cfslider</code>	Java applet-based control that defines a slider.	“Building slider bar controls” on page 621.
<code>cfinput</code>	Places radio buttons, check boxes, text input boxes, and password entry boxes. Equivalent to the HTML <code>input</code> tag with the addition of input validation.	“Input validation with cfform controls” on page 637.

Control	Description	For more information
cf <code>tree</code>	Java applet-based controls that define a tree control and individual tree control items.	“Building tree controls with cf<code>tree</code>” on page 611.
cf <code>textInput</code>	Java applet-based control that defines a text input box.	“Building text input boxes” on page 620.
cf <code>select</code>	Drop-down list box (not a Java applet). Equivalent to the HTML <code>select</code> tag with the addition of input validation and data binding.	“Building drop-down list boxes” on page 619.
cf <code>applet</code>	Embed your own Java applets in the form.	“Embedding Java applets” on page 633.

Preserving input data with `preservedata`

The `cfform` attribute `preservedata` tells ColdFusion to continue displaying the data that a user entered in the 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 form’s action posts to the same ColdFusion page as the form itself, and the control names are the same.

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 using the `preservedata` attribute, follow these guidelines:

- In `cftree`, the `preservedata` attribute causes the tree to expand the tree 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 `cfgrid`. If you populate the control from a query, you must update the data source with the new data (typically by using `cfgridupdate`) before redisplaying the grid. The grid then displays the updated database information.

Browser considerations

The applet-based controls for `cfform`—`cfgrid`, `cfslider`, `cftextinput`, and `cftree`—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 install. Other browsers, particularly older versions of Netscape, require you to navigate some simple option screens.

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 "notsupported" message appears in place of the control.

You can use the `cfform` tag's `notsupported` attribute to specify an alternate error message.

Building tree controls with cftree

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 `cftree`, you use individual `cftreeitem` tags to populate the control. You can specify one of six built-in icons to represent individual items in the tree control, or supply a file path or URL to your GIF image.

Note: The `cftree` tag requires the client to download a Java applet. Downloading an applet takes time; therefore, using `cftree` can be slightly slower than using an HTML form element to retrieve the same information. In addition, browsers must be Java-enabled for `cftree` to work properly.

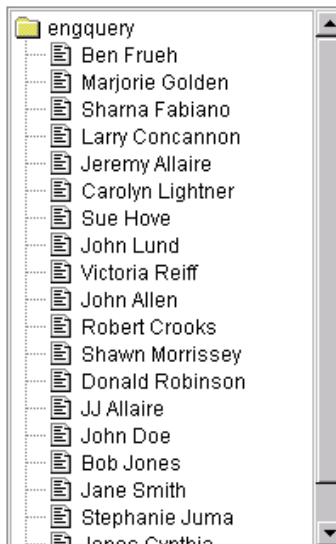
To create and populate a tree control from a query:

- 1 Create a ColdFusion page with the following content:

```
<cfquery name="engquery" datasource="CompanyInfo">
    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 code:



Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
<code><cftree name="tree1"</code>	Creates a tree and name it tree1.
<code>required="Yes"</code>	Specifies that a user must select an item in the tree.
<code>hscroll="No"</code>	Does not allow horizontal scrolling.
<code><cftreeitem value="FullName" query="engquery"</code>	Creates an item in the tree and put 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>queryasroot="Yes"</code>	Specifies the query name as the root level of the tree control.
<code>img="folder,document"</code>	Uses the images "folder" and "document" that ship with ColdFusion in the tree structure. When populating a <code>cftree</code> with data from a <code>cfquery</code> , 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 the 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="CompanyInfo">
    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="cd, folder">

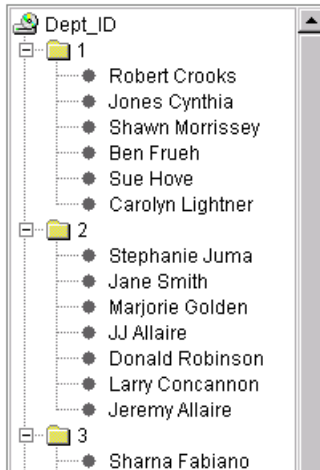
</cftree>
```

```

<br>
<br><input type="Submit" value="Submit">
</cform>

```

2 Save the page and view it in your browser.



Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
ORDER BY Dept_ID	Order the query results by department.
<cftreeitem value="Dept_ID,FullName"	Populate the tree with the Department ID, and under each department, the Full Name for each employee in the department.
queryasroot="Dept_ID"	Label the root "Dept_ID".
img="cd, folder">	Use the ColdFusion-supplied CD image for the root level and Folder image for the department IDs. The names are preceded by a bullet.

The `cftreeitem` comma-separated `img` and the `value` attributes both correspond to the tree level structure. If you leave out the `img` attribute, ColdFusion uses the folder image for all levels in the tree except the individual items, which have bullets.

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 `cform` 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 `cftree` to `Yes`; otherwise, the path value starts with the first node. If you specify a root name for a tree item using `queryasroot`, 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 = Dept_ID\3\John Allen
Form.tree1.node = John Allen
```

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` 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 `cftree` can be very complex. Knowing how to specify the relationship between multiple `cftreeitem` entries helps you handle the most complex of `cftree` constructs.

Creating a one-level tree control

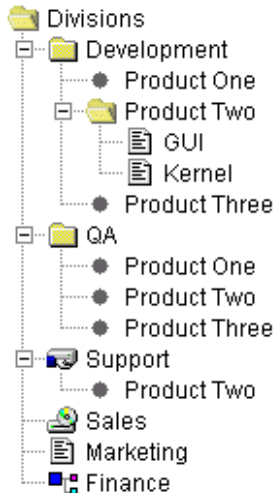
The following example consists of a single root and a number of individual items:

```
<cfquery name="deptquery" datasource="CompanyInfo">
    SELECT Dept_ID, FirstName + ' ' + LastName
    AS FullName
    FROM Employee
    ORDER BY Dept_ID
</cfquery>

<cform name="form1" action="submit.cfm">
    <cftree name="tree1">
        <cftreeitem value="FullName"
            query="deptquery"
            queryasroot="Department">
        </cftreeitem>
    </cftree>
    <br>
    <input type="submit" value="Submit">
</cform>
```


Creating a multilevel tree control

The following figure shows an example of a multilevel tree:



When populating a `cftree`, you create the multilevel structure of the tree by specifying a parent for each `cftreeitem` in the tree. The `parent` attribute of `cftreeitem` allows your `cftree` to show relationships between elements in the tree.

In this example, every `cftreeitem`, except the top level *Divisions*, specifies a parent. For example, the `cftreeitem` *Development* specifies a parent of *Divisions*.

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">
  <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">
  <cftreeitem value="QA"
    parent="Divisions" img="folder">
  <cftreeitem value="Product One"
    parent="QA">
  <cftreeitem value="Product Two" parent="QA">
```

```

<cfreeitem value="Product Three"
  parent="QA">
<cfreeitem value="Support"
  parent="Divisions" img="fixed">
<cfreeitem value="Product Two"
  parent="Support">
<cfreeitem value="Sales"
  parent="Divisions" img="cd">
<cfreeitem value="Marketing"
  parent="Divisions" img="document">
<cfreeitem value="Finance"
  parent="Divisions" img="element">
</cftree>

</cform>

```

Image names in a cftree

The default image displayed in a tree is a folder. However, you can use the `img` attribute of `cftreeitem` to specify a different image.

When you use the `img` attribute, ColdFusion displays the specified image beside the tree items. 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`. As a general rule, make the height of your custom images less than 20 pixels.

When populating a `cftree` with data from a `cfquery`, you can use the `img` attribute of `cftreeitem` 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:

- `cd`
- `computer`
- `document`
- `element`
- `folder`
- `floppy`
- `fixed`
- `remote`

Note: You can also control the tree appearance by using the `lookAndFeel` attribute to specify a Windows, Motif, or Metal look.

Embedding URLs in a cftree

The `href` attribute in the `cftreeitem` tag lets you designate tree items as links. To use this feature in a `cftree`, you define the destination of the link in the `href` attribute of `cftreeitem`. 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:

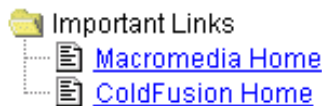
- 1 Create a ColdFusion page named `tree3.cfm` with the following contents:

```
<cform 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 Home"
    parent="Important Links"
    img="document"
    href="http://www.coldfusion.com">
</cftree>
</cform>
```

- 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
<code>href="http://www.macromedia.com"></code>	Makes the node of the tree a link.
<code>href="http://www.coldfusion.com"></code>	Makes the node of the tree a link. Although this example does not show it, <code>href</code> 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_value
```

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 `cform` tag with `cfselect` is similar to the HTML `select` tag. However, `cfselect` gives you more control over user inputs, provides error handling, and, most importantly, allows you to 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 `cfselect` is the same as for the HTML `option` tag.

When you populate a `cfselect` with data from a query, you only need to specify the name of the query that is supplying data for the `cfselect` and the query column name for each list element to display.

To populate a drop-down list box with query data using `cfselect`:

- 1 Create a ColdFusion page with the following content:

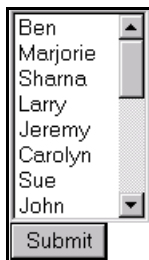
```
<cfquery name="getNames"
  datasource="CompanyInfo">
  SELECT * FROM Employee
</cfquery>

<cform 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">
</cform>
```

- 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 `cform` action attribute.

Building text input boxes

The `cftextinput` tag in a `cfform` tag is similar to the HTML `input type=text` tag or the CFML `cinput type=text` tag. With `cftextinput`, however, you can also specify font and alignment options, use the `validate` attribute to enable input validation using ColdFusion validation methods or your own JavaScript validation function, and use the `required` attribute to force the user to enter or change text.

The following example shows a basic `cftextinput` control. This example validates a date entry, which means that a user must enter a valid date in the form *mm/dd/yy* (the year can be up to four digits). For a complete list of validation formats, see *CFML Reference*.

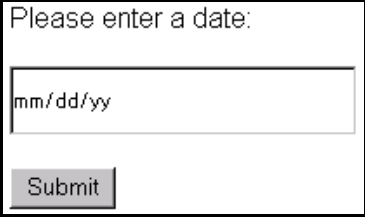
- 1 Create a ColdFusion page with the following content:

```
Please enter a date:<br>
<cfform name="Form1"
  action="submit.cfm">

  <cftextinput name="entertext"
    value="mm/dd/yy"
    maxlength="10"
    validate="date"
    width=100
    font="Trebuchet MS">
<br>
<br>
<input type="Submit"
  value="Submit">

</cfform>
```

- 2 Save the file as `textentry.cfm` and view it in your browser. The following figure shows the output of this code:



The screenshot shows a web browser window displaying the output of the ColdFusion code. At the top, the text "Please enter a date:" is displayed. Below this text is a text input field with a light gray border and a light gray background. The input field contains the placeholder text "mm/dd/yy". Below the input field is a "Submit" button with a light gray background and a dark gray border.

To get the value of the input text in the action page, use the variable `Form.textinput_name`; in this case, `Form.entertext`.

Building slider bar controls

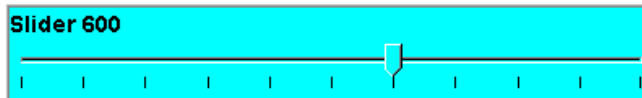
You can use the `cfslider` control in a `cfform` tag to create a slider control and define a wide range of formatting options for slider label text, label font name, size, boldface, italics, and color, as well as slider scale increments, range, positioning, tick marks, and behavior. Slider bars are useful because they are highly visual and users cannot enter invalid values.

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"  
        tickmarkmajor="True"  
        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 cfgrid

The `cfgrid` tag creates a `cform` grid control that resembles a spreadsheet table and can contain data populated from a `cfquery` or from other sources of data. As with other `cform` tags, `cfgrid` 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 do the following tasks with `cfgrid`:

- Sort data in the grid alphanumerically
- Update, insert, and delete data
- Display images in the grid

Users can sort the grid entries in ascending order by double-clicking any column header. Double-clicking again sorts the grid in descending order. 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 `cform` `action` attribute.

Just as the `cftree` tag uses `cftreeitem`, `cfgrid` 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 `cfgrid`, 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 *CFML Reference*.

Working with a data grid and entering data

The following figure shows an example grid created using the `cfgrid` tag:



	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

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.
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.

To populate a grid from a query:

- 1 Create a new ColdFusion page named `grid1.cfm` with the following contents:

```
<cfquery name="empdata" datasource="CompanyInfo">
    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><input 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 `cfgrid`.

- 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	Lighner	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

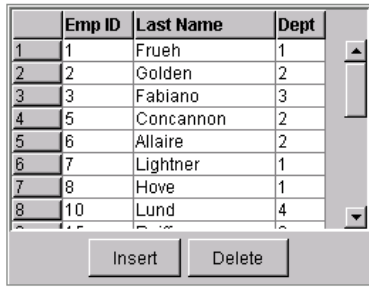
The following table describes the highlighted code and its function:

Code	Description
<code><cfgrid name="employee_grid" query="empdata"></code>	Create a grid named "employee_grid" and populate it with the results of the query "empdata". If you specify a <code>cfgrid</code> tag with a <code>query</code> attribute defined and no corresponding <code>cfgridcolumn</code> attributes, the grid contains all the columns in the query.
<code>selectmode="single"></code>	Allow the user to select only one cell. Other modes are row, column, and edit.
<code><cfgridcolumn name="Emp_ID"></code>	Put the contents of the <code>Emp_ID</code> column in the query results in the first column of the grid.
<code><cfgridcolumn name="LastName"></code>	Put the contents of the <code>LastName</code> column in the query results in the second column of the grid.
<code><cfgridcolumn name="Dept_ID"></code>	Put the contents of the <code>Dept_ID</code> 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.

To let users add or delete grid rows, you also have to set the `insert` or `delete` attributes in `cfgrid` to `Yes`. Setting `insert` or `delete` to `Yes` causes the `cfgrid` tag to display `insert` and `delete` buttons as part of the grid, as the following figure shows:



	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

Insert Delete

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 base on the form values returned by `cfgrid`.
- 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 `cfquery` 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

The `value`, `valuesDisplay`, and `valuesDelimiter` attributes of the `cfgridcolumn` tag let you 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 `type` attribute to control sorting order, to make the fields check boxes, or to display an image.
- Use the `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.
- While `cfgrid` 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 in *CFML Reference*.

How user edits are returned

ColdFusion creates the following arrays as Form variables to return edits to grid rows and cells:

Array reference	Description
<code>gridname.colname[change_index]</code>	Stores the new value of an edited cell.
<code>gridname.Original.colname [change_index]</code>	Stores the original value of the edited grid cell.
<code>gridname.RowStatus.Action [change_index]</code>	Stores the edit type made to the edited grid row: D for delete, I for insert, or U for update.

When a user selects and changes data in a row, ColdFusion creates arrays to store the following information for rows that are updated, inserted, or deleted:

- The original values for all columns
- The new column values
- The type of change

For example, the following arrays are created if you update a `cfgrid` called "mygrid" consisting of two displayable columns, (col1, col2) and one hidden column (col3):

```
Form.mygrid.col1[ change_index ]
Form.mygrid.col2[ change_index ]
Form.mygrid.col3[ change_index ]
Form.mygrid.original.col1[ change_index ]
Form.mygrid.original.col2[ change_index ]
Form.mygrid.original.col3[ change_index ]
Form.mygrid.RowStatus.Action[ change_index ]
```

The value of `change_index` increments for each row that changes, and does not indicate the specific row number. When the user updates data or inserts or deletes rows, the action page gets one array for each changed column, and the `RowStatus.Action` array. The action page does not get arrays for unchanged columns.

If the user makes a change to a single cell in col2, you can access the edit operation, the original cell value, and the edited cell value in the following arrays:

```
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]
```

Editing data in cfgrid

To enable grid editing, specify the `selectmode="edit"` attribute. When enabled, a user can edit cell data and insert or delete grid rows. When the user submits a `cfform` tag containing a `cfgrid` tag, data about changes to grid cells gets returned in the one-dimensional arrays described in the preceding section. You can reference these arrays as you would any other ColdFusion array.

Note: For code brevity, the following example handles only three of the fields in the Employee table. A more realistic example would include, at a minimum, all seven table fields. You might also consider hiding the contents of the `Emp_ID` column and automatically generating its value for new records, and displaying the Department name, from the `Departmt` table, in place of the Department ID.

To make the grid editable:

- 1 Create a new ColdFusion page with the following contents:

```
<cfquery name="empdata" datasource="CompanyInfo">
    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>
<input type="Submit" value="Submit">
</cfform>
```

2 Save the file as grid2.cfm and view it 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

Insert Delete

Submit

The following sections describe how to write `handle_grid.cfm` to process user edits to the grid.

Reviewing the code

The following table describes the code and its function:

Code	Description
<pre><cfgrid name="employee_grid" height=425 width=300 vspace=10 selectmode="edit" query="empdata" insert="Yes" delete="Yes"></pre>	Populates a <code>cfgrid</code> control with data from the <code>empdata</code> 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.
<pre><cfgridcolumn name="Emp_ID" header="Emp ID" width=50 headeralign="center" headerbold="Yes" select="No"></pre>	Creates a 50-pixel wide column for the data in the <code>Emp_ID</code> column of the data source. Center a header named <code>Emp ID</code> and make 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 <code>autoincrement</code> value.

Code	Description
<pre><cfgridcolumn name="LastName" header="Last Name" width=100 headeralign="center" headerbold="Yes"></pre>	Creates a 100-pixel wide column for the data in the LastName column of the data source. Center a header named Last Name and make it bold.
<pre><cfgridcolumn name="Dept_ID" header="Dept" width=35 headeralign="center" headerbold="Yes"></pre>	Creates a 35-pixel wide column for the data in the Dept_ID column of the data source. Center a header named Dept and make it bold.

Updating the database with cfgridupdate

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 particularly 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 `cfquery` provides. In particular, using the `cfgridupdate` tag, you can make the following changes:

- 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 cfgridupdate:

- 1 Create a file 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="CompanyInfo"
    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 `grid2.cfm` in your browser, make changes to the grid, and then submit them.

Note: To update a grid cell, modify the cell contents, then press Return.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
<code><cfgridupdate grid="employee_grid"</code>	Update the database from the Employee_grid grid.
<code>datasource="CompanyInfo"</code>	Update the CompanyInfo data source.
<code>tablename="Employee"</code>	Update the Employee table.

Updating the database with cfquery

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 cfquery:

- 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="CompanyInfo">
        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="CompanyInfo">
        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="CompanyInfo">
            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, then save this file as `handle_grid.cfm`.
- 3 View `grid2.cfm` 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
<pre> <cfif isdefined ("Form.employee_grid.rowstatus.action"> <cfloop index = "Counter" from = "1" to = #arraylen(Form.employee_grid.rowstatus.action)#> </pre>	<p>If there is an array of edit types, then change the table. Otherwise, do nothing. Loops through the remaining code once for each row to be changed. Counter is the common index into the arrays of change information for the row being changed.</p>
<pre> <cfoutput> The row action for #Counter# is: #Form.employee_grid.rowstatus.action[Counter]#
 </cfoutput> </pre>	<p>Displays the action code for this row: U, I, or D.</p>

Code	Description
<pre><cfif Form.employee_grid.rowstatus.action[counter] is "D"> <cfquery name="DeleteExistingEmployee" datasource="CompanyInfo"> DELETE FROM Employee WHERE Emp_ID=#Form.employee_grid.original.Emp_ID[Counter]# </cfquery></pre>	<p>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.</p>
<pre><cfelseif Form.employee_grid.rowstatus.action [counter] is "U"> <cfquery name="UpdateExistingEmployee" datasource="CompanyInfo"> UPDATE Employee SET LastName='#Form.employee_grid.LastName[Counter]#', Dept_ID=#Form.employee_grid.Dept_ID[Counter]# WHERE Emp_ID=#Form.employee_grid.original.Emp_ID[Counter]# </cfquery></pre>	<p>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.</p>
<pre><cfelseif Form.employee_grid.rowstatus.action[counter] is "I"> <cfquery name="InsertNewEmployee" datasource="CompanyInfo"> INSERT into Employee (LastName, Dept_ID) VALUES ('#Form.employee_grid.LastName[Counter]#', #Form.employee_grid.Dept_ID[Counter]#) </cfquery></pre>	<p>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 assumes that the DBMS automatically increments the Emp_ID primary key. If you use the Dbase version of the CompanyInfo database that is provided for UNIX installations, the record is inserted without an Emp_ID number.</p>
<pre></cfif> </cfloop> </cfif></pre>	<p>Closes the cfif tag used to select among deleting, updating, and inserting.</p> <p>Closes the loop used for each row to be changed.</p> <p>Closes the cfif tag that surrounds all the active code.</p>

Embedding Java applets

The `cfapplet` tag lets you embed Java applets either on a ColdFusion page or in a `cfform`. To use `cfapplet`, you must first register your Java applet using the ColdFusion Administrator Java Applets page (under Extensions on the Server tab). In the 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** Since `cfapplet` requires a form field `name` attribute, 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** Since the applet's interface is defined in the Administrator, 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 Administrator. You can override these values by specifying parameter value pairs in `cfapplet`.

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 Administrator.

To register a Java applet:

- 1 Open the ColdFusion Administrator by clicking on the Administrator icon in the ColdFusion 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/Registered Java Applet page appears.
- 4 Enter options for the following settings:

Setting	Description
Applet Name	Applet name.
Code	Name of the file that contains the applet subclass. Must be relative to the code base URL. The class extension is optional.

Setting	Description
Code Base	Base URL of the applet: directory that contains the applet components. The applet class files must be located within the web server root directory, such as <code>http://servername/classes</code> .
Archive	File name for the applet archive.
Method	Method name in the applet that returns a string value. You use the name in the NAME attribute of the <code>cfapplet</code> tag to populate a form variable with the method value. If the applet has no method, leave this field blank.
Height	Applet height, in pixels.
Width	Applet width, in pixels.
VSpace	Measurement, in pixels, for the space above and below the applet.
HSpace	Measurement, in pixels, for the space on each side of the applet.
Align	Applet alignment.
Not Supported Message	Message to display if the user's web browser does not support Java applets. To override this message, specify a different one in the <code>cfapplet</code> tag <code>notsupported</code> attribute.
Parameter Name	Name for a required applet parameter, typically provided by the applet.
Value	Default value for the parameter.

5 Click Submit.

Applet registration fields

The following table explains the applet registration fields:

Field	Description
Codebase	Enter the base URL of the applet: the directory that contains the applet components. The applet class files must be located within the web browser root directory; for example: <code>http://servername/classes</code>
Code	The name of the file that contains the compiled applet. The filename is relative to the code base URL. The *.class file extension is not required.
Method	Enter the name of a method in the applet that returns a string value. If you specify the method name in the <code>cfapplet</code> tag <code>name</code> attribute, the value returned by the method is available in the form's action page as <code>Form.name</code> . If the applet has no method, leave this field blank.
Height	Enter a measurement in pixels for the vertical space for the applet.
Width	Enter a measurement in pixels for the horizontal space for the applet.

Field	Description
Vspace	Enter a measurement in pixels for the space above and below the applet.
Hspace	Enter a measurement in pixels for the space on each side of the applet.
Align	Select the alignment.
Not Supported Message	This message is displayed by browsers that do not support Java applets. To override this message, you specify a different message in the <code>cfapplet notsupported</code> attribute.
Parameter Name	Enter a name for a required applet parameter. Your Java applet typically provides the parameter name needed to use the applet. Enter each parameter in a separate parameter field.
Value	For every parameter that you enter, define a default value. Your applet documentation provides guidelines on valid entries.

Using `cfapplet` 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 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 also override the values that you assigned to applet parameters in the ColdFusion Administrator by providing new values for any parameter. In order to override a parameter, you must have already defined the parameter and a default value for it in the ColdFusion Administrator Applets page, as follows:

```
<cfapplet appletSource="myapplet"
  name="applet1_var"
  Param1="registered parameter1"
  Param2="registered parameter2">
```

Handling form variables from an applet

The `cfapplet` tag requires you to specify a form variable name for the applet. This variable, referenced like other ColdFusion form variables, `Form.variable_name` holds the value the applet method returns when it is executed in the `cfform`.

Not all Java applets return values. For instance, many graphical widgets do not return a specific value; they do their flipping, spinning, fading, exploding, and that is all. For this kind of applet, the method field in the Administrator remains empty. Other applets, however, do have a method that returns a value. 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 Administrator.
- 2 Specify the method name in the `name` attribute of the `cfapplet` tag when you code your `cfform`.

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.

Input validation with cform controls

The `cfinput` and `cfinput` tags include the `validate` attributes, which lets you specify a valid data entry type for the control. You can validate user entries on the following data types:

Data type	Description
Date	Verifies US date entry in the form mm/dd/yyyy (where the year can have one through four digits).
Eurodate	Verifies valid European date entry in the form dd/mm/yyyy (where the year can have one through four digits).
Time	Verifies a time entry in the form hh:mm:ss.
Float	Verifies a floating point entry.
Integer	Verifies an integer entry.
Telephone	Verifies a telephone entry. You must enter telephone data as ###-###-####. You can replace the hyphen separator (-) with a blank. The area code and exchange must begin with a digit between 1 and 9.
Zipcode	(U.S. formats only) Number can be a five-digit or nine-digit zip in the form #####-####. You can replace the hyphen separator (-) with a blank.
Creditcard	Blanks and dashes are stripped and the number is verified using the mod10 algorithm.
Social_security_number	You must enter the number as ###-##-####. You can replace the hyphen separator (-) with a blank.
Regular_expression	Matches the input against a JavaScript regular expression pattern. You must use the <code>pattern</code> attribute to specify the regular expression. Any entry containing characters that matches the pattern is valid.

When you specify an input type in the `validate` attribute, ColdFusion tests for the specified input type when you submit the form, and submits form data only on a successful match. A successful form submission returns the value `True` and returns the value `False` if validation fails.

Validating with regular expressions

You can use **regular expressions** to match and validate the text that users enter in `cfinput` and `cfinput` 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 allow you to check input text for a wide variety of conditions. For example, if a date field must only contain dates between 1950 and 2050, you can create a regular expression that matches only numbers in that range. 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 133.

Special characters

Because special characters are the operators in regular expressions, in order to represent a special character as an ordinary one, you must precede it with a backslash. For example, use double 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 the 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 bracket 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]"`.
- You can use the following escape sequences to match specific characters or character classes:

Escape seq	Matches	Escape seq	Meaning
<code>[\b]</code>	Backspace	<code>\s</code>	Any of the following white space characters: space, tab, form feed, and line feed.
<code>\b</code>	A word boundary such as a space	<code>\S</code>	Any character except the white space characters matched by <code>\s</code>

Escape seq	Matches	Escape seq	Meaning
\B	A non-word 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_]
\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"
\n	Line feed	\octal	The character represented in the ASCII character table by the specified octal number
\r	Carriage return	\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 (+) 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 ({}) are used to indicate a range of occurrences of a regular expression, 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 syntax {,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 nth 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 the syntax you use for backreferencing in regular expressions:

```
(\b[A-Za-z+]) [ ]+\1
```

This code matches text that contains a word (specified by the \b word boundary special character and the [A-Za-z+]) followed by one or more spaces []+, followed by the first matched subexpression in parentheses. For example, it would match "is is, or "This is is", but not "This is".

Exact and partial matches

Entered data is normally valid if any of it matches the regular expression pattern. Often you might 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 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 DOS/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.

Expression	Description
<code>a{2,4}</code>	A string containing two to four occurrences of 'a': aa, aaa, aaaa; for example aardvark, but not automatic.
<code>(ba){2,}</code>	A string containing least two 'ba' pairs; for example Ali baba, but not Ali Baba.

Resources

An excellent reference on regular expressions is *Mastering Regular Expressions* by Jeffrey E.F. Friedl, published by O'Reilly & Associates, Inc.

Input validation with JavaScript

In addition to native ColdFusion input validation using the `validate` attribute of the `cfinput` and `cfinput` tags, the following tags support the `onvalidate` attribute, which lets you specify a JavaScript function to handle your `cfinput` input validation:

- `cfgrid`
- `cfinput`
- `cfslider`
- `cfinput`
- `cfinput`

ColdFusion passes the following arguments to the JavaScript function you specify in the `onvalidate` attribute:

- The form object
- The JavaScript input object corresponding to the tag whose value is being validated
- The value of the control to validate

For example, if you code 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>
```

Handling failed validation

The `onerror` attribute lets you specify a JavaScript function to execute if a 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 `validate` attribute, you can also use the `onerror` attribute to specify a JavaScript function handle validation errors. The following `cfinput` tags support the `onerror` attribute:

- `cfgrid`
- `cfinput`
- `cfselect`
- `cfslider`
- `cfinput`
- `cfinput`

ColdFusion passes the following JavaScript objects to the function in the `onerror` attribute:

- `form_object`

- input_object
- object_value
- error message text

Example: validating an e-mail address

The following example validates an e-mail entry. If the string is invalid, it displays a message box. If the address is valid, it redisplay the page. To be valid, the e-mail address must not be an empty string, contain an at sign (@) that is at least the second character, and contain a period (.) that is at least the fourth character.

To use JavaScript to validate form data:

- 1 Create a ColdFusion page with the following content:

```
<html>
<head>
  <title>JavaScript Validation</title>

  <script>
  <!--
function testbox(form, ctrl, value) {
  if (value == "" || value.indexOf('@', 1) == -1 ||
      value.indexOf('.', 3) == -1)
  {
    return (false);
  }
  else
  {
    return (true);
  }
}
  //-->
</script>

</head>

<body>
<h2>JavaScript validation test</h2>

<p>Please enter your email address:</p>
<cfform name="UpdateForm" preservedata="Yes"
  action="validjs.cfm" >

  <cfinput type="text"
    name="inputbox1"
    required="YES"
    onvalidate="testbox"
    message="Sorry, your entry is not a valid email address."
    size="15"
    maxlength="30">

  <input type="Submit" value=" Update... ">
</cfform>

</body>
```

</html>

- 2 Save the page as validjs.cfm.
- 3 View validjs.cfm in your browser.

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
<pre><script> <!-- function testbox(form) { Ctrl = Form.inputbox1; if (Ctrl.value == "" Ctrl.value.indexOf('@', 1) == -1 Ctrl.value.indexOf('.', 3) == -1) { return (false); } else { return (true); } } //--> </script></pre>	JavaScript code that tests for valid entry in the text box. The if statement checks to making sure that the field is not empty and contains an at sign (@) that at least the second character and a period (.) that is at least the fourth character.
<pre>onvalidate="testbox"</pre>	Calls the JavaScript testbox function to validate entries in this control.
<pre>message="Sorry, your entry is not a valid email address."</pre>	Displays a message if the validation function returns a false value.

CHAPTER 28

Charting and Graphing Data

This chapter explains how to use the `cfchart` tag to display charts and graphs. It describes ways that you can chart data and gives you the tools you need to create effective charts.

Contents

- [Creating a chart](#) 646
- [Administering charts](#)..... 649
- [Charting data](#) 650
- [Controlling chart appearance](#)..... 658
- [Linking charts to URLs](#) 667

Creating a chart

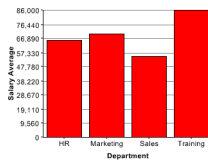
The ability to display data in a chart or graph can make data interpretation much easier. Rather than present a simple table of numeric data, you can display a bar, pie, line, or other applicable type of chart using colors, captions, and a two-dimensional or three-dimensional representation of your data.

The `cfchart` tag, along with the tags `cfchartseries` and `cfchartdata`, provide many different chart types. The attributes to these tags let you customize your chart appearance.

Chart types

You can create 11 types of charts in ColdFusion in two and three dimensions. The following figure shows a sample of each type of chart in two dimensions.

Note: Horizontal bar charts are bar charts rotated 90 degrees. In two dimensions, bar and cylinder charts appear the same, as do cone and pyramid charts.



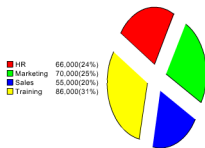
bar, cylinder, and horizontal bar



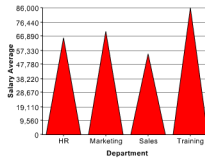
line



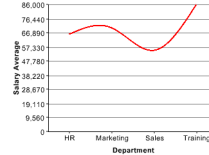
area



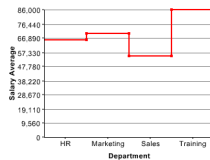
pie



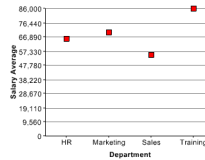
cone and pyramid



curve



step



scatter

Creating a basic chart

To create a chart, 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
<code>cfchart</code>	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 <code>cfchartseries</code> tag within the <code>cfchart</code> tag.
<code>cfchartseries</code>	Specifies a database query that supplies the data to the chart and/or one or more <code>cfchartdata</code> tags specifying individual data points. Specifies the chart type, colors for the chart, and other optional attributes.
<code>cfchartdata</code>	Optionally specifies individual data point to the <code>cfchartseries</code> tag.

The following shows the basic code you use to create a chart:

```
<cfchart
  <!-- optional attributes to cfchart -->
>

  <!-- one or more cfchartseries tags -->
  <cfchartseries
    type="type"
    <!-- optional attributes to cfchartseries -->
  />

  <cfchartseries
    type="type"
    <!-- optional attributes to cfchartseries -->
  >
    <!-- zero or more cfchartdata tags -->
    <cfchartdata
      value="number"
      <!-- optional attributes to cfchartdata -->
    >
  </cfchartseries>

</chart>
```

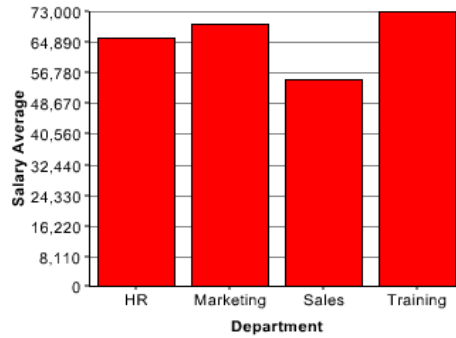
Often, you use these tags to chart the data stored in a ColdFusion query. If you have a query that contains average salary information by department, the following code displays a bar chart that shows the data in the query:

```
<cfchart
  xAxisTitle="Department"
  yAxisTitle="Salary Average"
>
  <cfchartseries
    type="bar"
    query="DataTable"
    valueColumn="AvgByDept"
```

```
        itemColumn="Dept_Name"  
    />  
</cfchart>
```

In this example, the data from the query column AvgByDept supplies the data for the y-axis, and the query column Dept_Name provides the data for the x-axis.

The resulting chart looks like the following:



Administering charts

Use the ColdFusion 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 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 inspects the cache to see if the chart has already been rendered. If so, ColdFusion loads the chart from the cache.

The following table describes the settings for the ColdFusion charting and graphing engine:

Option	Description
Cache Type	Set 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	Specify 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	Specify 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, specify the directory in which to store the generated charts.

Charting data

One of the most important considerations when you chart data is the way you supply the data to the `cfchart` tag. You can supply data in the following ways:

- Provide all the data in a single query using `cfchartseries` tags.
- Specify individual data points using `cfchartdata` tags.
- Combine data from a query with additional data points from `cfchartdata` tags.

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 a query

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.

You use the following attributes of the `cfchartseries` tag when working with queries:

Attribute	Description
<code>query</code>	The query that contains the data. You must also specify <code>valueColumn</code> and <code>itemColumn</code> .
<code>valueColumn</code>	The query column that contains the values to be charted.
<code>itemColumn</code>	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 horizontal bar charts, and in the legend in pie charts.

Using queries 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 461](#).

You can also take advantage of the ability to reference and modify query data dynamically. 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 CompanyInfo 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="CompanyInfo">
    SELECT Deptmt.Dept_Name,
           Employee.Salary
    FROM Deptmt, Employee
    WHERE Deptmt.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>
```

```
</cfchart>
```

```
<br>
```

```
</body>
```

```
</html>
```

2 Save the page as `chartdata.cfm` in `myapps` under the web root directory. For example, the directory path on Windows might be `C:\inetpub\wwwroot\myapps`.

3 Return to your browser and enter the following URL to view `chartdata.cfm`:

`http://127.0.0.1/myapps/chartdata.cfm`

The following figure appears:



Note: If a query contains two rows with the same value for the `itemColumn`, ColdFusion graphs the last row in the query for that value. For the previous 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
<pre><cfquery name="GetSalaries" datasource="CompanyInfo"> SELECT Deptmt.Dept_Name, Employee.Salary FROM Deptmt, Employee WHERE Deptmt.Dept_ID = Employee.Dept_ID </cfquery></pre>	Query the <code>CompanyInfo</code> database to get the <code>Dept_Name</code> and <code>Salary</code> for each employee. Because the <code>Dept_Name</code> is in the <code>Deptmt</code> table and the <code>Salary</code> is in the <code>Employee</code> table, you need a table join in the <code>WHERE</code> clause. The raw results of this query could be used elsewhere on the page.
<pre><cfquery dbtype = "query" name = "DeptSalaries"> SELECT Dept_Name, AVG(Salary) AS AvgByDept FROM GetSalaries GROUP BY Dept_Name </cfquery></pre>	Generate a new query from the <code>GetSalaries</code> query. Use the <code>AVG</code> aggregating function to get statistical data on the employees. Use the <code>GROUP BY</code> statement to ensure that there is only one row for each department.

Code	Description
<pre><cfloop index="i" from="1" to="#DeptSalaries.RecordCount#"> <cfset DeptSalaries.AvgByDept[i]= Round(DeptSalaries.AvgByDept[i] /1000)*1000> </cfloop></pre>	<p>Loop through all the rows in DeptSalaries query and round the salary data to the nearest thousand. This loop uses the query variable RecordCount to get the number of rows and changes the contents of the query object directly.</p>
<pre><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></pre>	<p>Create a bar chart using the data from the AvgByDept column of the DeptSalaries query. Label the bars with the Department names.</p>

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>
```

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 description for the legend.

Note: If two data points have the same item name, ColdFusion graphs 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 horizontal bar charts, and in the legend in pie charts.

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 `cfchart` to sort data alphabetically along the x-axis.

One use of combining queries and data points could be if the database is missing data for one department, you can add the information manually. The following example adds data for the Facilities and Documentation departments to the salary data obtained from the query shown in the previous section:

```
<cfchart
    <cfchartseries
        type="bar"
        query="DataTable"
        itemColumn="Dept_Name"
        valueColumn="AvgByDept"
    >
        <cfchartdata item="Facilities" value="35000">
        <cfchartdata item="Documentation" value="725000">
    </cfchartseries>
</cfchart>
```

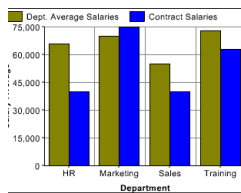
Charting multiple data collections

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.

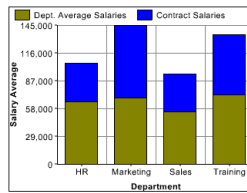
To combine multiple data series into a single chart, insert multiple `cfchartseries` tags within a single `cfchart` tag. You control how the multiple data collections are charted using the `seriesPlacement` attribute of the `cfchart` tag. Using this attribute, you can specify the following options:

- `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.

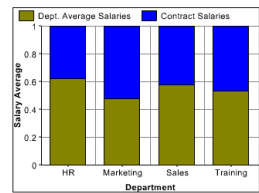
The following figure shows these options for combining two bar charts:



clustered

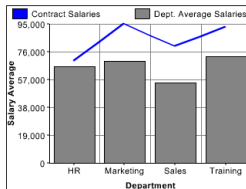


stacked



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 following example creates the previous figure showing 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 a bar chart is specified 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 and a line chart:

- 1 Open `chartdata.cfm` 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 page as `chart2queries.cfm` in `myapps` under the web root directory. For example, the directory path on Windows might be `C:\inetpub\wwwroot\myapps`.
- 4 Return to your browser and enter the following URL to view `chart2queries.cfm`:
`http://127.0.0.1/myapps/chart2queries.cfm`

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 containing 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 movie (.swf file), or as a JPG or PNG image file. If you save the image as a Flash movie, you can pass the variable back to a Flash client using ColdFusion Flash Remoting. For more information, see [Chapter 29, “Using the Flash Remoting Service” on page 673](#).

Note: If you write the chart to a JPG or PNG file, mouseover tips and URLs embedded in the chart for data drill-down will not work when you redisplay the image from the file. However, if you save the image a Flash movie, both tips and drill-down URLs will work. For more information on data drill-down, see [“Linking charts to URLs” on page 667](#).

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"
  charset="ISO-8859-1"
  file="c:\inetpub\wwwroot\charts\vehicle.jpg"
  output="#myChart#">


```

- 2 Save the page as chartToFile.cfm in myapps under the web root directory.
- 3 Return to your browser and enter the following URL to view chartToFile.cfm:
<http://127.0.0.1/myapps/chartToFile.cfm>

The chart is saved to disk as c:\inetpub\wwwroot\charts\vehicle.jpg

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
<pre><cfchart name="myChart" format="jpg"></pre>	Define a chart written to the variable myChart using the JPG format.
<pre><cffile action="WRITE" charset="ISO-8859-1" file= "c:\inetpub\wwwroot\charts\vehicle.jpg" output="#myChart#"></pre>	Use <code>cffile</code> to write the chart to a file. You must specify a charset of ISO-8859-1 when writing binary chart data to a file.
<pre></pre>	Use the HTML <code>img</code> tag to display the chart.

Controlling chart appearance

Use the `cfchart` and `cfchartseries` tags to customize the appearance of your charts.

Common chart characteristics

You can optionally specify the following characteristics to `cfchart` on all types of charts:

Chart characteristic	Attributes used	Description
File type	<code>format</code>	Whether to send the chart to the user as a JPG, PNG, or Flash Movie (.swf) file. Flash is the default format.
Dimensions	<code>chartWidth</code> <code>chartHeight</code>	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.
Column labels	<code>sortXAxis</code>	Specifies to display the column labels along the x-axis in alphabetical order. The default is no.
Foreground and background color	<code>foregroundColor</code> <code>dataBackgroundColor</code> <code>backgroundColor</code>	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 or use any valid HTML color format. If you use the numeric format, you must use double pound signs, for example, blue or <code>##FF33CC</code> . For the complete list of colors, see <i>Administering ColdFusion MX</i> .
Border	<code>showBorder</code>	Specifies to draw a border around the chart. The border color is the same as specified by the <code>foregroundColor</code> attribute. Default is no.
Labels	<code>font</code> <code>fontSize</code> <code>fontBold</code> <code>fontItalic</code> <code>labelFormat</code> <code>xAxisTitle</code> <code>yAxisTitle</code>	<code>font</code> specifies the font for all text. Default is Arial. If you are using a double-byte character set on UNIX, or using a double-byte character set on Windows with a file type of Flash, you must specify <code>ArialUnicodeMs</code> as the font. <code>fontSize</code> specifies an Integer font size used for all text. Default is 11. <code>fontBold</code> specifies to display all text as bold. Default is no. <code>fontItalic</code> specifies to display all text as italic. Default is no. <code>labelFormat</code> specifies the format of the y-axis labels, number, currency, percent, or date. Default is number. <code>xAxisTitle</code> and <code>yAxisTitle</code> specify the title for each axis.

Chart characteristic	Attributes used	Description
3-D Appearance	show3D xOffset yOffset	show3D displays the chart in 3-D. Default is no. xOffset and yOffset specify the amount to which the chart should be rotated on a horizontal axis (xOffset) or vertical axis (yOffset). 0 is flat (no rotation), -1 and 1 are for a full 90 degree rotation left (-1) or right (1). Default is .1
Rotation	rotated	Rotates the entire chart 90 degrees. Set to yes to create a horizontal chart, such as a horizontal bar chart. Default is no.
Multiple series	showLegend seriesPlacement	showLegend specifies to display the chart's legend when the chart contains more than one series of data. Default is yes. seriesPlacement 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.
Tips	tipStyle tipBGColor	tipStyle specifies to display a small popup window that shows information about the chart element pointed to by the cursor. Options are none, mousedown, or mouseover. Default is mouseover. tipBGColor specifies the background color of the tip window for Flash format only. Default is white.
Markers	showMarkers markerSize	showMarkers specifies to show markers at the data points for 2-D line, curve, and scatter charts. Default is yes. markerSize specifies an integer number of pixels for the marker size. ColdFusion determines default.

You can also use the `cfchartseries` tag to specify attributes of chart appearance. The following table describes these attributes:

Chart characteristic	Attributes used	Description
Multiple series	<code>seriesLabel</code> <code>seriesColor</code>	<code>seriesLabel</code> specifies the text displayed for the series label. <code>seriesColor</code> 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 <code>colorList</code> attribute.
Paint	<code>paintStyle</code>	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. Default is solid.
Data markers	<code>markerStyle</code>	For line, curve, and scatter charts, specifies the shape used to mark the data point. Supported for 2-dimensional charts. Default is rectangle.

Setting x-axis and y-axis characteristics

You can specify the following additional characteristics to control the look of the x-axis and y-axis of charts, except for pie charts:

Chart characteristic	Attributes used	Description
Value axis	<code>scaleFrom</code> <code>scaleTo</code>	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.
Grid lines	<code>showXGridlines</code> <code>showYGridlines</code> <code>gridLines</code>	<code>showXGridlines</code> and <code>showYGridlines</code> specify to display x-axis and y-axis grid lines. Default no for x-axis gridlines, and yes for y-axis gridlines. <code>gridLines</code> 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 <code>cfchart</code> tag displays horizontal grid lines only. A value of 0 (the default) means no grid lines.

Creating a bar chart

The example in the following procedure adds a title to the bar chart and changes its appearance from the default, flat look, to a 3-D look. It adds grid lines, sets the maximum y-axis value to 100,000, 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
  scaleTo = 100000
  fontSize=16
  gridLines = 4
  show3D="yes"
  >

  <cfchartseries
    type="bar"
    query="DeptSalaries"
    valueColumn="AvgByDept"
    itemColumn="Dept_Name"
  />

</cfchart>
```

- 3 Save the file.
- 4 Return to your browser and enter the following URL to view `chartdata.cfm`:
<http://127.0.0.1/myapps/chartdata.cfm>

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
<code>scaleTo = 100000</code>	Set the maximum value of the vertical axis to 100000. The minimum value is the default, 0.
<code>fontSize=16</code>	Make the point size of the labels 16 points.
<code>gridLines = 4</code>	Display four grid lines between the top and bottom of the chart.
<code>show3D = "yes"</code>	Show the chart in 3-D.

Setting pie chart characteristics

You can specify the following additional characteristics for pie charts:

Chart characteristic	Attributes used	Description
Slice style (<code>cfchart</code> tag)	<code>pieSliceStyle</code>	Display pie chart as solid or sliced. Default is sliced.
Data point colors (<code>cfchartseries</code> tag)	<code>colorList</code>	<p>A comma-separated list of colors to use for each pie slice.</p> <p>You can specify 16 color names or use any valid HTML color format. If you use the numeric format, you must use double pound signs, for example, blue or <code>##FF33CC</code>. For the complete list of colors, see <i>Administering ColdFusion MX</i>.</p> <p>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.</p>

The example in the following procedure adds a pie chart to the page.

To create a pie chart:

- 1 Open `chartdata.cfm` 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"
>
```



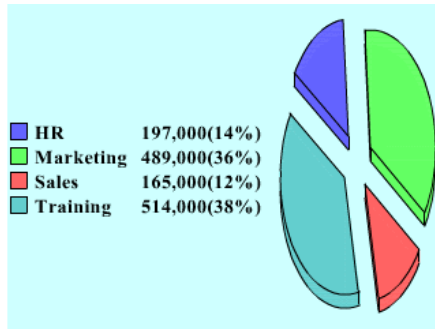
```

    <cfchartseries
      type="pie"
      query="DeptSalaries"
      valueColumn="SumByDept"
      itemColumn="Dept_Name"
      colorlist="##6666FF,##66FF66,##FF6666,##66CCCC"
    />
  </cfchart>
<br>

```

- 4 Save the file.
- 5 Return to your browser and enter the following URL to view chartdata.cfm:
<http://127.0.0.1/myapps/chartdata.cfm>

The following figure appears:



Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
<code>SUM(Salary) AS SumByDept,</code>	In the DeptSalaries query, add a SUM aggregation function to get the sum of all salaries per department.
<code><cfset DeptSalaries.SumByDept[i]= Round(DeptSalaries.SumByDept[i]/ 1000)*1000></code>	In the cfloop tag, round the salary sums to the nearest thousand.

Code	Description
<pre><cfchart tipStyle="mousedown" font="Times" fontBold="yes" backgroundColor = "##CCFFFF" show3D="yes" ></pre>	<p>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 3-D.</p>
<pre><cfchartseries type="pie" query="DeptSalaries" valueColumn="SumByDept" itemColumn="Dept_Name" colorlist= "##6666FF,##66FF66,##FF6666,##66CCCC" /></pre>	<p>Create a pie chart using the SumByDept salary sum values from the DeptSalares query.</p> <p>Use the contents of the Dept_Name column for the item labels displayed in the chart legend.</p> <p>Get the pie slice colors from a custom list, which uses hexadecimal color numbers. The double pound signs prevent ColdFusion from trying to interpret the color data as variable names.</p>

Creating an area chart

The example in the following procedure adds an area chart showing 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 `chartdata.cfm` your editor.
- 2 Edit the `GetSalaries` query so that it appears as follows:

```
<!-- Get the raw data from the database. -->
<cfquery name="GetSalaries" datasource="CompanyInfo">
  SELECT Deptmt.Dept_Name,
         Employee.StartDate,
         Employee.Salary
  FROM Deptmt, Employee
  WHERE Deptmt.Dept_ID = Employee.Dept_ID
</cfquery>
```

- 3 Add the following code before the `html` tag:

```
<!-- Convert start date to start year. -->
<!-- You must explicitly 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 Return to your browser and enter the following URL to view `chartdata.cfm`:
<http://127.0.0.1/myapps/chartdata.cfm>

Reviewing the code

The following table describes the highlighted code and its function:

Code	Description
Employee.StartDate,	Add the employee start date to the data in the GetSalaries query.
<pre> <cfloop index="i" from="1" to="#GetSalaries.RecordCount#"> <cfset GetSalaries.StartDate[i]= NumberFormat(DatePart("yyy", GetSalaries.StartDate[i]), "9999")> </cfloop> </pre>	Use a <code>cfloop</code> tag to extract the year of hire from each employee's hire data, and convert the result to a four-digit number.
<pre> <cfquery dbtype = "query" name = "HireSalaries"> SELECT StartDate, AVG(Salary) AS AvgByStart FROM GetSalaries GROUP BY StartDate </cfquery> </pre>	Create a second query from the GetSalaries query. This query contains the average salary for each start year.

Code	Description
<pre><cfloop index="i" from="1" to="#HireSalaries.RecordCount#"> <cfset HireSalaries.AvgByStart[i] =Round(HireSalaries.AvgByStart[i] /1000)*1000> </cfloop></pre>	Round the salaries to the nearest thousand.
<pre><cfchart chartWidth=400 BackgroundColor="###FFFFFF00" show3D="yes" > <cfchartseries type="area" query="HireSalaries" valueColumn="AvgByStart" itemColumn="StartDate" /> </cfchart></pre>	<p>Create a line chart using the HireSalaries query. Chart the average salaries against the start date.</p> <p>Limit the chart width to 400 pixels, show the chart in 3-D, and set the background color to white.</p>

Setting curve chart characteristics

Curves 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, then reference them in your application pages. For information on creating offline charts, see [“Writing a chart to a variable” on page 656](#).

Linking charts to URLs

ColdFusion provides a data drill-down capability with charts. This means you can click on an area of a chart, both the data and the legend areas, 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, define a chart that opens the page `moreinfo.cfm` when a user clicks on the chart using the following code:

```
<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 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 code the `url` attribute as follows:

```
url="moreinfo.cfm?Series=$SERIESLABEL$&Item=$ITEMLABEL$&Value=$VALUE$"
```

The variables are not enclosed in `#` signs like ordinary ColdFusion variables. They are enclosed in dollar signs. Clicking on a chart that uses this `url` attribute value 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 670](#).

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 departments 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="CompanyInfo">
    SELECT Deptmt.Dept_Name,
           Employee.FirstName,
           Employee.LastName,
           Employee.StartDate,
           Employee.Salary,
           Employee.Contract
    FROM Deptmt, Employee
    WHERE Deptmt.Dept_Name = '#URL.Item#'
    AND Deptmt.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 `myapps` under the web root directory.

Reviewing the code

The following table describes the code and its function:

Code	Description
<pre><cfquery name="GetSalaryDetails" datasource="CompanyInfo"> SELECT Deptmt.Dept_Name, Employee.FirstName, Employee.LastName, Employee.StartDate, Employee.Salary, Employee.Contract FROM Deptmt, Employee WHERE Deptmt.Dept_Name = '#URL.Item#' AND Deptmt.Dept_ID = Employee.Dept_ID ORDER BY Employee.LastName, Employee.Firstname </cfquery></pre>	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.
<pre><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></pre>	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="#CCCCCC"
    show3D="yes"
    url="Salary_Details.cfm?Item=$ITEMLABEL$"
>

<cfchartseries
    type="pie"
    query="DeptSalaries"
    valueColumn="SumByDept"
    itemColumn="Dept_Name"
```

```

        colorlist="###6666FF,###66FF66,###FF6666,###66CCCC"
    />
</cfchart>

```

- 3 Save the file.
- 4 Return to your browser and enter the following URL to view chartdata.cfm:
http://127.0.0.1/myapps/chartdata.cfm
- 5 Click the slices of the pie chart to request Salary_details.cfm 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
<pre> url= "Salary_Details.cfm?Item=\$ITEMLABEL\$" </pre>	<p>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 containing the department name of the selected wedge.</p>

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.

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 myapps under the web root directory.

- 3 Return to your browser and enter the following URL to view `chartdata_withJS.cfm`:
`http://127.0.0.1/myapps/chartdata_withJS.cfm`
- 4 Click the slices of the pie chart to display the pop-up window.

CHAPTER 29

Using the Flash Remoting Service

Using the Macromedia Flash Remoting service of Macromedia ColdFusion MX, ColdFusion developers can work together with Macromedia Flash MX designers to build dynamic Flash user interfaces for ColdFusion applications.

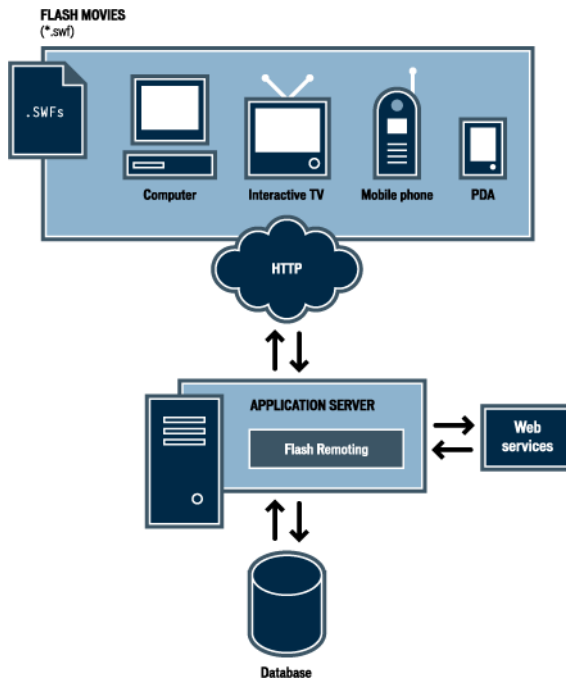
Contents

- [About using the Flash Remoting service with ColdFusion](#) 674
- [Using the Flash Remoting service with ColdFusion pages.....](#) 675
- [Using Flash with ColdFusion components.....](#) 680
- [Using the Flash Remoting service with server-side ActionScript.....](#) 682
- [Using the Flash Remoting service with ColdFusion Java objects](#) 683
- [Handling errors with ColdFusion and Flash](#) 684

About using the Flash Remoting service with ColdFusion

Using the Flash Remoting service of ColdFusion MX, ColdFusion developers can work together with Macromedia Flash MX designers to build Flash UIs for ColdFusion applications. Building Flash UIs requires the separation of UI code from business logic code. User interface controls are built in Flash MX, while business logic is built in ColdFusion.

The following figure displays a simplified representation of the relationship between Flash and ColdFusion:



Planning your Flash application

When planning ColdFusion application development with Flash UIs, remember the importance of separating display code from business logic. Separating display code, such as HTML, from business logic, such as CFML, enables your ColdFusion applications to interact with multiple client types, such as Flash movies, web browsers, and web services. Building ColdFusion applications for multiple clients means that 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. 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 natively support Flash interaction. In addition, you can use the ColdFusion

server-side ActionScript functionality, which lets you query databases and perform HTTP operations in ActionScript files on the server. The public methods of Java objects are also available to the Flash Remoting service.

The remaining sections in this chapter explain developing Flash applications with ColdFusion.

Using the Flash Remoting service with ColdFusion pages

When building a ColdFusion page that interacts with Flash movies, the directory name that contains the ColdFusion pages translates to the Flash service name in ActionScript. The individual ColdFusion page names contained in that directory translate to service functions in ActionScript.

In your CFML, you use the Flash variable scope to access parameters passed from Flash movies and return values to Flash movies. To access parameters passed from Flash movies, you use the parameter name appended to the `Flash` variable or the `Flash.Params` array. To return values to the Flash application, use the `Flash.Result` variable. To set an increment value for records 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
<code>Flash.Params</code>	A structure containing the parameters passed from the Flash movie.	See “Accessing parameters passed from Flash” on page 676.
<code>Flash.Result</code>	The variable returned to the Flash movie that called the function.	See “Returning results to Flash” on page 677.
<code>Flash.Pagesize</code>	The number of records returned at a time to Flash.	See “Returning records in increments to Flash” on page 678.

In addition, the following table compares the ColdFusion data types and their ActionScript equivalents:

ActionScript data type	ColdFusion MX data type
Number (primitive data type)	Number
boolean (primitive data type)	boolean
String	String
ActionScript (AS) object	Structure
AS Object (as the only argument passed to a service function)	Arguments to the service function. ColdFusion pages (.cfm): <code>flash</code> variable scope, ColdFusion components (.cfc): named arguments
null	null (ASC returns 0, which translates to not defined)
undefined	null (ASC returns 0, which translates to not defined)

ActionScript data type	ColdFusion MX data type
Ordered array	Array
Named array	Struct
Date object	Date
XML object	XML document
RecordSet	Query object

Accessing parameters passed from Flash

To access variables passed from Flash movies, 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 describes the collections and examples:

Collection	ActionScript example	Notes
Strict array	<pre>var myArray = new Array(); myArray[1] = "one"; myArray[2] = "two"; myService.myMethod(myArray);</pre>	The Flash Remoting service converts the array argument to a ColdFusion MX array. All CFML array operations work as expected.
Named or associative array	<pre>var myStruct = new Array(); myStruct["one"] = "banana"; myStruct["two"] = "orange";</pre>	In ActionScript, named array keys are not case-sensitive.

Collection	ActionScript example	Notes
Mixed array	<pre>var myMixedArray = new Array(); myMixedArray["one"] = 1; myMixedArray[2] = true;</pre>	<p>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. The following ColdFusion component example throws an exception:</p> <pre><cfargument name="ca" type="struct"> <cfreturn ca.2></pre> <p>The following ColdFusion component example does not throw an exception:</p> <pre><cfargument name="ca" type="struct"> <cfreturn ca["2"]></pre>
Using an ActionScript object initializer for named arguments	<pre>myService.myMethod({ x:1, Y:2, z:3 });</pre>	<p>This notation provides a convenient way of passing named arguments to a ColdFusion MX-based Flash Remoting service. You can access these arguments in ColdFusion pages as members of the Flash scope, or as normal named arguments of a ColdFusion component function</p>

The `Flash.Params` array retains the order of the parameters as they were passed to the function. You use standard structure name syntax to reference the parameters; for example:

```
<cfquery name="flashQuery" datasource="exampleapps" dbtype="ODBC">
    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 array. If the parameters were passed as an ordered array from Flash, you use standard structure name syntax; for example:

```
<cfset flash.result = "Variable 1:#{Flash.params[1]}#, Variable 2:
    #{Flash.params[2]}#">
```

In this ActionScript example, notice that ActionScript starts the array index at zero. ColdFusion array indexes start at one.

Returning results to Flash

In ColdFusion pages, only the value of `Flash.Result` variable is returned to the Flash application. For more information about supported data types between ColdFusion and Flash, see the data type table in [“Using the Flash Remoting service with ColdFusion pages” on page 675](#). The following procedure creates the service function `helloWorld`, which returns a structure containing simple messages to the Flash application.

To create a ColdFusion page that passes a structure to Flash:

- 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">
```
- 4 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.
- 5 Save the file.

Remember, the directory name is used the service address, and the `helloWorld.cfm` file is a method of the `helloExamples` Flash Remoting service. The following ActionScript example calls the `helloWorld` ColdFusion page:

```
include "NetServices.as"
NetServices.setDefaultGatewayUrl("http://localhost:8500/flasheservices/gateway");
gatewayConnection = NetServices.createGatewayConnection();
CFMService = gatewayConnection.getService("helloExamples", this);
CFMService.helloWorld();
```

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 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 lets you minimize the time that Flash application waits for the application server data to load.

To create a ColdFusion page that returns an 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="ExampleApps">
    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 default is set to 10. Next, a `cfquery` statement queries the database. After that, the `pagesize` variable is assigned into the `Flash.Pagesize` variable. Finally, the query results are assigned into the `Flash.Result` variable, which is returned to Flash.

3 Save your work.

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`. For example:

```
include "NetServices.as"
NetServices.setDefaultGatewayUrl("http://localhost:8500/flashservices/gateway");
gatewayConnection = NetServices.createGatewayConnection();
CFMService = gatewayConnection.getService("helloExamples", this);
CFMService.getData();
```

After the initial delivery of records, the `RecordSet` ActionScript class becomes responsible for fetching records. You can configure the client-side `RecordSet` object to fetch records in various ways using the `setDeliveryMode` ActionScript function.

Using Flash with ColdFusion components

ColdFusion components require little modification to work with Flash. The `cffunction` tag names the function and contains the CFML logic, and the `cfreturn` tag returns the result to Flash. The name of the ColdFusion component file (*.cfc) translates to the service name in ActionScript.

Note: For ColdFusion component methods to communicate with Flash movies, 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 675.

To create a ColdFusion component that interacts with a Flash movie:

- 1 Create a ColdFusion component, 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 movie.

- 3 Save the file.

The `helloWorld` service function is now available through the `flashComponent` service to ActionScript. The following ActionScript example calls this function:

```
#include "NetServices.as"
NetServices.setDefaultGatewayUrl("http://localhost:8500/flashservices/gateway");
gatewayConnection = NetServices.createGatewayConnection();
CFService = gatewayConnection.getService("flashExamples.flashComponent", this);
CFService.helloWorld();
```

In this example, the `getService` references the `flashComponent` component in the `flashExamples` directory. You can now call the `CFService` object `sayHello` and `getTime` functions.

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 rather than backslashes.

Using component metadata with the Flash Remoting service

Flash MX designers can use the Service Browser in the Flash MX authoring environment to discover business logic functionality built in ColdFusion. You use the `description` attribute of the `cffunction` and `cfargument` tags to describe the ColdFusion functionality to the Service Browser.

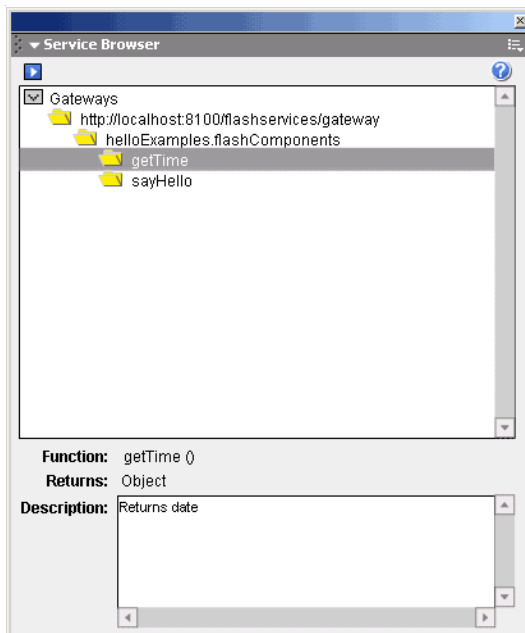
To create a ColdFusion component that describes itself to the Service Browser:

- 1 Open `flashComponents.cfc`, and modify the code so that it appears as follows:

```
<cfcomponent name="flashComponent">
    <cffunction name="helloWorld" access="remote" returnType="Struct"
        description="Returns hello message">
        <cfset tempStruct = StructNew()>
        <cfset tempStruct.timeVar = DateFormat(Now ())>
        <cfset tempStruct.helloMessage = "Hello World">
        <cfreturn tempStruct>
    </cffunction>
</cfcomponent>
```

In this example, the `description` attribute of the `cffunction` tag supplies a short text description of the component method.

- 2 Save the file.
- 3 Open the Flash MX authoring environment, and open the Service Browser.
- 4 If not already present, add your Flash Remoting service URL, such as `http://localhost:8500/flashservices/gateway`.
- 5 To add the `flashComponent` service, enter `helloExamples.flashComponent`.
- 6 When you click the `getTime` folder, the description appears in the Service Browser as shown in the following figure:



Using the Flash Remoting service with server-side ActionScript

The ability to create server-side ActionScript provides a familiar way for Flash developers to access ColdFusion query and HTTP features without learning CFML. You can place ActionScript files (*.asr) on the server that you want to call from the Flash application anywhere below the web server's root directory. To specify subdirectories of the webroot or a virtual directory, use package dot notation. For example, in the following assignment code, the stockquotes.asr file lives in the mydir/stock/ directory:

```
stockService = gatewayConnection.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 a subdirectory of that mapping. The CF.query and CF.http functions give you a well-defined interface for building SQL queries and HTTP operations of ColdFusion.

For example, the following server-side ActionScript function definition returns a RecordSet object:

```
function basicQuery()
{
    mydata = CF.query({datasource:"customers",
        sql:"SELECT * FROM myTable"});
    return mydata;
}
```

Note: For more information about server-side ActionScript, see *Using Server-Side ActionScript in ColdFusion*.

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 Administrator to add additional directories to the classpath.

To add a directory to ColdFusion classpath:

- 1 Open the ColdFusion 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 String sayHello()
    {
        return "Hello";
    }
}
```

Note: You cannot call constructors with Flash Remoting. You must use the default constructor.

In ActionScript, the following `getService` call invokes the `sayHello` public method of the `utils.UIComponents` class:

```
#include "NetServices.as"
NetServices.setDefaultGatewayUrl("http://localhost:8500/flashservices/gateway");
gatewayConnection = NetServices.createGatewayConnection();
javaService = gatewayConnection.getService("utils.UIComponents", this);
javaService.sayHello();
function sayHello_Result(result)
{
    trace(result);
}
```

Note: For more information about using Java objects with ColdFusion, see [Chapter 32, "Using Java objects" on page 769](#).

Handling errors with ColdFusion and Flash

To help with debugging, use the `cftry` and `cfcatch` tags to return error messages to the Flash Player, as in the following example:

```
<cftry>
  <cfset Flash.Result = undefinedVar>
  <cfcatch>
    <cfset Flash.Result="Failed">
  </cfcatch>
</cftry>
```

In this example, the first `cfset` tag fails to assign the value into `Flash.Result` because of an undefined variable.

Note: When you create a ColdFusion page that communicates with Flash, ensure that the ColdFusion page works before using it with Flash.

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:

Using XML and WDDX	687
Using Web Services	729
Integrating J2EE and Java Elements in CFML Applications	759
Integrating COM and CORBA Objects in CFML Applications	785

CHAPTER 30

Using XML and WDDX

This chapter describes how to use ColdFusion 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](#) 688
- [The XML document object](#) 689
- [ColdFusion XML tag and functions](#) 694
- [Using an XML object](#) 696
- [Creating and saving an XML document object](#) 698
- [Modifying a ColdFusion XML object](#) 700
- [Transforming documents with XSLT](#) 710
- [Extracting data with XPath](#) 711
- [Example: using XML in a ColdFusion application](#)..... 712
- [Moving complex data across the web with WDDX](#)..... 717
- [Using WDDX](#) 722

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, or 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 712](#) 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.
- Transform XML using XSLT.
- Extract data from XML documents using XPath expressions.
- Convert ColdFusion XML document objects to text and save them in files.

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 700](#).

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.

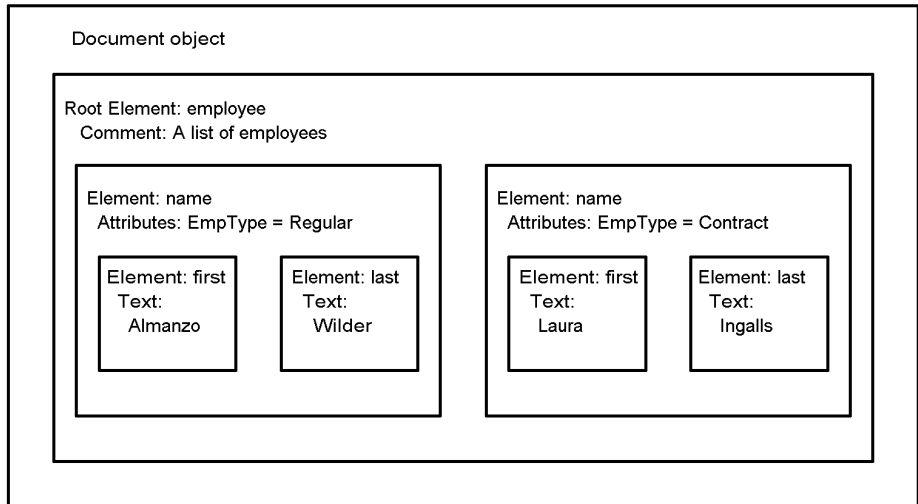
A simple XML document

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 <http://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 allows you to 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 693](#). 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 the output of a `cfdump` tag that displays the document object for the XML in “A simple XML document” on page 689. The following figure shows the output of the `cfdump` tag:

XmlComment																																																
XmlRoot	XmlName employee																																															
	XmlNsPrefix																																															
	XmlNsURI																																															
	XmlText																																															
	XmlComment A list of employees																																															
	XmlAttributes [empty struct]																																															
	XmlChildren																																															
	<table border="1"> <tr> <td>XmlName name</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes EmpType Regular</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName first</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Almanzo</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Wilder</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table> </td> </tr> </table> </td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName name</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes EmpType Contract</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName first</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Laura</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Ingalls</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table> </td> </tr> </table> </td> </tr> </table> </td></tr></table>	XmlName name	XmlNsPrefix	XmlNsURI	XmlText	XmlComment	XmlAttributes EmpType Regular	XmlChildren	<table border="1"> <tr> <td>XmlName first</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Almanzo</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Wilder</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table> </td> </tr> </table>	XmlName first	XmlNsPrefix	XmlNsURI	XmlText Almanzo	XmlComment	XmlAttributes [empty struct]	XmlChildren	<table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Wilder</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table>	XmlName last	XmlNsPrefix	XmlNsURI	XmlText Wilder	XmlComment	XmlAttributes [empty struct]	XmlChildren	<table border="1"> <tr> <td>XmlName name</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes EmpType Contract</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName first</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Laura</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Ingalls</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table> </td> </tr> </table> </td> </tr> </table>	XmlName name	XmlNsPrefix	XmlNsURI	XmlText	XmlComment	XmlAttributes EmpType Contract	XmlChildren	<table border="1"> <tr> <td>XmlName first</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Laura</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Ingalls</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table> </td> </tr> </table>	XmlName first	XmlNsPrefix	XmlNsURI	XmlText Laura	XmlComment	XmlAttributes [empty struct]	XmlChildren	<table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Ingalls</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table>	XmlName last	XmlNsPrefix	XmlNsURI	XmlText Ingalls	XmlComment	XmlAttributes [empty struct]	XmlChildren
	XmlName name																																															
	XmlNsPrefix																																															
	XmlNsURI																																															
	XmlText																																															
	XmlComment																																															
	XmlAttributes EmpType Regular																																															
	XmlChildren																																															
<table border="1"> <tr> <td>XmlName first</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Almanzo</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Wilder</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table> </td> </tr> </table>	XmlName first	XmlNsPrefix	XmlNsURI	XmlText Almanzo	XmlComment	XmlAttributes [empty struct]	XmlChildren	<table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Wilder</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table>	XmlName last	XmlNsPrefix	XmlNsURI	XmlText Wilder	XmlComment	XmlAttributes [empty struct]	XmlChildren																																	
XmlName first																																																
XmlNsPrefix																																																
XmlNsURI																																																
XmlText Almanzo																																																
XmlComment																																																
XmlAttributes [empty struct]																																																
XmlChildren																																																
<table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Wilder</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table>	XmlName last	XmlNsPrefix	XmlNsURI	XmlText Wilder	XmlComment	XmlAttributes [empty struct]	XmlChildren																																									
XmlName last																																																
XmlNsPrefix																																																
XmlNsURI																																																
XmlText Wilder																																																
XmlComment																																																
XmlAttributes [empty struct]																																																
XmlChildren																																																
<table border="1"> <tr> <td>XmlName name</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes EmpType Contract</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName first</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Laura</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Ingalls</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table> </td> </tr> </table> </td> </tr> </table>	XmlName name	XmlNsPrefix	XmlNsURI	XmlText	XmlComment	XmlAttributes EmpType Contract	XmlChildren	<table border="1"> <tr> <td>XmlName first</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Laura</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Ingalls</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table> </td> </tr> </table>	XmlName first	XmlNsPrefix	XmlNsURI	XmlText Laura	XmlComment	XmlAttributes [empty struct]	XmlChildren	<table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Ingalls</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table>	XmlName last	XmlNsPrefix	XmlNsURI	XmlText Ingalls	XmlComment	XmlAttributes [empty struct]	XmlChildren																									
XmlName name																																																
XmlNsPrefix																																																
XmlNsURI																																																
XmlText																																																
XmlComment																																																
XmlAttributes EmpType Contract																																																
XmlChildren																																																
<table border="1"> <tr> <td>XmlName first</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Laura</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> <tr> <td> <table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Ingalls</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table> </td> </tr> </table>	XmlName first	XmlNsPrefix	XmlNsURI	XmlText Laura	XmlComment	XmlAttributes [empty struct]	XmlChildren	<table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Ingalls</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table>	XmlName last	XmlNsPrefix	XmlNsURI	XmlText Ingalls	XmlComment	XmlAttributes [empty struct]	XmlChildren																																	
XmlName first																																																
XmlNsPrefix																																																
XmlNsURI																																																
XmlText Laura																																																
XmlComment																																																
XmlAttributes [empty struct]																																																
XmlChildren																																																
<table border="1"> <tr> <td>XmlName last</td> </tr> <tr> <td>XmlNsPrefix</td> </tr> <tr> <td>XmlNsURI</td> </tr> <tr> <td>XmlText Ingalls</td> </tr> <tr> <td>XmlComment</td> </tr> <tr> <td>XmlAttributes [empty struct]</td> </tr> <tr> <td>XmlChildren</td> </tr> </table>	XmlName last	XmlNsPrefix	XmlNsURI	XmlText Ingalls	XmlComment	XmlAttributes [empty struct]	XmlChildren																																									
XmlName last																																																
XmlNsPrefix																																																
XmlNsURI																																																
XmlText Ingalls																																																
XmlComment																																																
XmlAttributes [empty struct]																																																
XmlChildren																																																

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 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.
XmlNsPrefix	String	The prefix of the Namespace.
XmlNsURI	String	The URI of the Namespace.
XmlText	String	A string made of the concatenation of all text and CDATA text in the element, but not inside any child elements.
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	<code>#cdata-section</code>	Content of the CDATA section
COMMENT	<code>#comment</code>	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	<code>#text</code>	Content of the text node
ENTITY	Entity name	Empty string
NOTATION	Notation name	Empty string
DOCUMENT	<code>#document</code>	Empty string
FRAGMENT	<code>#document - fragment</code>	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 created from the XML example in [“A simple XML document” on page 689](#):

```
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
<code><cfxml variable="objectName" [caseSensitive="Boolean"]></code>	<p>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.</p> <p>If you specify the <code>CaseSensitive="True"</code> attribute, the case of names of elements and attributes in the document is meaningful. The default is <code>False</code>.</p> <p>For more information on using the <code>cfxml</code> tag, see “Creating a new XML document object using the cfxml tag” on page 698.</p>
<code>XmlParse("XMLStringVar" [, caseSensitive])</code>	<p>Converts an XML document that is represented as a string variable into an XML document object.</p> <p>If you specify the optional second argument as <code>True</code>, the case of names of elements and attributes in the document is meaningful. The default is <code>False</code>.</p> <p>For more information on using the <code>XmlParse</code> function, see “Creating an XML document object from existing XML” on page 699.</p>
<code>XmlNew([caseSensitive])</code>	<p>Returns a new, empty XML document object.</p> <p>If you specify the optional argument as <code>True</code>, the case of names of elements and attributes in the document is meaningful. The default is <code>False</code>.</p> <p>For more information on using the <code>XmlNew</code> function, see “Creating a new XML document object using the XmlNew function” on page 698.</p>
<code>XmlElemNew(objectName, "elementName")</code>	<p>Returns a new XML document object element with the specified name.</p> <p>For more information on using the <code>XmlElemNew</code> function, see “Adding an element” on page 704.</p>
<code>XmlChildPos(element, "elementName", position)</code>	<p>Returns the position (index) in an <code>XmlChildren</code> array of the <code>N</code>th child with the specified element name. For example, <code>XmlChildPos(mydoc.employee, "name", 2)</code> returns the position in <code>mydoc.employee.XmlChildren</code> of the <code>mydoc.employee.name[2]</code> element. This index can be used in the <code>ArrayInsertAt</code> and <code>ArrayDeleteAt</code> functions. For more information on using the <code>XmlChildPos</code> function, see “Determining the position of a child element with a common name” on page 704, “Adding an element” on page 704, and “Deleting elements” on page 706.</p>

Tag or function	Description
XMLTransform(<i>XMLVar</i> , <i>XSLTStringVar</i>)	Applies an Extensible Stylesheet Language Transformation (XSLT) to an XML document. The document can be represented <i>either</i> as a string variable <i>or</i> as an XML document object. The function returns the resulting XML document as a string. For more information on using the <code>XmlTransform</code> function, see “Transforming documents with XSLT” on page 710 .
XMLSearch(<i>objectName</i> , "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 <code>XmlSearch</code> function, see “Extracting data with XPath” on page 711
IsXmlDoc(<i>objectName</i>)	Returns True if the function argument is an XML document object.
IsXmlElement(<i>elementName</i>)	Returns True if the function argument is an XML document object element.
IsXMLRoot(<i>elementName</i>)	Returns True if the function argument is the root element of an XML document object.
ToString(<i>objectName</i>)	Converts an XML document object to a string representation.

Note: 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.

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.
- 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 689](#):

```
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”](#):

```
mydoc.employee.name[1].XmlAttributes.EmpType
mydoc.employee.name[1].XmlAttributes["EmpType"]
mydoc.employee.XmlChildren[1].EmpType
mydoc.XmlRoot.name[1].XmlAttributes["EmpType"]
mydoc.XmlRoot.XmlChildren[1].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, 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](#)” apply to `mydoc.employee.name[1].first` in the following expression:

```
mydoc.employee.name[1].first.MyNewElement = XmlElementNew(mydoc, NewElement);
```

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 = XmlElementNew(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 = XmlElementNew(mydoc, "phoneNumber");
```

The following expression causes an error:

```
mydoc.employee.name.phoneNumber = XmlElementNew(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 = XmlElementNew(mydoc, "AreaCode");
```

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 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.

Creating a new XML document object using the XmlNew function

The `XmlNew` function creates a new XML document object, which you must then populate. The following example creates and displays the same ColdFusion document object as in [“Creating a new XML document object using the cfxml tag”](#):

```
<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 a text string into a ColdFusion document object.

If the XML document is already represented by a string variable, use the `XmlParse` tag directly on the variable. 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)>
```

If the XML document is in a file, use `cffile` to convert the file to a CFML variable, then use the `XmlParse` tag on the resulting variable. For example, if the XML document is in the file `C:\temp\myxmldoc.xml`, use the following code to convert the file to an XML document object:

```
<cffile action="read" file="C:\temp\myxmldoc.xml" variable="XMLFileText">
<cfset myXMLDocument=XmlParse(XMLFileText)>
```

Note: If the file is not encoded with the ASCII or Latin-1 character set, use the `cffil` tag `charset` attribute to specify the file's character set. For example, if the file is encoded in UTF, specify `charset="UTF-8"`.

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 702](#) 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 <code>XmlChildren</code> array.
ArrayIsEmpty	Determines whether an element has any elements in its <code>XmlChildren</code> array.
StructCount	Determines the number of attributes in an element’s <code>XmlAttributes</code> structure.
StructIsEmpty	Determines whether an element has any attributes in its <code>XmlAttributes</code> 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 <code>XmlAttributes</code> structure. Returns the names of the children of an XML element.
ArrayInsertAt	Adds a new element at a specific location in an element’s <code>XmlChildren</code> array.
ArrayAppend ArrayPrepend	Adds a new element at the end or beginning of an element’s <code>XmlChildren</code> array.
ArraySwap	Swaps the children in the <code>XmlChildren</code> array at the specified position.
ArraySet	Sets a range of entries in an <code>XmlChildren</code> 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 <code>XmlChildren</code> array.
ArrayClear	Deletes all child elements from an element’s <code>XmlChildren</code> array.

Function	Use
StructDelete	<p>Deletes a selected attribute from an element's <code>XMLAttributes</code> structure.</p> <p>Deletes all children with a specific element name from an element's <code>XmlChildren</code> array.</p> <p>Deletes all attributes of an element.</p> <p>Deletes all children of an element.</p> <p>Deletes a selected property value.</p>
StructClear	Deletes all attributes from an element's <code>XMLAttributes</code> structure.
Duplicate	Copies an XML document object, element, or node structure.
IsArray	Returns True for the <code>XmlChildren</code> array. Returns false if you specify an element name, such as <code>mydoc.XmlRoot.name</code> , even if there are multiple name elements in <code>XmlRoot</code> .
IsStruct	Returns False for XML document objects, elements, and nodes. Returns True for <code>XmlAttributes</code> structures.
StructGet	Returns the specified structure, including XML document objects, elements, nodes, and <code>XmlAttributes</code> structures.
StructAppend	Appends a document fragment XML document object to another XML document object.
StructInsert	Adds a new entry to an <code>XmlAttributes</code> structure.
StructUpdate	Sets or replaces the value of a document object property such as <code>XmlName</code> , or of a specified attribute in an <code>XmlAttributes</code> 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
<code>IsArray(elemPath.elemName)</code>	Always returns False.
<code>ArrayClear(elemPath.elemName)</code>	Removes all the elements with name <code>elemName</code> from the <code>elemPath</code> element.
<code>ArrayLen(elemPath.elemName)</code>	Returns the number of elements named <code>elemName</code> in the <code>elemPath</code> element.
<code>ArrayDeleteAt(elemPath.elemName, n)</code>	Deletes the <code>n</code> th child named <code>elemName</code> from the <code>elemPath</code> element.

Array function	Result
<code>IsEmpty(<i>elemPath.elemName</i>)</code>	Always Returns False.
<code>ArrayToList(<i>elemPath.elemName</i>, <i>n</i>)</code>	Returns a comma separated list of all the <code>XmlText</code> properties of all the children of <i>elemPath</i> named <i>elemName</i> .

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.

Adding

Use the following techniques to add new information to an element:

Type	Using a function	Using an assignment statement
Attribute	<code>StructInsert(<i>xmlElemPath.XmlAttributes</i>, "key", "value")</code>	<code><i>xmlElemPath.XmlAttributes</i>.key = "value"</code> <code><i>xmlElemPath.XmlAttributes</i>["key"] = "value"</code>
Child element	To append: <code><i>ArrayAppend</i>(<i>xmlElemPath.XmlChildren</i>, <i>newElem</i>)</code> To insert: <code><i>StructInsertAt</i>(<i>xmlElemPath.XmlChildren</i>, <i>position</i>, <i>newElem</i>)</code>	To append: <code><i>xmlElemPath.XmlChildren</i>[<i>i</i>] = <i>newElem</i></code> <code><i>xmlElemPath.newChildName</i> = <i>newElem</i></code> (where <i>newChildName</i> must be the same as <i>newElem.XmlName</i> and cannot be an indexed name such as <code>name[3]</code>)

Deleting

Use the following techniques to delete information from an element:

Type	Using a function	Using an assignment statement
Property	<code><i>StructDelete</i>(<i>xmlElemPath</i>, <i>propertyName</i>)</code>	<code><i>xmlElemPath.propertyName</i>=""</code>
Attribute	All attributes: <code><i>StructDelete</i>(<i>xmlElemPath</i>, <i>XmlAttributes</i>)</code> A specific attribute: <code><i>StructDelete</i>(<i>xmlElemPath.XmlAttributes</i>, "attributeName")</code>	Not available

Type	Using a function	Using an assignment statement
Child element	<p>All children of an element:</p> <pre>StructDelete(xmlElemPath, "XmlChildren") or ArrayClear(xmlElemPath.XmlChildren)</pre> <p>All children with a specific name:</p> <pre>StructDelete(xmlElementpath, "elemName") ArrayClear(xmlElemPath.elemName)</pre> <p>A specific child:</p> <pre>ArrayDeleteAt(xmlElemPath.XmlChildren, position) ArrayDeleteAt(xmlElemPath.elemName, position)</pre>	Not available

Changing

Use the following techniques to change the contents of an element:

Type	Using a function	Using an assignment statement
Property	<pre>StructUpdate(xmlElemPath, "propertyName", "value")</pre>	<pre>xmlElemPath.propertyName = "value" xmlElemPath["propertyName"] = "value"</pre>
Attribute	<pre>StructUpdate(xmlElemPath.XmlAttributes, "attributeName", "value")</pre>	<pre>xmlElemPath.XmlAttributes. attributeName="value" xmlElemPath.XmlAttributes ["attributeName"] = "value"</pre>
Child element (replace)	<pre>ArraySet(xmlElemPath.XmlChildren, index, index, newElement)</pre> <p>(use the same value for both index entries to change one element)</p>	<p>Replace first or only child named <i>elementName</i>:</p> <pre>parentElemPath.elementName = newElement parentElemPath["elementName"] = newElement</pre> <p>Replace a specific child named <i>elementName</i>:</p> <pre>parentElemPath.elementName [index] = newElement or parentElemPath["elementName"] [index] = newElement</pre>

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 689](#).

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 `XmlElementNew` function to create a new, empty element. This function has the following form:

```
XmlElementNew(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 706](#) 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>
```

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.XmlChildren[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. For 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].XmlAttributes.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")>
```

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:\Neo\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 `CompanyInfo` database and saves the information as an XML document.

```

<!-- Query the database and get the names in the employee table --->
<cfquery name="myQuery" datasource="CompanyInfo">
    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)#>

```

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:\Neo\wwwroot\testdocs\simpletransform.xsl"
        variable="xslDoc">
<cfset transformedXML = XmlTransform(mydoc, xslDoc)>
<cffile action="write" file="C:\Neo\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 <http://www.w3.org/Style/XSL/>. There are also several books 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 `XmlPath` 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 <http://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 `CompanyInfo` 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>
    </item>
  </items>
</order>
```

```

        </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:\Neo\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="CompanyInfo">
    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
<pre><cffile action="read" file="C:\Neo\wwwroot\examples\order.xml" variable="myxml"> <cfset mydoc = XmlParse(myxml)> <cfset accountNum=#mydoc.order. customer.XmlAttributes.accountNum#></pre>	<p>Reads the XML from a file and convert it to an XML document object.</p> <p>Sets the accountNum variable from the customer entry's accountnum attribute.</p>
<pre><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></pre>	<p>Converts the XML document object into a query object.</p> <p>Creates a query with columns for the item_id, name, qty, and unitPrice values for each item.</p> <p>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.</p>
<pre><cfquery name="discountQuery" datasource="CompanyInfo"> SELECT * FROM employee WHERE Emp_Id = #accountNum# </cfquery> <cfset drate = 0> <cfif #discountQuery.RecordCount# is 1> <cfset drate = 10> </cfif></pre>	<p>If the account number is the same as an employee ID in the CompanyInfo 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%.</p>
<pre><cfquery name="priceQuery" dbType="query"> SELECT SUM(qty*unitPrice) AS totalPrice FROM orderquery </cfquery> <cfset discountPrice = priceQuery.totalPrice * (1 - drate/100)></pre>	<p>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.</p>

Code	Description
<pre><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></pre>	<p>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.</p>

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 http://www.openwddx.org/downloads/dtd/wddx_dtd_10.txt.

While WDDX is a valuable tool for ColdFusion developers, 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 22KB.
- 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 <http://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
<pre>x = new Object(); x.a = "Property a"; x.b = "Property b";</pre>	<pre>x = structNew(); x.a = "Property a"; x.b = "Property b";</pre>

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 <http://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 <http://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 <code>cfwddx</code> 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 $\pm 1.7E\pm 308$. 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 <code>[_A-Za-z][_0-9A-Za-z]*</code> 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` and `WddxRecordset`, 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 723 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 `cfsnippets` data source that is installed with ColdFusion:

```
<!-- Create a simple query -->
<cfquery name = "q" datasource = "cfsnippets">
    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/wddxserializeddeserialize.cfm.
- 2 Display `http://localhost/myapps/wddxserializeddeserialize.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>
<br>
```

```
<!-- Now read the data from client scope into a new structure -->
<cfwddx action="wddx2cfml" input="#Client.wddxRelatives#" output="sameRelatives">
```

```
A dump of the sameRelatives structure <br>
generated from client.wddxRelatives<br>
<br>
<cfdump var="#sameRelatives#">
```


CHAPTER 31

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 730](#)
- [Working with WSDL files 733](#)
- [Consuming web services 736](#)
- [Publishing web services 744](#)
- [Handling complex data types 753](#)

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 ColdFusion page or CGI script.

Web services are a new technology that lets you access application functionality, which resides on remote computers, that someone created and made available. 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 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 need to know the URL of the WSDL file defining the service. For example, you can access the WSDL file for the BabelFish web service at the following URL:

`http://www.xmethods.net/sd/2001/BabelFishService.wsdl`

For an overview of WSDL syntax, see [“Working with WSDL files” on page 733](#)

2 Make a request to the web service.

The following example invokes an operation on the BabelFish web service to translate the string “Hello World” from English into Spanish:

```
<cfinvoke
  webservice='http://www.xmethods.net/sd/2001/BabelFishService.wsdl'
  method='BabelFish'
  translationmode="en_es"
  sourcedata="Hello World"
  returnVariable='foo'>
</cfoutput>#foo#</cfoutput>
```

For more information on consuming web services, see [“Consuming web services” on page 736](#).

Basic web service concepts

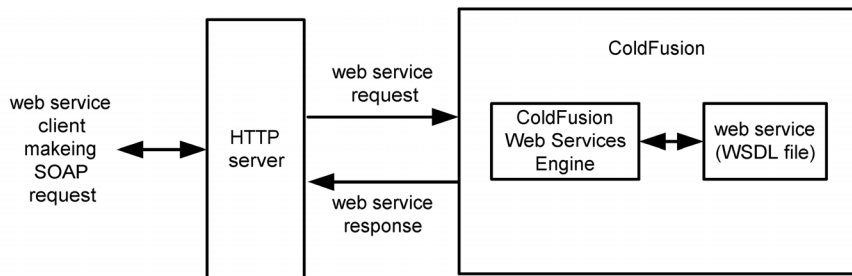
You must be familiar with the underlying architecture of a web service provider in order to fully understand how web services work.

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 Access Open Protocol)
- WSDL (Web Services Description Language)
- UDDI (Universal Description, Discovery, and Integration)

The following simple figure shows how the ColdFusion 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 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, 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 the following URL:

<http://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 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 733](#).

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 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 the following URL:

<https://www-3.ibm.com/services/uddi/protect/registry.html>

You can find additional information about UDDI at the following URL:

<http://www.uddi.org/about.html>

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 the following URL:

<http://www.w3.org/TR/wsdl>

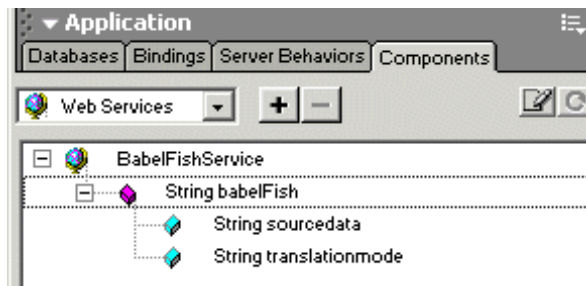
Creating a WSDL file

To publish a web service, you construct the service's functionality and then create the WSDL file defining the service. In ColdFusion, 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 744](#).

For more information on components, see [Chapter 11, “Building and Using ColdFusion Components” on page 217](#).

Viewing a WSDL file using Dreamweaver MX

Dreamweaver MX contains a utility to view web services, including operation names, parameter names, and parameter data types. The following figure shows a WSDL file for the BabelFish web service:



This figure shows that the web service method babelFish returns a string, and that it takes string parameters named sourcedata and translationmode as input.

To open the Components tab in the Dreamweaver MX and add a web service:

- 1 Choose Window > Components, or use Ctrl-F7, to open the Components panel.
- 2 In the Components panel, choose Web Services from the dropdown 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.

For more information on using Dreamweaver MX, see its online Help system.

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, 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 BabelFish web service:

```
<?xml version="1.0" ?>
  <definitions name="BabelFishService"
    xmlns:tns="http://www.xmethods.net/sd/BabelFishService.wsdl"
    targetNamespace="http://www.xmethods.net/sd/BabelFishService.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="BabelFishRequest">
      <part name="translationmode" type="xsd:string" />
      <part name="sourcedata" type="xsd:string" />
    </message>
    <message name="BabelFishResponse">
      <part name="return" type="xsd:string" />
    </message>
    <portType name="BabelFishPortType">
      <operation name="BabelFish">
        <input message="tns:BabelFishRequest" />
        <output message="tns:BabelFishResponse" />
      </operation>
    </portType>
    <binding name="BabelFishBinding" type="tns:BabelFishPortType">
      <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
      <operation name="BabelFish">
        <soap:operation soapAction="urn:xmethodsBabelFish#BabelFish" />
        <input>
          <soap:body use="encoded" namespace="urn:xmethodsBabelFish"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </input>
        <output>
          <soap:body use="encoded" namespace="urn:xmethodsBabelFish"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </output>
      </operation>
    </binding>
    <service name="BabelFishService">
      <documentation>Translates text of up to 5k in length, between a variety of
        languages.</documentation>
      <port name="BabelFishPort" binding="tns:BabelFishBinding">
        <soap:address location="http://services.xmethods.net:80/perl/soaplite.cgi" />
      </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 736](#).

Consuming web services

ColdFusion provides two methods for consuming web services. The method that you choose depends on your ColdFusion programming style and application.

The following table describes these methods:

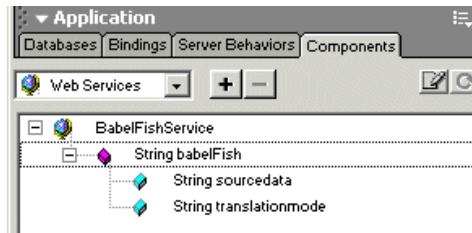
Method	CFML operator	Description
CFScript	CreateObject()	Consumes a web service from within a CFScript block
CFML tag	cfinvoke	Consumes a web service from within a block of CFML code

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 BabelFish web service from AltaVista. BabelFish can translate string up to 5 KB in length from one language to another. You can read the WSDL file for this web service in [“Reading a WSDL file” on page 734](#).

If you add the BabelFish web service in Dreamweaver MX, you see the following description of it in the Application panel.



For information on adding a web service in Dreamweaver, see [“Viewing a WSDL file using Dreamweaver MX” on page 733](#). For more information on BabelFish, see <http://babelfish.altavista.com/>.

Passing parameters to a web service

One type of information in the WSDL file defines the web service operations and the input and output parameters of each operation, including the data type of each parameter. If you register the web service in Dreamweaver MX, as shown in the previous section, you see that the data type of both input parameters is string.

The following example shows a portion of the WSDL file for the BabelFish web service:

```
<message name="BabelFishRequest">
  <part name="translationmode" type="xsd:string" />
  <part name="sourcedata" type="xsd:string" />
</message>
<message name="BabelFishResponse">
  <part name="return" type="xsd:string" />
</message>
```

```

</message>
<portType name="BabelFishPortType">
  <operation name="BabelFish">
    <input message="tns:BabelFishRequest" />
    <output message="tns:BabelFishResponse" />
  </operation>
</portType>

```

The operation name used in the examples in this section is BabelFish. This operation takes a single input parameter defined as a message of type BabelFishRequest.

You can see that the message BabelFishRequest contains two string parameters: `translationmode` and `sourcedata`. When you call the BabelFish operation, you pass both parameters 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 the WSDL file for the web service.

If you register the web service in Dreamweaver MX, you see that the data type of the return value is string.

The following example shows a portion of the WSDL file for the BabelFish web service:

```

<message name="BabelFishRequest">
  <part name="translationmode" type="xsd:string" />
  <part name="sourcedata" type="xsd:string" />
</message>
<message name="BabelFishResponse">
  <part name="return" type="xsd:string" />
</message>
<portType name="BabelFishPortType">
  <operation name="BabelFish">
    <input message="tns:BabelFishRequest" />
    <output message="tns:BabelFishResponse" />
  </operation>
</portType>

```

The operation BabelFish returns a message of type BabelFishResponse. The message statement in the WSDL file defines the BabelFishResponse 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 has the following syntax:

```

<cfinvoke
  webservice = "URLtoWSDL"
  method = "operationName"
  inputParam1 = "val1"
  inputParam2 = "val2"

```

```
...
returnVariable = "varName"
>
```

where:

- `webservice` specifies the URL to the WSDL file for the web service.
- `method` specifies the operation of the web service to invoke.
- `inputParamN` specifies an input parameter passed to the operation.
- `returnVariable` specifies the name of the variable containing any results returned from the web service.

To access a web service using `cfinvoke`:

- 1 Create a ColdFusion page with the following content:

```
<cfinvoke
  webservice = "http://www.xmethods.net/sd/2001/BabelFishService.wsdl"
  method = "BabelFish"
  translationmode = "en_es"
  sourcedata = "Hello world, friend"
  returnVariable = "foo">
</cfoutput>#foo#</cfoutput>
```

- 2 Save the page as `wscfc.cfm` in your web root directory.
- 3 View the page in your browser.

The following string appears in your browser:

```
Hola mundo, amigo
```

You can pass parameters to web services using two other mechanisms: the `cfinvokeargument` tag and the `argumentCollection` attribute of the `cfinvoke` tag.

To pass parameters using the `cfinvokeargument` tag, you write your call to the web service, as the following code shows:

```
<cfinvoke
  webservice ="http://www.xmethods.net/sd/2001/BabelFishService.wsdl"
  method ="BabelFish"
  returnVariable = "varName" >
  <cfinvokeargument name="translationmode" value="en_es">
  <cfinvokeargument name="sourcedata" value="Hello world, friend">
</cfinvoke>
</cfoutput>#varName#</cfoutput>
```

The `cfinvokeargument` tag is a nested tag of the `cfinvoke` tag that lets you specify the name and value of a parameter passed to the web service.

You can also use an attribute collection to pass parameters. An attribute collection 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.translationmode = "en_es";
  stArgs.sourceData= "Hello world, friend";
</cfscript>
```

```

<cfinvoke
  webservice = "http://www.xmethods.net/sd/2001/BabelFishService.wsdl"
  method     = "BabelFish"
  argumentCollection = "#stArgs#"
  returnVariable = "varName" >
</cfoutput>#varName#</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. The `CreateObject` function has the following syntax:

```
webServiceName = CreateObject("webservice", "URLtoWSDL")
```

where `URLtoWSDL` specifies the URL to the WSDL file for the web service.

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 follows:

```
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/BabelFishService.wsdl");
  xlatstring = ws.BabelFish("en_es", "Hello world, friend");
  writeOutput(xlatstring);
</cfscript>

```

- 2 Save the page as `wscfscript.cfm` in your web root directory.

- 3 View the page in your browser.

The following string appears in your browser:

```
Hola mundo, amigo
```

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/BabelFishService.wsdl");
  xlatstring = ws.BabelFish(translationmode = "en_es",
    sourcedata = "Hello world, friend");

```

```
</cfscript>
<cfoutput>#xlatstring#</cfoutput>
```

Calling web services from a 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 on Flash Remoting, see [Chapter 29, “Using the Flash Remoting Service”](#) on page 673.

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 281.

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 pound signs.

Note: ColdFusion supports the use of inout and out parameters to consume web services. However, ColdFusion does not support inout and out parameters when creating web services for publication.

Configuring web services in the ColdFusion Administrator

The ColdFusion 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 automatically registers it in the Administrator.

For example, the following code references the URL to the BabelFish WSDL file:

```
<cfscript>
    ws = CreateObject("webservice",
        "http://www.xmethods.net/sd/2001/BabelFishService.wsdl");
    xlatstring = ws.BabelFish("en_es", "Hello world, friend");
    writeoutput(xlatstring);
</cfscript>
```

If you register the BabelFish web service in the ColdFusion Administrator using, for example, the name wsBabel, you could then reference the web service as follows:

```
<cfscript>
    ws = CreateObject("webservice", "wsBabel");
    xlatstring = ws.BabelFish("en_es", "Hello world, friend");
    writeoutput(xlatstring);
</cfscript>
```

Not only does this enable you to shorten your code, registering a web service in the ColdFusion Administrator lets you change a web service's URL without modifying your code. So, if the BabelFish web service moves to a new location, you only update the administrator setting; not your application code.

For more information, see the online help in the ColdFusion Administrator.

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 BabelFish web service contains the following definition of input and return parameters:

```
<message name="BabelFishRequest">
    <part name="translationmode" type="xsd:string" />
    <part name="sourcedata" type="xsd:string" />
</message>
<message name="BabelFishResponse">
    <part name="return" type="xsd:string" />
</message>
```

As part of consuming web services, you must understand how ColdFusion 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
array	SOAP-ENC:Array
binary	xsd:base64Binary
date	xsd:dateTime
void (operation returns nothing)	
struct	complex type

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 753](#).

Consuming ColdFusion web services

Your application might 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.

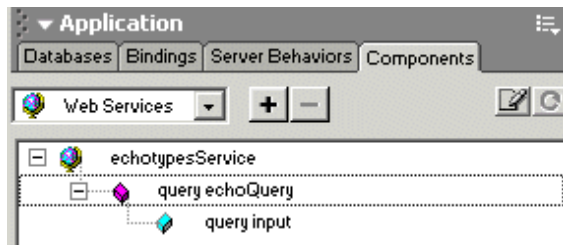
For example, when ColdFusion publishes a web service that returns a query, or takes a query as an input, the WSDL file for that service lists its data type as QueryBean. However, a ColdFusion application consuming this web service can pass a ColdFusion query object to the function as an input, or write 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 741](#).

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>
```


If you add this web service in Dreamweaver MX, you see the following description of it in the Application panel:



Note: This figure assumes that you create a web component named `echotypes.cfc` that contains the `echoQuery` function definition shown above, and write `echotypes.cfc` to your web root directory.

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="echoQueryRequest">
  <wsdl:part name="input" type="tnsl:QueryBean"/>
</wsdl:message>
<wsdl:message name="echoQueryResponse">
  <wsdl:part name="return" type="tnsl:QueryBean"/>
</wsdl:message>
```

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="CompanyInfo">
  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 11, “Building and Using ColdFusion Components” on page 217](#).

Creating components for web services

ColdFusion components 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, 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. If the function does not return anything, set its `returnType` attribute to `void`.
- 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
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
any	complex type
component definition	complex type

In most cases, consumers of ColdFusion web services will be able to easily pass data to and return results from component functions by mapping their data types to the WSDL data types shown above.

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 756.

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 748.

Producing WSDL files

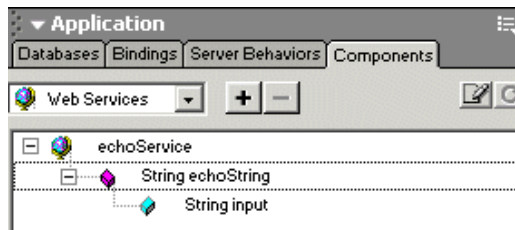
ColdFusion automatically creates a WSDL file for any component referenced as a web service. For example, if you have a component named `echo.cfc` in your web root directory, you can view its corresponding WSDL file by requesting the component as follows:

```
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>
```

If you register the component in Dreamweaver MX, it appears in the Application panel as the following figure shows:



Requesting the WSDL file returns the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://webservices"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:intf="http://webservices"
  xmlns:impl="http://webservices-impl"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:message name="echoStringResponse">
    <wsdl:part name="return" type="SOAP-ENC:string" />
  </wsdl:message>
  <wsdl:message name="echoStringRequest">
    <wsdl:part name="input" type="SOAP-ENC:string" />
  </wsdl:message>
  <wsdl:portType name="echo">
    <wsdl:operation name="echoString" parameterOrder="in0">
      <wsdl:input message="intf:echoStringRequest" />
      <wsdl:output message="intf:echoStringResponse" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="echo.cfcSoapBinding" type="intf:echo">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
  </wsdl:binding>
  <wsdl:operation name="echoString">
    <wsdlsoap:operation soapAction="" style="rpc" />
    <wsdl:input>
```

```

        <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/
        soap/encoding/" namespace="http://webservices" />
    </wsdl:input>
    <wsdl:output>
        <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/
        soap/encoding/" namespace="http://webservices" />
    </wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="echo.cfcService">
    <wsdl:port name="echo.cfc" binding="intf:echo.cfcSoapBinding">
        <wsdlsoap:address location="http://SMGILSON02/webservices/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 components let you define both methods and properties of the component. Once defined, you can use components to define data types for web services. The following code defines a component in the file `address.cfc`:

```
<cfcomponent>
  <cfproperty name="Number" type="numeric">
  <cfproperty name="Street" type="string">
  <cfproperty name="City" type="string">
  <cfproperty name="State" type="string">
  <cfproperty name="Country" type="string">
</cfcomponent>
```

This component contains properties that represent a street address. 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>
```

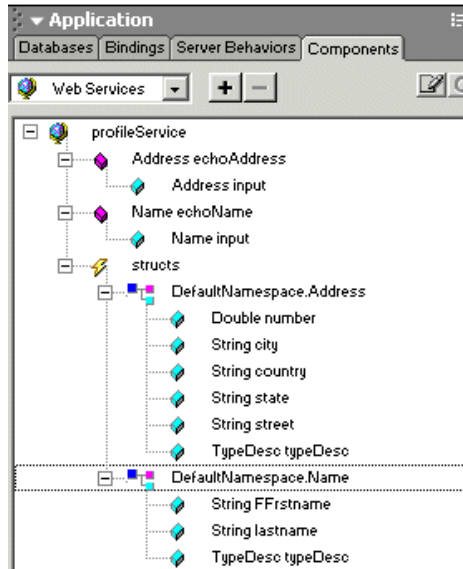
You can then use `address` and `name` to define data types in a ColdFusion component created to publish a web service, as the following example shows:

```
<cfcomponent>
  <cffunction name="echoName" returnType="name" access="remote">
    <cfargument name="input" type="name">
    <cfreturn #arguments.input#>
  </cffunction>

  <cffunction name="echoAddress" returnType="address" access="remote">
    <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 ColdFusion mapping to the directory containing them.

If you register the component in Dreamweaver MX, it appears in the Application panel as the following figure shows:



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, shown below is the definition for name:

```
<complexType name="name">
  <all>
    <element name="Firstname" nillable="true" type="xsd:string" />
    <element name="Lastname" nillable="true" type="xsd:string" />
  </all>
</complexType>
```

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 Administrator to disable the RDS password.

For more information, see [Chapter 11, “Building and Using ColdFusion Components”](#) on page 217.

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.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 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 Administrator, see [“Configuring web services in the ColdFusion Administrator”](#) on page 741.

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.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 example `Application.cfm` file shows:

```
<cfsilent>
<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>
</cfsilent>

```

This example does not show how to perform user verification. For more information on verification, see [Chapter 16, “Securing Applications” on page 347](#).

Assigning security roles to functions

ColdFusion components offer role-based security. The following example creates a component method that deletes files:

```

<cfcomponent>
  <cffunction name="deleteFile" access="remote" roles="admin,manager">
    <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.

Role based security can be used with any ColdFusion component, not just for web services. For more information on roles, see [Chapter 16, “Securing Applications” on page 347](#).

Using programmatic security

You can implement your own security within the a function 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>
```

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.

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
binary	xsd:base64Binary
date	xsd:dateTime
void (operation returns nothing)	
struct*	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:sequence>
</s:complexType>
```

```

        <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, you create a `ColdFusion` structure, initialize six fields of the structure that correspond to the six elements of `Employee`, 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/echosimple.asmx?wsdl");
    ws.echoStruct(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 the dot notation in the same way 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 742](#).

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="Map">
  <sequence>
    <element name="item" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <all>
          <element name="key" type="xsd:anyType" />
          <element name="value" type="xsd:anyType" />
        </all>
      </complexType>
    </element>
  </sequence>
</complexType>
```

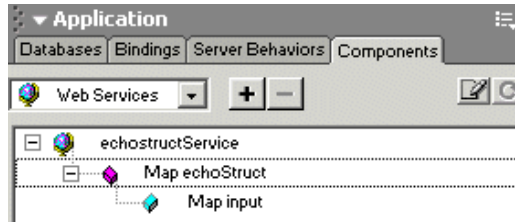
```

        </complexType>
    </element>
</sequence>
</complexType>

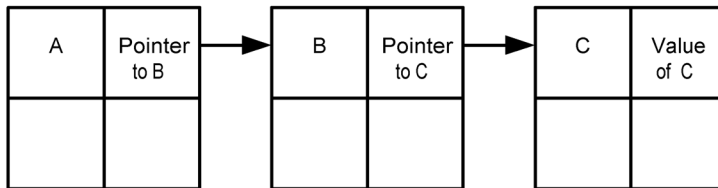
```

This complex type defines a representation of a structure, where the structure keys and values can be any type.

If you register the component in Dreamweaver MX, it appears in the Application panel as the following figure shows:



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. For example, if you have a structure containing the field A.B.C, that field is represented as the following figure shows:



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	2-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>
```


CHAPTER 32

Integrating J2EE and Java Elements in CFML Applications

This chapter describes how to integrate J2EE elements, including the following, into your ColdFusion application:

- JSP pages and servlets
- JSP tags
- Java objects, including Enterprise JavaBeans (EJBs)

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](#) 760
- [Using JSP tags and tag libraries](#) 762
- [Interoperating with JSP pages and servlets](#) 764
- [Using Java objects](#) 769

About ColdFusion, Java, and J2EE

ColdFusion 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 Administrator Java Applets Extensions page. (For information on registering applets, see the ColdFusion Administrator online Help.)
- 2 Use the `cfapplet` tag to call the applet. The `appletSource` attribute must be the Applet name assigned in ColdFusion 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 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 `include` and `import` tags to include pages and import tag libraries.
- **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 `Application`, `Session`, and `Request` 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 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 ColdFusion Administrator JVM and Java Settings page.
- 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 Server.

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/sdk_1.3/techdocs/api/javax/servlet/jsp/PageContext.html.

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 762 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 764.

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.
- 2 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`.

- 3 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/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 766](#).

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
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 774](#).

To share session variables, you must specify J2EE session management in the ColdFusion Administrator. For more information on configuring and using J2EE Session scope management, see [“ColdFusion and J2EE session management,” in Chapter 15](#).

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(URLEncoder.format("/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”](#).

Accessing ColdFusion application and session variables in JSP pages

ColdFusion 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 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 ColdFusion `cfapplication` tag, the application is unnamed. ColdFusion supports only a single unnamed application, so if multiple `cfapplication` tags do not specify an application name, all pages affected by the tags share the single unnamed application Scope. This scope maps directly to the J2EE

application scope. Similarly, all sessions of unnamed applications correspond directly to the J2EE application server's session scope.

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
<pre><cfapplication name="myApp" sessionmanagement="yes"></pre>	Specifies the application name as myApp and enables session management. In most applications, this tag is in the Application.cfm page.
<pre><cfscript> Request.myVariable = "This"; Session.myVariable = "is a"; Application.myVariable = "test."; GetPageContext().include ("hello.jsp?name=Bobby"); </cfscript></pre>	Sets ColdFusion Request, Session, and Application, scope variables. Uses the same name, myVariable, for each variable.
	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 CFML. 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:

Code	Description
<pre><%@page import="java.util.*" %></pre>	Imports the <code>java.util</code> package. This contains methods required in the JSP page.
<pre><h2>Hello <%= request.getParameter("name")%>!</h2></pre>	Displays the name passed as a URL parameter from the ColdFusion page. The parameter name is case-sensitive,
<pre>
request.myVariable: <%= request. getAttribute("myvariable")%></pre>	Uses the <code>getAttribute</code> method of the JSP request object to displays the value of the Request scope variable <code>myVariable</code> .
<pre>
session.myVariable: <%= ((Map)(session.getAttribute("myApp"))) .get("myVariable")%></pre>	Uses the <code>getAttribute</code> method of the JSP session object to get the <code>myApp</code> object (the Application scope). Casts this to a Java <code>Map</code> object and uses the <code>get</code> method to obtain the <code>myVariable</code> value for display.
<pre>
Application.myVariable: <%= ((Map)(application.getAttribute("myApp"))) .get("myVariable")%></pre>	Uses the <code>getAttribute</code> method of the JSP <code>myApp</code> application object to obtain the value of <code>myVariable</code> in the Application scope.

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");%>
<% application.setAttribute("myApp.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
<pre><%@page import="java.util.*" %></pre>	Imports the java.util package. This contains methods required in the JSP page.
<pre><% request.setAttribute("myvariable", "This");%></pre>	Uses the setAttribute method of the JSP request object to set the value of the Request scope variable myVariable.
<pre><% ((Map)session.getAttribute("myApp")) .put("myVariable", "is a");%></pre>	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.
<pre><% application.setAttribute ("myApp.myvariable", "test.");%></pre>	Uses the setAttribute method of the JSP application object to set the value of myVariable in the myApp application scope.
<pre><jsp:include page="hello.cfm"> <jsp:param name="name" value="Robert" /> </jsp:include></pre>	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
<pre><cfapplication name="myApp" sessionmanagement="yes"></pre>	Specifies the application name as myApp and enables session management. In most applications, this tag is in the Application.cfm page.
<pre><h2>Hello #URL.name#!</h2></pre>	Displays the name passed using the jsp:param tag on the JSP page. The parameter name is <i>not</i> case-sensitive.
<pre>Request.myVariable: #Request.myVariable#
 Session.myVariable: #Session.myVariable#
 Application.myVariable: #Application.myVariable#
</pre>	Displays the Request.myVariable, Session.myVariable, and Application.myVariable values.

Using Java objects

You use the `cfoobject` 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 774](#).

The examples in the following sections assume that the `name` attribute in the `cfoobject` 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 `cfoobject` tag makes Java objects available in ColdFusion. 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:

```
<cfoobject type="Java" class="MyClass" name="myObj">
```

Although the `cfoobject` tag loads the class, it does **not** create an instance object. Only static methods and fields are accessible immediately after the call to `cfoobject`.

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 `cfoobject` tag; for example:

```
<cfoobject 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 `cfoobject` does not.

An object created using `cfoobject` 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 `getPropertyName()` and `setPropertyName(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 774.

Calling JavaBean get and set methods

ColdFusion can automatically invoke `getPropertyName()` and `setPropertyName(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 void Employee() {
        FirstName = "";
        LastName = "";
        Salary = 0.0f;
        JobGrade = 0;
    }

    public void Employee(String First, String Last) {
        FirstName = First;
        LastName = Last;
        Salary = 0.0f;
        JobGrade = 0;
    }

    public void 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
<pre><cfobject action=create type=java class=Employee name=emp></pre>	Loads the Employee Java class and gives it an object name of emp.
<pre><!-- <cfset emp.init() --></pre>	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.
<pre><cfset emp.firstname="john"> <cfset emp.lastname="doe"></pre>	Sets the public fields in the emp object to your values.
<pre><cfset firstname=emp.firstname> <cfset lastname=emp.lastname></pre>	Gets the field values back from emp object.
<pre><cfoutput> Employee name is #firstname# #lastname# </cfoutput></pre>	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).

CFML	Java
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 771](#), 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, 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 on 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, see [Chapter 14, “Handling Errors” on page 281](#).

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, <http://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:

```

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();
}
```

Using an EJB

ColdFusion 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 `cfoject 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).

While 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 `cfoject` 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 `cfoject` 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 `cfoject` 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 `InitialContext` 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 771](#). 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
<code>ReverseString</code>	Reverses the order of a string.
<code>ReverseStringArray</code>	Reverses the order of elements in an array of strings.
<code>Add</code>	Overloaded: Adds and returns two integers or floats or adds the <code>mPublicInt</code> members of two Example class objects and returns an Example class object.
<code>SumArray</code>	Returns the sum of the elements in an integer array.
<code>SumObjArray</code>	Adds the values of the <code>mPublicInt</code> members of an array of Example class objects and returns an Example class object.
<code>ReverseArray</code>	Reverses the order of an array of integers.
<code>Flip</code>	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 33

Integrating COM and CORBA Objects in CFML Applications

This chapter describes how to use the `cfoobject` 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](#) 786
- [Creating and using objects](#) 788
- [Getting started with COM and DCOM](#)..... 790
- [Creating and using COM objects](#) 793
- [Getting started with CORBA](#) 797
- [Creating and using CORBA objects](#) 797
- [CORBA example](#)..... 805

About COM and CORBA

This section provides some basic information on COM and CORBA objects supported in ColdFusion and provides resources for further inquiry.

About objects

COM and CORBA are two of the **object** technologies supported by ColdFusion. Other object technologies include Java and ColdFusion components. For more information on ColdFusion components see [Chapter 11, “Building and Using ColdFusion Components” on page 217](#).

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 `cfoobject` 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 788](#).

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, <http://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: <http://www.omg.com>.

Creating and using objects

You use the `cfoject` 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 `cfoject` 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 793](#) and ["Creating CORBA objects" on page 797](#). The following examples use a `cfoject` tag to create a COM object and a `CreateObject` function to create a CORBA object:

```
<cfoject type="COM" action="Create" name="obj" class="sample.MyObject">  
obj = CreateObject("CORBA", "d:\temp\tester.ior", "IOR", "Visibroker")
```

ColdFusion releases any object created by `cfoject` 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 `cfoject` 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 <http://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 `cfoject` 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 the OLE/COM Object Viewer to view the component's interface.

Using the OLE/COM Object Viewer

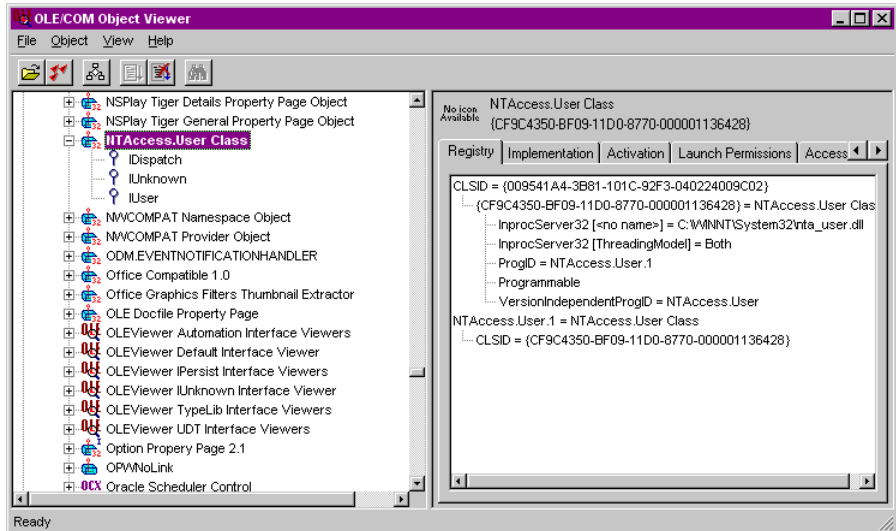
The OLE/COM Object Viewer installation installs the executable, by default, as `\mstools\bin\oleview.exe`. You use the Object Viewer to retrieve a COM object's Program ID, as well as its methods and properties.

To find an object in the Object Viewer, it must be registered, as described in [“Registering the object” on page 790](#). 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 Program ID of a COM object. The Object Viewer also provides access to options for the operation of the object.

To view an object's properties:

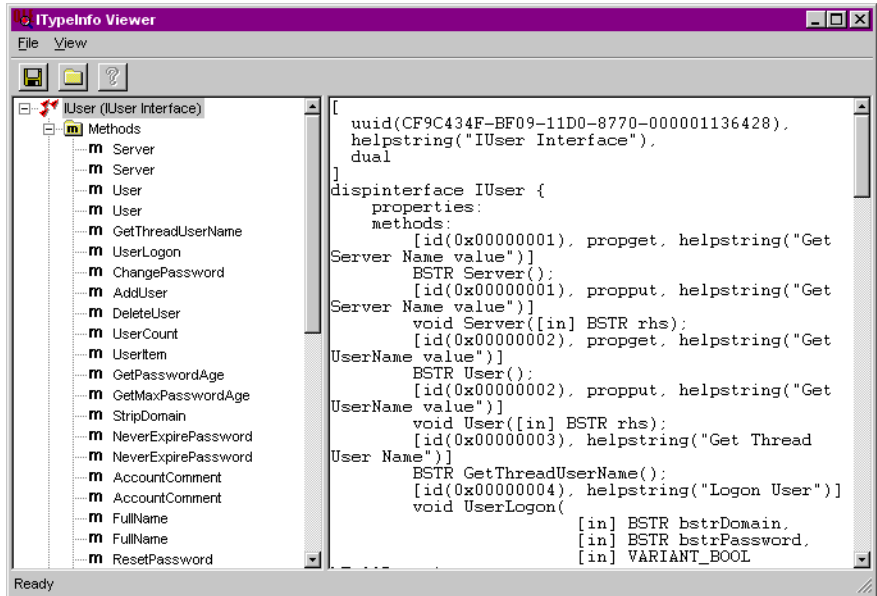
- 1 Open the Object Viewer, as shown in the following figure, and scroll to the object you want to examine.



- 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 `cfoject` 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 `cfoject` tag to create the Windows CDO (Collaborative Data Objects) for NTS NewMail object to send mail:

```
<cfoject 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 `cfoject` tag provides the following two ways to connect to COM objects:

- **Create method** (`cfoject action="Create"`) Takes a COM object, typically a DLL, and instantiates it prior to invoking methods and assigning properties.
- **Connect method** (`cfoject 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 <code>remote</code> , you must also use the <code>server</code> 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 [“Handling runtime exceptions with ColdFusion tags,” in Chapter 14.](#)

COM object considerations

When you use COM objects, consider the following to prevent and resolve errors:

- Ensure correct threading.
- Use input and output arguments correctly.
- Understand 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 [“Locking code with cflock,” in Chapter 15.](#)

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.

Getting started with CORBA

The ColdFusion `cobject` 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 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 `cobject` 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 `cobject` tag to create a CORBA object:

Attribute	Description
<code>type</code>	Must be CORBA. COM is the default.
<code>context</code>	Specifies the CORBA binding method, that is, how the object is obtained, as follows: <ul style="list-style-type: none">• <code>IOR</code> Uses a file containing the object's unique Interoperable Object Reference.• <code>NameService</code> Uses a naming service.

Attribute	Description
class	<p>Specifies the information required for the binding method to access the object.</p> <p>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.</p> <p>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"</p>
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 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 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:

```
<cfoject 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 cfoject tag:

```
handle = CreateObject("CORBA", "d:\temp\tester.ior", "IOR", "Visibroker")
```

For the complete cfoject and CreatObject 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 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 788](#). 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

When you use the `cfoobject` tag or the `CreateObject` function to create a CORBA object, ColdFusion creates a handle to a CORBA interface, which is identified by the `cfoobject` name attribute or the `CreateObject` function return variable. For example, the following CFML creates a handle named `myHandle`:

```
<cfoobject action = "create" type = "CORBA" context = "IOR"
           class = "d:\temp\tester.iior" name = "myHandle" locale="visibroker">
<cfset myHandle = CreateObject("CORBA", "d:\temp\tester.iior", "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 quotes, as shown in the following example:

IDL	<code>void method(in string a);</code>
CFML	<code><cfset foo="my string"></code> <code><cfset ret=handle.method(foo)></code>

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 quotes 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
octet	One-character string

IDL type	ColdFusion type
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 <code>coldfusion.runtime.corba.CorbaUserException</code>
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	<pre>module Tester { interface TManager { void testBoolean(in boolean a); void testOutBoolean(out boolean a); void testInoutBoolean(inout boolean a); boolean returnBoolean(); } }</pre>
CFML	<pre><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></pre>

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 considerations

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	<pre>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(); } }</pre>
CFML	<pre><CFSET handle = CreateObject("CORBA", "d:\temp\tester.ior", "IOR", "") > <cfset ret = handle.testEnum(1)></pre>

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	<pre>interface myInterface { exception PrimitiveException { long l; string s; float f; }; void testPrimitiveException() raises (PrimitiveException); }</pre>
------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

CFML	<pre><cftry> <cfset ret0 = handle.testPrimitiveException()> <cfcatch type=coldfusion.runtime.corba.CorbaUserException> <cfset exceptStruct= cfcatch.getContents()> <cfdump var ="#exceptStruct#"> </cfcatch> </cftry></pre>
-------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

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 loanAmount1
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 ColdFusion to access and use the following external services: mail servers, remote HTTP and FTP servers, and files and directories.

The following chapters are included:

Sending and Receiving E-Mail	809
Interacting with Remote Servers	829
Managing Files on the Server	845

CHAPTER 34

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 ColdFusion e-mail capability a vital link to your users.

Contents

- Using ColdFusion with mail servers..... 810
- Sending e-mail messages 811
- Sample uses of `cfmail`..... 813
- Customizing e-mail for multiple recipients 815
- Using `cfmailparam`..... 817
- Advanced sending options 818
- Receiving e-mail messages..... 819
- Handling POP mail..... 821

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.

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 Administrator to use the SMTP server. If you later need to override the SMTP server information, 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 Administrator, select **Server Settings > Mail Server**.
- 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 (Optional) Change the Server Port and Connection Timeout default settings.
- 6 Click Submit Changes.

ColdFusion saves the settings. The page displays a message indicating success or failure for connecting to the server.

For more information on the Administrator's mail settings, see *Administering ColdFusion MX*.

Sending SMTP e-mail with `cfmail`

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 SMTP mail messages 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 important `cfmail` tag attributes:

Attribute	Description
<code>subject</code>	The subject of the message.
<code>from</code>	The e-mail address of the sender.
<code>to</code>	The e-mail address of the recipient. Use a comma-delimited list to specify multiple recipients.
<code>cc</code>	(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.
<code>bcc</code>	(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.
<code>SpoolEnable</code>	(Optional) When <code>SpoolEnable="yes"</code> , saves a copy of the message until the sending operation is complete.

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 811](#).

Sample uses of cfmail

An application page containing the `cfmail` tag dynamically generates e-mail messages based on the tag's settings. Some of the tasks that you can accomplish with `cfmail` include the following:

- 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
- Sending a MIME file attachment with a mail message

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. The list is then sent, 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
<pre><cfoutput> #ProductRequests.FirstName# #ProductRequests.LastName# (#ProductRequests.Company#) - #ProductRequests.EmailAddress##013; </cfoutput></pre>	Presents a dynamic list embedded within a normal cfmail message, repeating for each row in the ProductRequests query. 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.

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
<pre><cfquery name="GetCustomers" datasource="myDSN"> SELECT * FROM Customers </cfquery></pre>	Retrieves all data from the Customers table into a query named GetCustomers.
<pre><cfmail query="GetCustomers" from="service@MyCompany.com" to="#GetCustomers.Email#" subject="Contact Info Verification"></pre>	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.
<pre>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#</pre>	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 cfmailparam

You use the `cfmailparam` tag to attach files or add a custom header to an e-mail message. You nest the `cfmailparam` tag within the `cfmail` tag.

Attaching files to a message

You 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.

Adding a custom header to a message

When the recipient of an e-mail message replies to the message, the reply is sent to the address specified in the `From` field of the original message. You can use `cfmailparam` to override the value in the `From` field and provide a `Reply-To` e-mail address. 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.

Advanced sending options

The ColdFusion implementation of SMTP mail uses a spooled architecture. 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 Server checks for spooled mail on messages on the Mail/Mail Logging page of the ColdFusion Administrator Server tab. (The default interval is 60 seconds.) If ColdFusion is extremely busy or has a large existing queue of messages, however, delivery can occur after the spool interval.

Sending mail as HTML

Most newer Internet mail applications are capable of reading and interpreting HTML code in a mail message. The `cfmail` tag lets you specify the message type as HTML. The `type="HTML"` attribute (the only valid value; the default is plain text) informs the receiving e-mail client that the message contains embedded HTML tags that must be processed. This feature is useful only when you are sending messages to mail clients that can interpret HTML. Also, you must escape any pound signs in the HTML, such as those used to specify colors, by using two `#` characters; for example, `bgcolor="###C5D9E5"`.

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:

- On Windows: `\CFusionMX\Mail\UnDelivr`
- On UNIX: `/opt/coldfusionmx/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.

For more information about the mail logging settings in the ColdFusion Administrator, see *Administering ColdFusion MX*.

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` on 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.

For more information on `cfpop` syntax and variables, see *CFML Reference*.

Using `cfpop`

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 ASCII 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 two variables that provide record number information:

- `RecordCount` The total number of records returned by the query.
- `CurrentRow` The current row of the query being processed by `cfoutput` or `cfloop` in a query-driven loop.

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 provides an example of each of the following uses of POP mail:

- Retrieving only message headers
- Retrieving a message
- Retrieving a message and its attachments
- Deleting messages

Retrieving only message headers

You can retrieve only the headers of your messages by using the `GetHeaderOnly` value for the `action` attribute of 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. The query object, whose name is specified in the `name` attribute of the `cfpop` tag, consists of the following fields:

- `date`
- `from`
- `messageNumber`
- `replyTo`
- `subject`
- `cc`
- `to`

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: #HTMLFormat(Sample.messageNumber)# <br>
    To: #HTMLFormat(Sample.to)# <br>
    From: #HTMLFormat(Sample.from)# <br>
    Subject: #HTMLFormat(Sample.subject)# <br>
    Date: #HTMLFormat(Sample.date)#<br>
    Cc: #HTMLFormat(Sample.cc)# <br>
    ReplyTo: #HTMLFormat(Sample.replyTo)# <br><br>
</cfoutput>

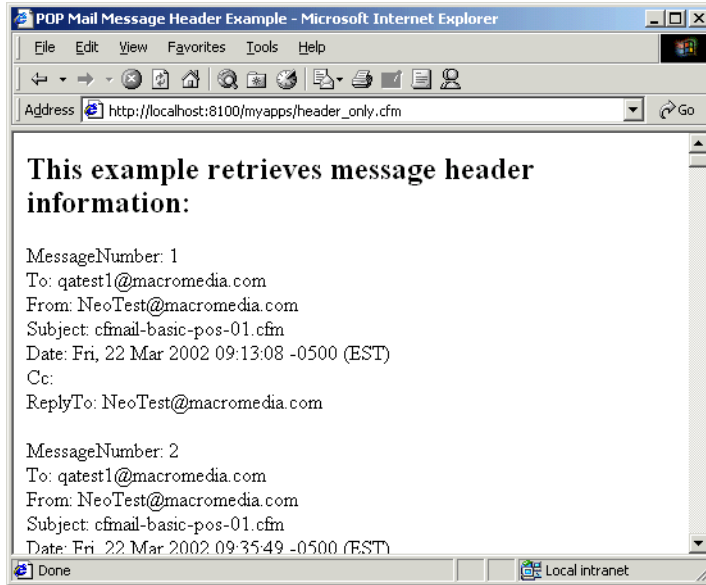
</body>
```

```
</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 461.

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 a message

When you use the `cfpop` tag with `action="GetAll"`, ColdFusion returns the same columns as with `getheaderonly`, plus two additional columns, `body` and `header`.

To retrieve an entire message:

- 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">

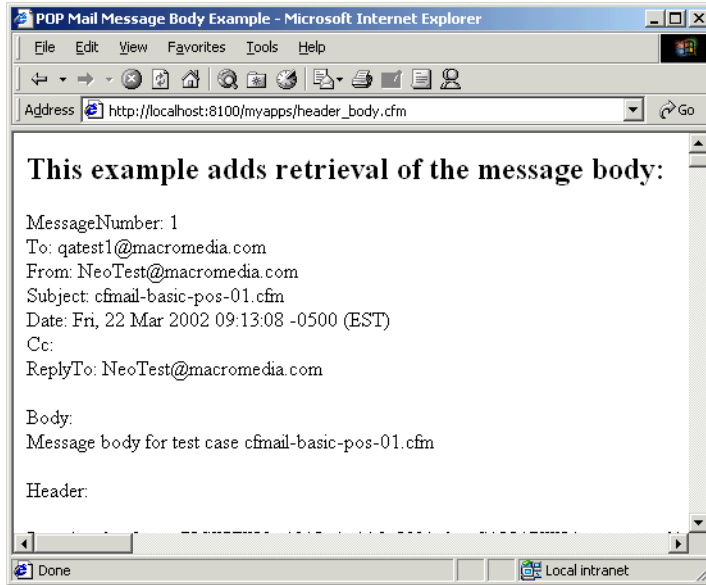
<cfoutput query="Sample">
  MessageNumber: #HTMLFormat(Sample.messageNumber)# <br>
  To: #Sample.to# <br>
  From: #HTMLFormat(Sample.from)# <br>
  Subject: #HTMLFormat(Sample.subject)# <br>
  Date: #HTMLFormat(Sample.date)#<br>
  Cc: #HTMLFormat(Sample.cc)# <br>
  ReplyTo: #HTMLFormat(Sample.replyTo)# <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 a message and its attachments

When you use the `cfpop` tag with `action="getAll"`, and use the `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 also adds the following two columns to the query it creates:

- `attachments` Contains a tab-separated list of all attachment names.
- `attachmentfiles` Contains a tab-separated list of the locations of the attachment files. Use the `cffile` tag to delete these temporary files.

You must ensure that the `attachmentpath` directory exists before you use the `cfpop` tag to retrieve attachments. ColdFusion generates an error if it attempts to write an attachment file to a nonexistent directory.

If a message has no attachments, the `attachments` and `attachmentfiles` columns contain empty strings.

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: #HTMLFormat(Sample.MessageNumber)# <br>
  To: #HTMLFormat(Sample.to)# <br>
  From: #HTMLFormat(Sample.from)# <br>
  Subject: #HTMLFormat(Sample.subject)# <br>
  Date: #HTMLFormat(Sample.date)# <br>
  Cc: #HTMLFormat(Sample.cc)# <br>
  ReplyTo: #HTMLFormat(Sample.ReplyTo)# <br>
  Attachments: #HTMLFormat(Sample.Attachments)# <br>
  Attachment Files: #HTMLFormat(Sample.AttachmentFiles)# <br>
  <br>
  Body:<br>
  #Sample.body# <br>
  <br>
  Header:<br>
  HTMLCodeFormat(Sample.header)# <br>
  <br>
</cfoutput>

</body>
</html>

```

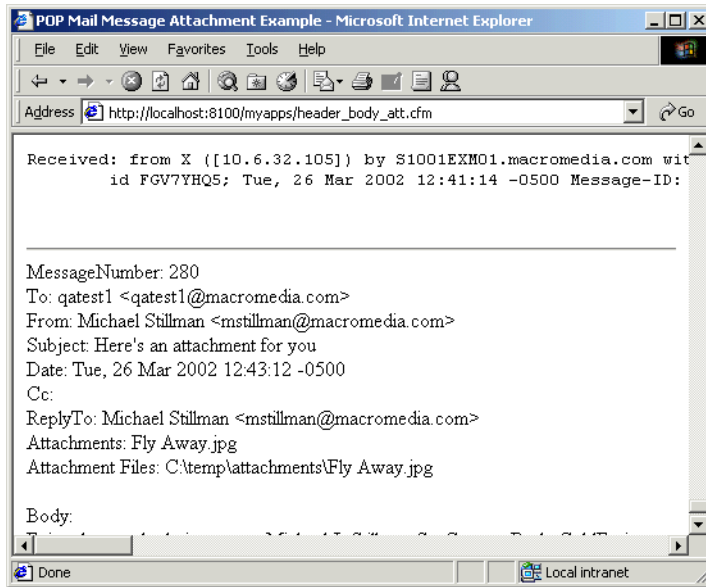
- 2 Edit the following lines so that they refer to valid values for your POP mail server, username, and password:

```

<cfpop server="mail.company.com"
  username=#myusername#
  password=#mypassword#

```

- 3 Save the file as `header_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

By default, retrieved messages remain on the POP mail server. If you want to delete retrieved messages, you must set the `action` attribute of the `cfpop` tag to `Delete`. You must also specify use the `messagenumber` attribute to specify the numbers of the messages to delete.

Using `cfpop` to delete a message permanently removes it from the server. If the `messagenumber` does not correspond to a message on the server, ColdFusion generates an error.

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>
```

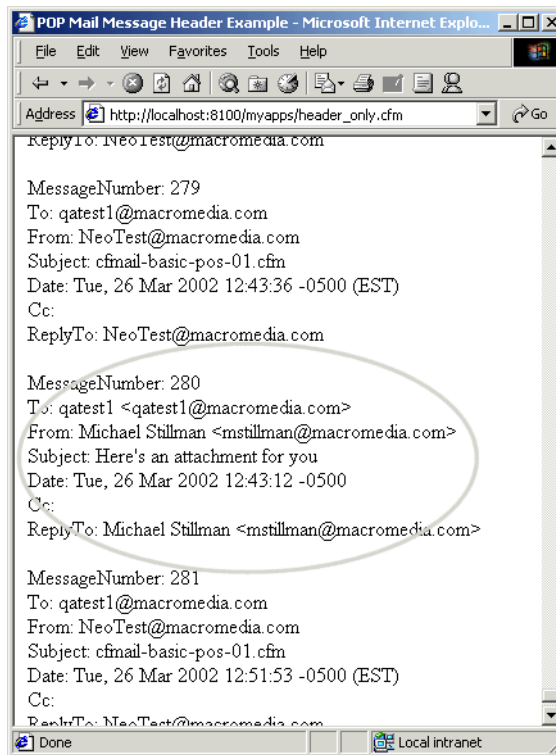
- 2 Edit the following lines so that they refer to valid values for your POP mail server, username, and password:

```
<cfpop server="mail.company.com"
  username=#username#
  password=#password#
```

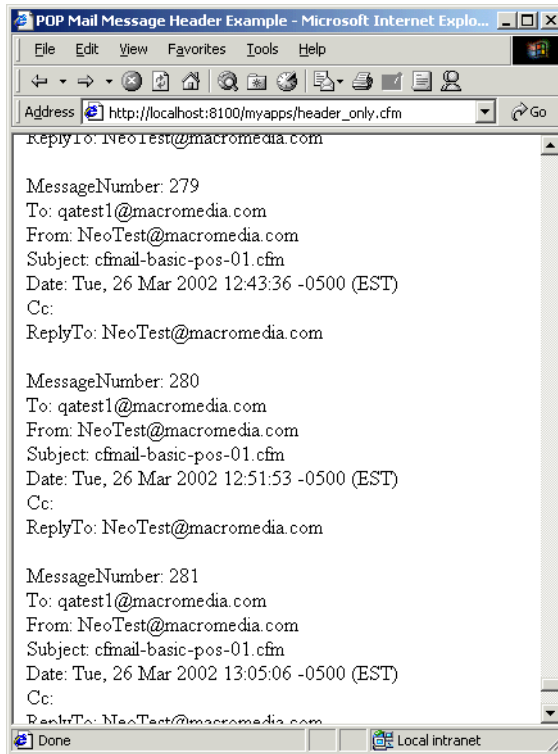
- 3 Save the file as message_delete.cfm in the myapps directory under your *web_root* and view the file in your web browser.

Caution: When you view this page in your web browser, ColdFusion immediately deletes the messages from the POP server.

The following figure shows the message list before the deletion of message 280:



The following figure shows the message list after the deletion of message number 280. ColdFusion reorders the remaining messages:



CHAPTER 35

Interacting with Remote Servers

This chapter describes how ColdFusion 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.....](#) 830
- [Using cfhttp to interact with the web.....](#) 830
- [Creating a query object from a text file](#) 835
- [Using the cfhttp Post method](#) 837
- [Performing file operations with cfftp.....](#) 841

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. Once 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"
```



```

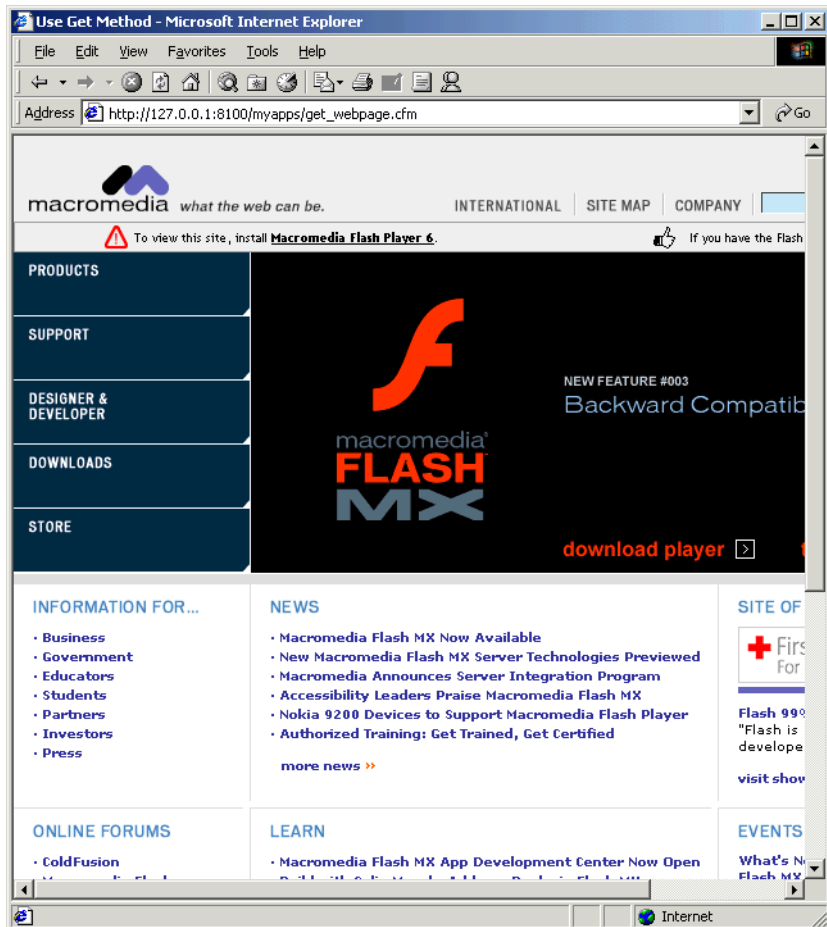
url="http://www.macromedia.com"
resolveurl="Yes">
<cfoutput>
  #cfhttp.FileContent# <br>
</cfoutput>

</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
<pre><cfhttp method="Get" url="http://www.macromedia.com" resolveurl="Yes"></pre>	Get the page specified in the URL and make the links absolute instead of relative so that they appear properly.
<pre><cfoutput> #cfhttp.FileContent#
 </cfoutput></pre>	Display the page, which is stored in the variable <code>cfhttp.fileContent</code> , in the browser.

To get a web page and save it in a file:

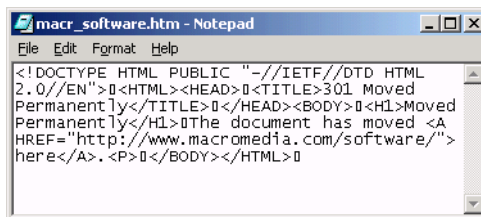
- 1 Create a ColdFusion page with the following content:

```
<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
<pre><cfhttp method = "Get" url="http://www.macromedia.com/software" path="c:\temp" file="macr_software.htm"></pre>	<p>Get the page specified in the URL and save it in the file specified by the path and file attributes.</p> <p>When you use the path and file attributes, ColdFusion ignores any <code>resolveurl</code> attribute. As a result, frames and other included files cannot appear when you view the saved page.</p>

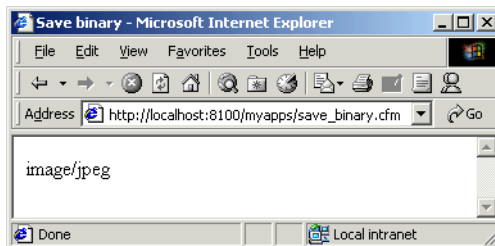
To get a binary file and save it:

- 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 view it in the web browser.

The MIME content type appears in your browser:



- 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
<pre><cfhttp method="Get" url="http://www.macromedia.com/macromedia/ accessibility/images/spotlight.jpg" path="c:\temp" file="My_SavedBinary.jpg"></pre>	Get a binary file and save it in the path and file specified.
<pre><cfoutput> #cfhttp.MimeType# </cfoutput></pre>	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>
```

```

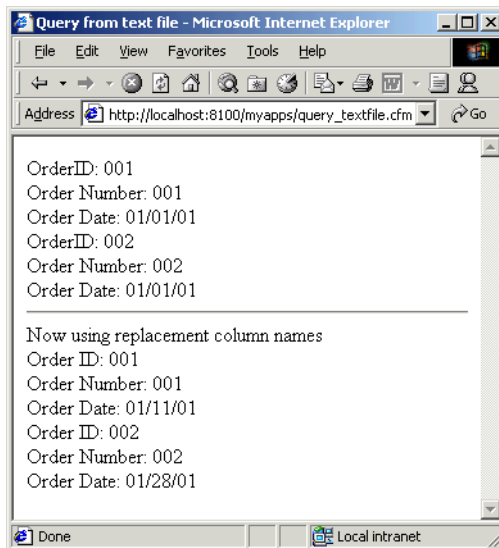
<!-- Now substitute different column names -->
<!-- by using the columns attribute -->
<hr>
Now using replacement column names<br>

<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="cgi var "
    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>
<br>
Mime Type: #cfhttp.MimeType#<br>
</cfoutput>
</body>
</html>
```

- 2 Replace the path to the GIF file to a path on your server (this is 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
<pre><cfhttp method="Post" url="http://127.0.0.1:8500/myapps/ post_test_server.cfm"></pre>	Post an HTTP request to the specified page.
<pre><cfhttpparam type="Cookie" value="cookiemonster" name="mycookie6"></pre>	Send a cookie in the request.
<pre><cfhttpparam type="CGI" value="cgivar " name="mycgi"></pre>	Send a CGI variable in the request.
<pre><cfhttpparam type="URL" value="theurl" name="myurl"></pre>	Send a URL in the request.
<pre><cfhttpparam type="Formfield" value="twriter@macromedia.com" name="emailaddress"></pre>	Send a Form field in the request.
<pre><cfhttpparam type="File" name="myfile" file="c:\pix\trees.gif"></pre>	Send a file in the request. The <code></cfhttp></code> tag ends the http request.
<pre><cfoutput> File Content:
 #cfhttp.filecontent#
</pre>	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 <code>cfoutput</code> tag in <code>server.cfm</code> .
<pre>Mime Type: #cfhttp.MimeType#
 </cfoutput></pre>	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
<pre><cffile destination="C:\temp\" nameconflict="Overwrite" filefield="Form.myfile" action="Upload" attributes="Normal"></pre>	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.
<pre><cfoutput></pre>	Output information. The results are not displayed by this page. They are passed back to the posting page in its cfhttp.filecontent variable.
<pre>The URL variable is: #URL.myurl#
</pre>	Output the value of the URL variable sent in the HTTP request.
<pre>The Cookie variable is: #Cookie.mycookie#
</pre>	Output the value of the Cookie variable sent in the HTTP request.

Code	Description
The CGI variable is: <code>#CGI.mycgi#</code> 	Output the value of the CGI variable sent in the HTTP request.
The Form variable is: <code>#Form.emailaddress#</code> . 	Output the Form variable sent in the HTTP request. You send the variable using the <code>type="formField"</code> attribute but the receiving page gets it as a Form variable.
The file was uploaded to <code>#File.ServerDirectory#\#File. ServerFile#</code> . </cfoutput>	Output the results of the <code>cffile</code> 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`, the `Enable cfftp Tag` option must be selected on the `Tag Restrictions` page of the `Basic Security` section of the `ColdFusion Administrator Security` tab.

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
<pre> <cfftp connection="Myftp" server="MyServer" username="MyUserName" password="MyPassword" action="Open" stoponerror="Yes"> </pre>	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.
<pre> <cfftp connection=Myftp action="GetCurrentDir" stoponerror="Yes"> <cfoutput> The current directory is: #cfftp.returnvalue#<p> </cfoutput> </pre>	Use the Myftp connection to get the name of the current directory; stop processing if an error occurs. Display the current directory.
<pre> <cfftp connection=Myftp action="ListDir" directory="#cfftp.returnvalue#" name="dirlist" stoponerror="Yes"> </pre>	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.

Code	Description
<pre><cfftp action="close" connection="Myftp"> <p>Did the connection close successfully? <cfoutput>#cfftp.succeeded#</cfoutput></p></pre>	<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 <code>cfftp.succeeded</code> variable, which is Yes if the connection is closed, and No if the operation failed.</p>
<pre><cftable query="dirlist" colheaders="yes" htmltable> <cfcol header="Name" TEXT="#name#"> <cfcol header="Path" TEXT="#path#"> <cfcol header="URL" TEXT="#url#"> <cfcol header="Length" TEXT="#length#"> <cfcol header="LastModified" TEXT="#DateFormat(lastmodified)#"> <cfcol header="IsDirectory" TEXT="#isdirectory#"> </cftable></pre>	<p>Display a table with the results of the ListDir FTP command.</p>

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 315.

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 `cftp` 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 36

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 841.

Contents

- [About file management](#)..... 846
- [Using cffile](#) 846
- [Using cfdirectory](#) 856
- [Using cfcontent](#) 858

About file management

ColdFusion 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 Administrator. Also, to access files that are not located on the local ColdFusion Server 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

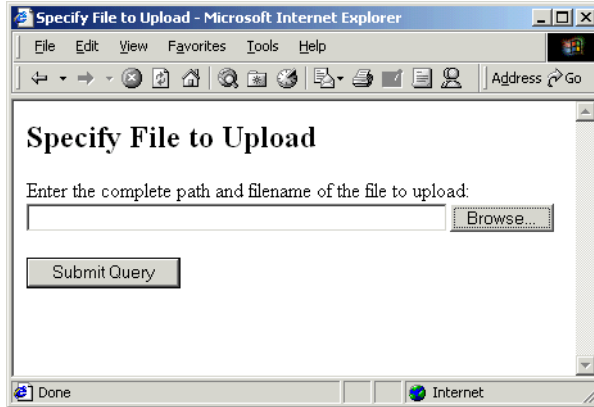
The following procedures describe how to create these files.

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
<pre><form action="uploadfileaction.cfm" enctype="multipart/form-data" method="post"></pre>	Create a form that contains file selection fields for upload by the user. The <code>action</code> attribute value specifies the ColdFusion template that will process the submitted form. The <code>enctype</code> attribute value tells the server that the form submission contains an uploaded file. The <code>method</code> attribute is set to <code>post</code> to submit a ColdFusion form.
<pre><input type="file" name="FiletoUpload" size="45"></pre>	Allow the user to specify the file to upload. The <code>file</code> type instructs the browser to prepare to read and transmit a file from the user's system to your server. It automatically includes a <code>Browse</code> button to allow the user to 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>

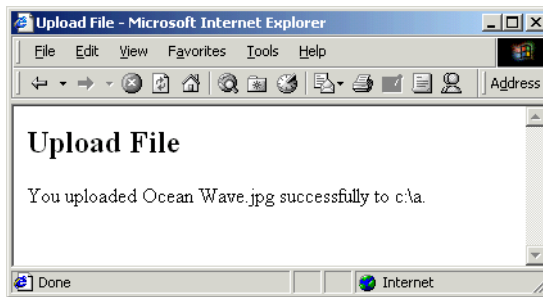
</body>
</html>

```

- 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, as the following figure shows:



Reviewing the code

The following table describes the code and its function:

Code	Description
<code><cffile action="upload"</code>	Output the name and location of the uploaded file on the client machine.
<code>destination="c:\temp\"</code>	Specify the destination of the file.
<code>nameConflict="overwrite"</code>	If the file already exists, overwrite it.
<code>fileField="Form.FiletoUpload"></code>	Specify the name of the file to upload. Do not enclose the variable in pound signs.
<code>You uploaded #cffile.ClientFileName#.#cffile.ClientFileExt# successfully to #cffile.ServerDirectory#.</code>	Inform the user of the file that was uploaded and its destination. For information on <code>cffile</code> scope variables, see “Evaluating the results of a file upload” on page 850 .

Note: This example performs no error checking and does not incorporate any security measures. Before deploying an application that performs file uploads, be sure to incorporate both error handling and security. For more information, see [Chapter 16, “Securing Applications” on page 347](#) and [Chapter 14, “Handling Errors” on page 281](#).

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 850](#).

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: Modern versions of Internet Explorer and Netscape support MIME type associations. Other browsers and older 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/pjpeg"` 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:

- Archive
- Hidden
- Normal
- ReadOnly
- System

To specify several attributes in CFML, use a comma-separated list for the `attributes` attribute; for example, `attributes="ReadOnly,Archive"`. If you do not use `attributes`, 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
<code>attemptedServerFile</code>	Initial name that ColdFusion uses when attempting to save a file; for example, <code>myfile.txt</code> . (see “Resolving conflicting filenames” on page 849).
<code>clientDirectory</code>	Directory on the client’s system from which the file was uploaded.
<code>clientFile</code>	Full name of the source file on the client’s system with the file extension; for example, <code>myfile.txt</code> .
<code>clientFileName</code>	Name of the source file on the client’s system without an extension; for example, <code>myfile</code> .
<code>clientFileExt</code>	Extension of the source file on the client’s system without a period; for example, <code>txt</code> (not <code>.txt</code>).
<code>contentType</code>	MIME content type of the saved file; for example, <code>image/gif</code> .
<code>contentSubType</code>	MIME content subtype of the saved file; for example, <code>gif</code> for <code>image/gif</code> .
<code>dateLastAccessed</code>	Date that the uploaded file was last accessed.
<code>fileExisted</code>	Indicates (Yes or No) whether the file already existed with the same path.
<code>fileSize</code>	Size of the uploaded file.
<code>fileWasAppended</code>	Indicates (Yes or No) whether ColdFusion appended the uploaded file to an existing file.
<code>fileWasOverwritten</code>	Indicates (Yes or No) whether ColdFusion overwrote a file.
<code>fileWasRenamed</code>	Indicates (Yes or No) whether the uploaded file was renamed to avoid a name conflict.
<code>fileWasSaved</code>	Indicates (Yes or No) whether ColdFusion saved the uploaded file.
<code>oldFileSize</code>	Size of the file that was overwritten in the file upload operation. Empty if no file was overwritten.
<code>serverDirectory</code>	Directory where the file was saved on the server.
<code>serverFile</code>	Full name of the file saved on the server with the file extension; for example, <code>myfile.txt</code> .

Variable	Description
serverFileName	Name of the file saved on the server without an extension; for example, myfile.
serverFileExt	Extension of the file saved on the server without a period; for example, txt (not .txt).
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	<pre><cffile action="move" source="c:\files\upload\KeyMemo.doc" destination="c:\files\memo\ "></pre>
Rename a file	<pre><cffile action="rename" source="c:\files\memo\KeyMemo.doc" destination="c:\files\memo\OldMemo.doc"></pre>
Copy a file	<pre><cffile action="copy" source="c:\files\upload\KeyMemo.doc" destination="c:\files\backup\ "></pre>
Delete a file	<pre><cffile action="delete" file="c:\files\upload\oldfile.txt"></pre>

This example sets the archive bit for the uploaded file:

```
<cffile action="Copy"
  source="c:\files\upload\keymemo.doc"
  destination="c:\files\backup\ "
  attributes="Archive">
```

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:

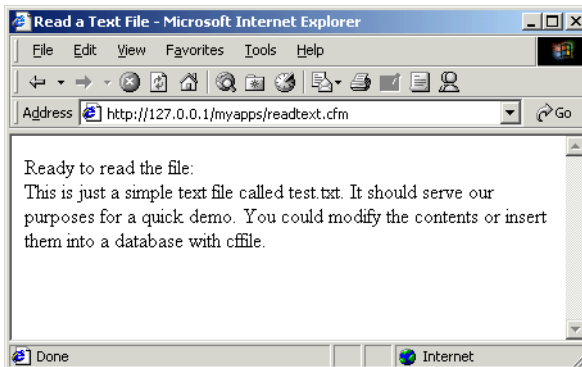
- 1 Create a ColdFusion page with the following content:

```
<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).

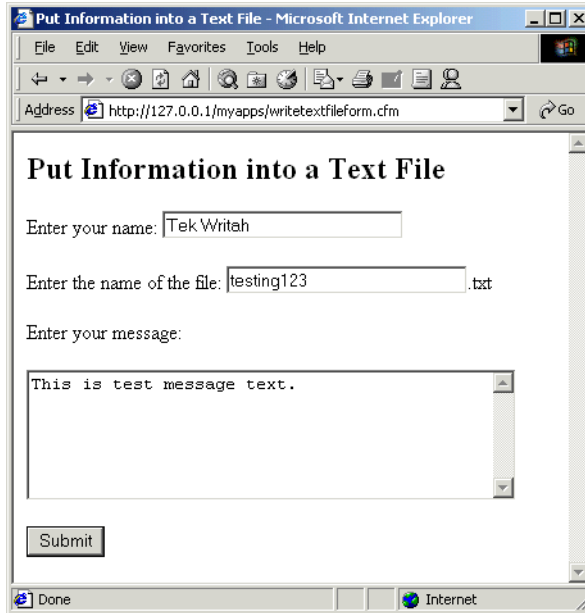
To write a text file:

- 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; as shown in the following figure.



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 in ColdFusion Studio.
- 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.

As with `cffile`, you can disable `cfdirectory` processing in the ColdFusion 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 five result columns that you can reference in a `cfoutput` tag, using the `name` attribute:

- `name` Directory entry name.
- `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>
```

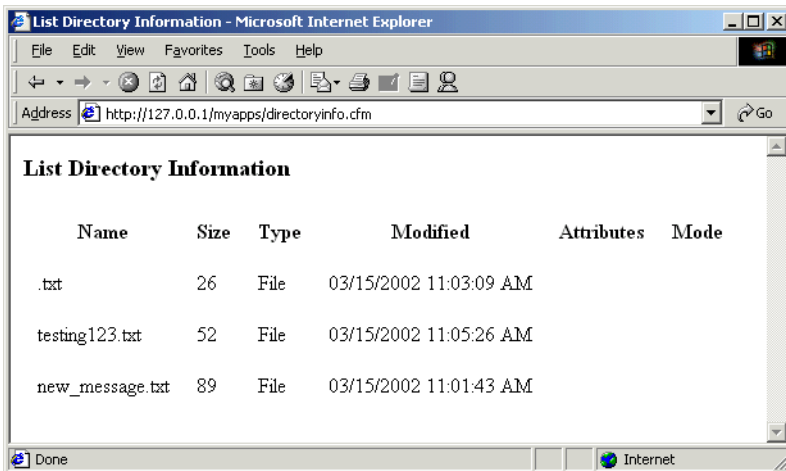
```

        <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 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>
```

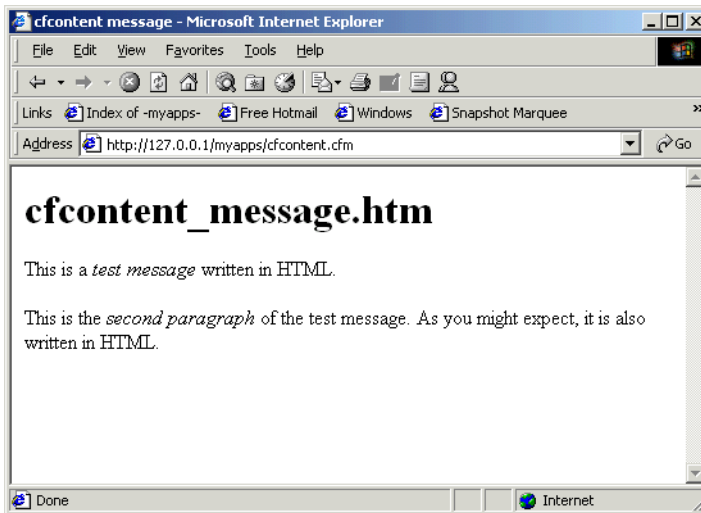
```

<cfcontent
  type = "text/html"
  file = "C:\CFusionMX\wwwroot\myapps\cfcontent_message.htm"
  deleteFile = "No">
</body>
</html>

```

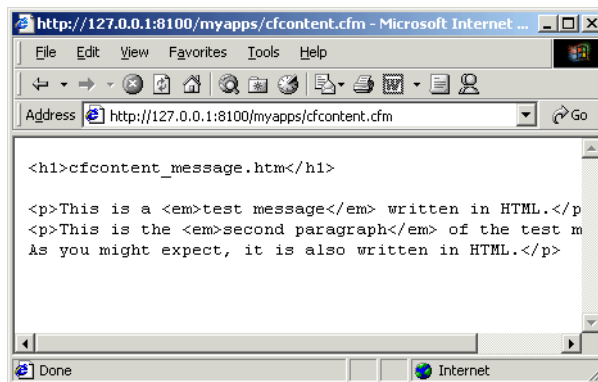
- 4 If necessary, edit the `file =` line to point to your `myapps` directory.
- 5 Save the file as `cfcontent.cfm` in the `myapps` directory under your `web_root` and view it in the browser.

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">

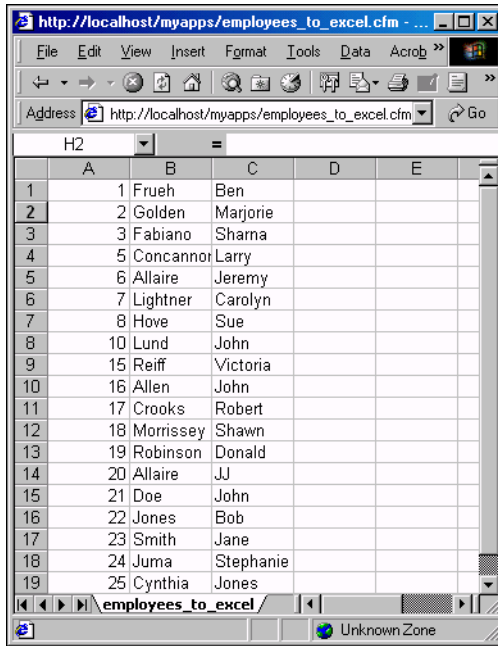
<!-- get employee info -->
<cfquery name="GetEmps" datasource="CompanyInfo">
    SELECT * FROM Employees
</cfquery>

<!-- set vars for special chars -->
<cfset TabChar = Chr(9)>
<cfset NewLine = Chr(13) & Chr(10)>
<!-- set content type to invoke Excel -->
<cfcontent type="application/msexcel">

<!-- suggest default name for XLS file -->
<!-- use "Content-Disposition" in cfheader for
Internet Explorer -->
<cfheader name="Content-Disposition" value="filename=Employees.xls">
<!-- output data using cfloop & cfoutput -->
<cfloop query="GetEmps">
    <cfoutput>#Employee_ID#&#TabChar##LastName#
    #&#TabChar##FirstName##&#TabChar##Salary##NewLine#</cfoutput>
</cfloop>
```

- 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:



The screenshot shows a web browser window with the address bar displaying `http://localhost/myapps/employees_to_excel.cfm`. The browser's menu bar includes File, Edit, View, Insert, Format, Tools, Data, and Acrobat. The address bar also shows a 'Go' button. The main content area displays an Excel spreadsheet with the following data:

	A	B	C	D	E
1	1	Frueh	Ben		
2	2	Golden	Marjorie		
3	3	Fabiano	Sharna		
4	5	Concannon	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		

The spreadsheet is displayed in a window titled `employees_to_excel`. The status bar at the bottom indicates an 'Unknown Zone'.

Special

, in search expressions 562

A

accessing

client variables 325, 331
generated content 210

action pages 585

ActionScript

on ColdFusion server 6

Active Server Pages 717

adding

data elements to structures 102
elements to an array 92

ancestor tags

data access 215
definition 212

AND operator, SQL,

definition 421

appearance, of charts 658

application 316

application framework

about 318
approaches to 266
custom error pages 293
mapping 265

application pages

errors 293
variables 22

Application scope 23, 56, 264, 316

JSP pages 765

application security, example 362

application servers, data exchange

across 718

application variables

configuring 333
description 264, 316
listing 335

usage tips 334

using 333

Application.cfm file

application-level settings 268

creating 268

example 270

processing 265

user-defined functions in 184

application-defined exception 286

application-level settings 263

applications

Application.cfm 265

authentication 354

caching 272

defaults 269

defined 262

directory structure 265, 266

elements of 262

error handling 270

framework 262

globalization of 374

in ColdFusion 262

internationalization 374

JSP tags 764

localization 374

login 269

naming 268

OnRequestEnd.cfm 265

optimizing 272

optimizing database access 277

page settings 269

reusable elements 264

security 264, 348

servlets in 764

shared variables 264

stored procedures in 277

storing variables 333

user security 362

variable options, setting 268

applications, unnamed 765

applicationToken 355

area chart

example 664

arguments

optional 175, 179

passing 180

user-defined function 180

using function names 186

Arguments scope 23, 56

as array 172

as structure 173

user-defined functions and 171

arithmetic operators 66

array variables 41

ArrayAppend CFML function 92

ArrayDeleteAt CFML function 93

ArrayInsertAt CFML function 93

ArrayNew CFML function 91

ArrayPrepend CFML function 93

arrays

2-dimensional 88

3-dimensional 89

adding data to 90

adding elements to 90, 92

as variables 41

copying 94

creating 90

description 88

dimensions 88

elements 88

elements, adding 92

elements, deleting 93

functions 98

in dynamic expressions 76

index 88

multidimensional 91

- passing to functions 189
 - populating 95
 - referencing elements in 90
 - resizing 93
 - user-defined functions and 189
- ArraySet CFML function 95
- ArraySort CFML function 104
- ASCII 377
- assignment, CFScript
 - statements 122
- attributes
 - for custom tags 203
 - passing values 202, 203
- Attributes scope 22, 55
- authentication
 - defined 351
 - example 363
 - login 354
 - logout 355
 - scenarios 357
 - web servers and 353
- authorization
 - defined 351
 - web servers and 353
- AVG SQL function 652
- B**
- BabelFish 736
- backreferences
 - about 144
 - case conversions with 145
 - in regular expression searches 640
 - in replacement strings 144
 - omitting from 146
- bar charts, specifying
 - appearance 660
- base tags 212
- Base64 variables 40
- basic authentication 353
 - scenarios 356
 - web services and 750
- basic exception types 285
- BETWEEN SQL operator 421
- binary data type 35
- binary files, saving 833
- binary variables 40
- BOM
 - Dreamweaver MX and 379
 - setting 379
 - using 379
- Boolean
 - operators 67
 - variables 38
- break, CFScript statement 128
- browsers
 - cform considerations 610
 - displaying e-mail in 819
 - transferring data to a server 723
- building
 - drop-down list boxes 619
 - queries 436
 - search interfaces 591
 - slider bar controls 621
 - text entry boxes 620
 - tree controls 611
- built-in variables
 - client 326
 - custom tags 208
 - server 335
 - session 330
- C**
- C++ CFX tags
 - implementing 256
 - LD_LIBRARY_PATH 256
 - registering 257
 - SHLIB_PATH 256
- C++ development environment 256
- caching
 - applications 272
 - attributes 273
 - flushing pages 273
 - locations of 273
 - page areas 274
 - pages 272
 - to variables 274
- caching connections 843
- Caller scope 22, 55
- calling
 - CFX tags 247
 - COM objects 788
 - CORBA objects 797
 - Java objects 769
 - nested objects 771, 789
 - object methods 770, 788
 - user-defined functions 177
- case sensitivity, of CFML 16
- catching security exceptions 357
- cfabort tag
 - about 30
 - OnRequestEnd.cfm 265
- cfapplet tag
 - description 609
 - using 633, 635
- cfapplication tag 321
- cfassociate tag 214
- cfbreak tag 28
- cfcache tag 273
 - location of tag 273
- cfcase tag 28
- cfcatch tag 299, 300
- cfchart tag 647
 - charting queries 650
 - common attributes 658
 - for bar charts 660
 - for pie charts 662
 - introduced 646
- cfchartdata tag 647, 654
- cfchartseries tag 647
- cfcollection tag 535
- cfcontent tag 841, 858
 - Excel spreadsheet 860
- cfdefaultcase tag 28
- cfdirectory tag 856
 - and queries 462
 - for file operations 841
- cfelse tag 27
- cfelseif tag 27
- cfencode utility 207
- cferror page 293
- cfexit tag 30
 - and OnRequestEnd.cfm 265
 - behavior of 211
- cffile tag 846
- cfflush tag 280
 - and HTML headers 595
 - using 594
- cfform controls
 - browser considerations 610
 - description 608
- cfform tag
 - passthrough attribute 608
 - usage notes 609
 - using with HTML 608
- cfftp tag
 - attributes 844
 - connection actions 844
 - using 841
- cffunction tag 353
 - attributes 177

- creating user-defined functions 170
 - web services, publishing 744
- cfgrid tag
 - browser considerations 610
 - controlling cell contents 625
 - editing data in 627
 - handling failed validation 642
 - returning user edits 626
 - using 622
 - validating with JavaScript 642
- cfhttp tag
 - and queries 462
 - creating queries 835
 - Get method 830
 - Post method 837
 - using 830
- cfhttpparam tag 837
- CFID
 - cookie 320
 - server-side variable 322
- cfif tag 27
- cfimport tag
 - about 200
 - calling custom tags 200
- cfinclude tag
 - about 158
 - recommendations for 160
 - using 159
- cfindex tag
 - and queries 462
 - external attribute 530
- cfinput tag
 - handling failed validation 642
 - passthrough attribute 608
 - validating with JavaScript 642
- cfinsert tag
 - creating action pages 448
 - form considerations 448
 - inserting data 446
- cfinvoke tag
 - example 738
 - web services, consuming 736, 737
- cfinvokeargument tag 738
- cfldap tag 495
 - and queries 462
 - indexing queries 545
 - output 514
 - queries and 514
- Verity and 514
- cflocation tag 327
- cflock tag
 - controlling time-outs 340
 - examples 343
 - for file access 346
 - name attribute 340
 - nesting 341
 - scope attribute 339
 - throwOnTimeout attribute 340
 - time-out attribute 340
 - using 336, 338
- cflog tag 297
- cflogin structure 356
- cflogin tag 353, 354, 355
- cfloginuser tag 353
- cflogout tag 353
- cfloop tag
 - about 28
 - emulating in custom tags 211
 - nested 96
- cfmail tag
 - attributes 811
 - sample uses 813
 - sending mail as HTML 818
- cfmailparam tag 817
- CFML
 - case sensitivity 16
 - CFScript 26
 - Code Compatibility Analyzer 409
 - code validation 409
 - comments 31
 - components 25
 - constants 21
 - converting data to JavaScript 723
 - data types 24
 - debugging 402
 - description 4
 - development tools 5
 - elements 16
 - expressions 21, 66
 - extending 244
 - extensions 5
 - flow control 27
 - functions 5, 19
 - reserved words 32
 - special characters 31
 - syntax errors 410
 - tags 4, 17
 - variables 22
- CFML functions
 - ArrayAppend 92
 - ArrayDeleteAt 93
 - ArrayInsertAt 93
 - ArrayNew 91
 - ArrayPrepend 93
 - ArraySet 95
 - ArraySort 104
 - AtructKeyArray 104
 - CreateObject 736
 - CreateTimeSpan 279, 329, 466
 - DateFormat 605
 - DeleteClientVariablesList 327
 - DollarFormat 605
 - dynamic evaluation 77
 - evaluate 78
 - for arrays 98
 - for queries 462
 - for structures 113
 - formatting data 589
 - GetClientVariablesList 326
 - GetLocale 378
 - HTMLEditFormat 726, 822
 - IIF 80
 - IsAuthenticated 357
 - IsCustomFunction 188
 - IsDefined 60, 104, 587, 603
 - IsStruct 103
 - JavaCast 776
 - ListQualify 600, 602
 - ListSort 104
 - MonthAsString 95
 - Rand 596
 - RandRange 596
 - REFind 147
 - REFindNoCase 147
 - set Encoding 386
 - SetLocale 378
 - setVariable 80
 - StructClear 107
 - StructCount 103
 - StructDelete 107
 - StructIsEmpty 103
 - StructKeyExists 104
 - StructKeyList 104
 - StructNew 102
 - syntax 70
 - URLEncodedFormat 411

- CFML syntax
 - Code Compatibility Analyzer 409
- cfmodule tag 199
 - calling custom tags 199
- cfoutput tag
 - data-type conversions 50
 - populating list boxes 597
 - use with cfquery tag 435
 - use with component objects 769, 788
- cfparam tag 61, 325
 - testing and setting variables 61
 - validating data types 62
- cfpop tag
 - and queries 462
 - query results 550
 - query variables 820
 - using 819
 - using cfindex with 545
- cfprocessingdirective tag 380
- cfquery tag
 - cachedWithin attribute 279
 - creating action pages 449, 457
 - debugging with 402
 - populating list boxes 597
 - syntax 435
 - using 435
 - using cfindex with 545
- cfrethrow tag
 - about 301, 309
 - nesting 310
 - using 309
- cfsavecontent tag 274
- CFScript
 - comments 119
 - conditional processing 122
 - creating user-defined functions 169
 - description 5
 - differences from JavaScript 120
 - example 116, 130
 - exception handling 129
 - expressions 118
 - function statement 174
 - introduction 26
 - language 118
 - looping 124
 - reserved words 120
 - return statement 175
 - statements 118
 - user-defined function syntax 174
 - using 116
 - var statement 175
 - variables 118
 - web services, consuming 739
- CFScript syntax
 - for user-defined functions 174
- cfsearch properties 542
- cfsearch tag
 - about 522
 - external attribute 530
- cfselect tag
 - handling failed validation 642
 - passthrough attribute 608
 - populating list boxes 619
- cfset tag
 - and component objects 769, 788
 - creating variables 34
- cfsetting tag
 - debugging with 402
- cfslider tag
 - browser considerations 610
 - description 608
 - handling failed validation 642
 - validating with JavaScript 642
- cfstat
 - enabling 391
 - Windows NT and 391
- cfstoredproc tag 277
- cfswitch tag 28
- cftextInput tag
 - browser considerations 610
 - handling failed validation 642
 - validating with JavaScript 642
- cfthrow tag
 - nesting 310
 - using 309
- CFToken
 - Cookie 320
 - server-side variable 322
- cftrace tag 404
 - attributes 407
 - using 404
- cfree tag
 - browser considerations 610
 - description 609
 - form variables 613
 - handling failed validation 642
 - image names 616
 - URLs in 617
 - validating with JavaScript 642
- cftry tag 299
 - example 304
 - nesting 310
- cfupdate tag
 - creating action pages 455
 - using 455
- cfwddx tag 718
- CFX tags
 - calling 164, 247
 - cfx.jar 247
 - compiling 256
 - creating in Java 247
 - debugging in C++ 257
 - debugging in Java 253
 - description 244
 - developing in C++ 256
 - Java 245
 - LD_LIBRARY_PATH 256
 - locking 346
 - locking access to 336, 340
 - recommendations for 164
 - registering 257
 - sample C++ 256
 - sample Java 245
 - scopes and 59
 - SHLIB_PATH 256
 - testing Java 248
 - using 164
- cfx.jar 247
- cfxml tag 694
- CGI
 - and cfhttp Post method 830
 - returning results to 840
- CGI scope 22, 56
- character classes 143
- character encodings
 - COM 388
 - CORBA 388
 - databases 387
 - e-mail 387
 - files 387
 - forms 385, 386
 - introduction 377
 - LDAP 388
 - search 388
 - Unicode 377
 - WDDX 388

- character sets
 - default 380
 - determining in ColdFusion
 - page 379
 - introduction 375
 - of output 380
 - setting for output 380
- charting 650
 - individual data points 653
- charts
 - 3-D 659
 - administering 649
 - appearance attributes 658
 - area 664
 - background color 658
 - bar charts 660
 - border 658
 - caching 649
 - column labels 658
 - curve chart considerations 666
 - data markers 660
 - dimensions 658
 - drill-down 667
 - embedding URLs 667
 - example 661
 - file type 658
 - foreground 658
 - labels 658
 - linking from 667
 - markers 659
 - multiple series 659
 - paint 660
 - pie chart 662
 - referencing JavaScript 667
 - rotation 659
 - threads 649
 - tips 659
 - types 646
- check boxes
 - errors 587
 - lists of values 599
 - multiple 599
- child tags 212
- class loading
 - Java 250
 - mechanism 761
- class reloading, automatic 250
- classes, debugging 254
- classpath
 - configuring 245
 - Java objects and 761
- client cookies 320
- Client scope 23, 56, 264, 316
- client state management
 - clustering 322
 - described 318
- Client variables 48
- client variables
 - and cflocation tag 327
 - built-in 326
 - caching 327
 - characteristics of 264, 316
 - configuring 323
 - creating 325
 - deleting 327
 - description 319
 - exporting from Registry 327
 - listing 326
 - setting options for 323
 - storage method 323
 - using 325, 331
- clustering, client state management 322
- Code Compatibility Analyzer
 - using 409
- code reuse 157
- code, protecting 336
- ColdFusion
 - ActionScript and 6
 - applications 262
 - CFScript 116
 - development tools 5
 - EJBs and 778
 - J2EE and 8
 - JSP and 760
 - login 354
 - logout 355
 - searching 522
 - servlets and 761
 - variables 34
 - XML and 688
- ColdFusion components 218
- ColdFusion MX
 - about 4
 - action pages, extension for 585
 - application services 6
 - architecture 8
 - CFML 4
 - CORBA type support 800
 - dynamic evaluation 77
 - error handling 287
 - error types 284
 - Flash connectivity 7
 - functions 5
 - integrating e-mail with 810
 - Java applets 760
 - Java objects and 761
 - JavaScript and 760
 - scripting environment 4
 - security 348
 - security features 348
 - support for LDAP 493
 - tags 4
 - using component metadata 680
- ColdFusion MX Administrator
 - creating collections 528
 - debugging settings 390
 - options 6
 - web services, consuming 741
- ColdFusion Studio
 - SQL Editor 430
- ColdFusion MX
 - features 10
- collections
 - creating 528
 - creating with cfcollection 535
 - indexing 530, 539, 540
 - optimizing 530
 - populating 530
 - repairing 530
 - searching 522
- column aliases 424
- columns 416
- COM
 - and WDDX 717
 - calling objects 788
 - character encodings 388
 - component ProgID and methods 790
 - connecting to objects 793
 - creating objects 793
 - description 786
 - error messages 796
 - getting started 790
 - input arguments 795
 - output arguments 795
 - requirements 790
 - setting properties 794

- threading 795
- using properties and methods 794
- viewing objects 791
- commas, in search expressions 562
- comments
 - CFScript 119
 - in CFML 31
- commits 418
- Common Object Request Broker Architecture. *See* CORBA
- compiler exception errors 287
- compiling, C++ CFX tags 256
- complex data type 35
- complex data types
 - web services 753
 - web services, publishing 753
- complex data types, returning 754
- complex variables 41
- Component Object Model. *See* COM
- component objects
 - invoking 788
 - overview 786
- components
 - applying design patterns 218
 - building 219
 - building secure components 234
 - calling 165
 - ColdFusion application security 235
 - component metadata 240
 - component packages 237
 - defining component methods 220
 - defining parameters 227
 - for web services 744, 748
 - inheritance 239
 - introductions 25
 - invoking component methods 222
 - naming components 238
 - passing parameters 228
 - programmtic security 237
 - recommendations for 165
 - requirements for web services 744
 - returned component method results 232
 - role-based security 236
 - saving component files 238
 - using 165
 - web server authentication 234
 - web services and 744
- connections, caching FTP 843
- constants 21
- constructors, using alternate 773
- continue, CFScript statement 128
- Cookie scope 22, 56
- cookie scope, catching errors 595
- cookie variables 48
- cookies
 - client 320
 - client state management 318
 - for security 356
 - for storing client variables 323
 - sending with cfhttp 837
- copying, server files 852
- CORBA
 - calling objects 800
 - case considerations 799
 - character encodings 388
 - description 786
 - double-byte characters 804
 - example 805
 - exception handling 804
 - getting started 797
 - interface 787
 - interface methods 799
 - naming services 798
 - parameter passing 799
- CreateObject CFML function 739
 - example 739
 - web services, consuming 736, 739
- CreateTimeSpan CFML function 279, 329, 466
- creating
 - action pages 586
 - action pages to insert data 448
 - action pages to update data 455
 - Application.cfm 268
 - arrays 90
 - basic charts 647
 - charts 646
 - client variables 325
 - collections 528, 535
 - data grids 622
 - dynamic form elements 599
 - error application pages 294
 - forms with cfform 608
 - graphs 646
 - HTML insert forms 446
 - insert action pages 448, 449
 - Java CFX tags 247
 - multidimensional arrays 91
 - queries from text files 835
 - queries of queries 462
 - structures 102
 - update action pages 455, 457
 - update forms 452
 - updateable grids 624
- criteria, multiple search 591
- curve charts 666
- custom exception types 286
- custom functions. *See* user-defined functions
- custom tag paths 199
- custom tags
 - ancestor 212
 - attributes 203
 - base 212
 - built-in variables 208
 - calling 162, 198, 199, 208
 - calling with cfimport 200
 - calling with cfmodule 199
 - CFX 244
 - children 212
 - data access example 215
 - data accessibility 213
 - data exchange 214
 - descendants 212
 - downloading 201
 - encoding 207
 - encrypting 207
 - example 204
 - execution modes 209
 - filename conflicts 201, 207
 - instance data 208
 - location of 199
 - managing 207
 - naming 199
 - nesting 212
 - parent 212
 - passing attributes 202, 203
 - passing data 212
 - path settings 199
 - recommendations for 163
 - restricting access to 201, 207
 - terminating execution 211

- types 18
 - using 162, 201
- D**
- data
 - accessibility with custom tags 213
 - charting 650
 - converting to JavaScript object 723
 - exchanging across application servers 718
 - exchanging with WDDX 718
 - graphing 650
 - passing between nested tags 213
 - selecting for retrieval 580
 - transferring from browser to server 723
 - data sharing, JSP pages 765
 - data sources
 - configuration problems 411
 - storing client variables in 323
 - troubleshooting 411
 - types of 434
 - data types 22
 - binary 24, 35
 - complex 24, 35
 - considerations 36
 - conversions 49
 - default conversion 774
 - in CFML 24
 - object 24, 35
 - simple 24, 35
 - validating 62
 - variables 35
 - database exceptions 303
 - database failures 285
 - Database Management System. *See* DBMS
 - databases
 - building queries 436
 - case sensitivity 422
 - character encodings 387
 - columns 416
 - commits 418
 - controlling access to 336
 - debug output 397
 - deleting data 459
 - deleting multiple records 460
 - deleting records 459
 - deleting rows 427
 - elements of 416
 - fields 416
 - forms for updating 446
 - insert form 448
 - inserting data 426
 - inserting data into 448
 - inserting records 446
 - introduction 415
 - locking 336
 - modifying 425
 - multiple tables 417
 - optimizing access 277
 - permissions 418
 - reading 422
 - record delete 459
 - record sets 423
 - records 416
 - retrieving data from 435
 - rollbacks 418
 - rows 416
 - SQL 420
 - stored procedures 277
 - stored procedures, debugging 398
 - tables 416
 - transactions 418
 - update form 452
 - updating 426, 446, 452
 - updating records 452
 - data-type conversions
 - ambiguous types 52
 - case sensitivity 50
 - cfoutput tag and 50
 - considerations 50
 - date-time values 53
 - date-time variables 51
 - default Java 774
 - example 54
 - issues in 51
 - Java 774
 - Java and 53
 - JavaCast and 53
 - numeric values 51
 - process 49
 - quotes 54
 - types 49
 - web services and 741
 - DateFormat CFML function 605
 - date-time format 39
 - date-time values
 - conversions 53
 - date-time variables 39
 - conversions 51
 - format 39
 - locale specific 40
 - representation of 40
 - DBMS 419
 - DCOM
 - description 786
 - getting started 790
 - See also* COM
 - deadlocks 341
 - debug information
 - for a query 402
 - outputting 253
 - debug pane 401
 - debugging
 - browser output 393
 - C++ CFX tags 257
 - ColdFusion MX Administrator and 390
 - configuring 390
 - custom pages and tags 293
 - Dreamweaver MX 389
 - enabling 390
 - Java CFX tags 253
 - Java classes for 254, 255
 - output 392
 - output format 390
 - programmatic control of 402
 - SQL queries 397
 - stored procedures 398
 - debugging output
 - cfquery tag 402
 - cfsetting tag 402
 - classic 390
 - database activity 397
 - dockable 390, 400
 - exceptions 399
 - execution time 395
 - format 390
 - general 394
 - in browsers 393
 - IP address for 392
 - IsDebugMode function 403
 - programmatic control 402
 - queries 397
 - sample 392
 - scopes 400

- SQL queries 397
- trace 399, 404
- debugging output, classic 390
- debugging output, dockable
 - about 390
 - application page 401
 - debug pane 401
 - format 400
- debugging output, general 394
- decision, or comparison,
 - operators 67
- declaring
 - arrays 90
 - structures and sequences 799
- default values, of variables 62
- DELETE SQL statement 421, 427, 459
- DeleteClientVariablesList CFML
 - function 327
- deleting
 - client variables 327
 - data 459
 - database records 459, 460
 - e-mail 826
 - server files 852
 - structures 107
- delimiters
 - search expression 563
 - text file 835
- descendant tags 212
- development environment
 - C++ 256
 - Java 245
- directories
 - indexing 522
 - information about 856
- directory operations 856, 858
- directory structure,
 - application 265, 266
- displaying
 - query results 438
 - query results, in tables 589
- distinguished name 493
- Distributed Component Object Model. *See* DCOM
- distributing CFX tags 258
- do while loop, CFScript 126
- DollarFormat function 605
- DOM node structure
 - XmlName 693
 - XmlType 693
 - XmlValue 693
- DOM node view
 - node types 690
 - XML 690
- Dreamweaver MX
 - BOM 380
 - debugging and 389
 - SQL editor 428
 - web services and 733
 - WSDL files and 733
- drop-down list boxes
 - See* list boxes
- dynamic evaluation
 - about 74
 - example 82
 - function arguments 77
 - functions 77
 - steps to 75
- dynamic expressions
 - about 74
 - string expressions 74
- dynamic variable names
 - about 74
 - arrays and 76
 - example 82
 - limitations 76
 - pound signs in 76
 - selecting 75
 - structures and 76
 - using 77
- dynamic variables, about 74
- E**
- editing, data in cfgrid 627
- EJB
 - calling 778
 - requirements for 778
 - using 778
- elements, of CFML 16
- e-mail
 - adding custom header 817
 - attaching files 817
 - attachments 824
 - character encodings 387
 - checking for spooled 818
 - customizing 815
 - deleting 826
 - error logging 818
 - for multiple recipients 815
 - form-based 813
 - handling POP 821
 - headers 821
 - indexing 522, 550
 - integrating ColdFusion 810
 - multiple recipients 814
 - query-based 813
 - receiving 819
 - searching 550
 - sending 811
 - undelivered 818
- embedding
 - Java applets 633, 635
 - URLs in a cftree 617
- enabling, session variables 329
- encoding custom tags 207
- error handling
 - ColdFusion 282
 - custom 264
 - in user-defined functions 191
 - strategies 291
- error messages
 - Administrator settings 289
 - COM 796
 - generating with cferror 293
- error pages
 - custom 293
 - example 296
 - rules for 294
 - specifying 293
 - variables 295
- errors
 - categories 283
 - causes 283
 - creating application pages 294
 - custom pages 293
 - input validation 296
 - logging 297
 - recovery 283
 - web services and 740
- EUC-KR 377
- evaluate CFML function 78
 - example 79
 - limitations 79
- evaluating
 - file upload results 850
 - strings in functions 188
- example
 - ancestor data access 215
 - Application.cfm 270, 363
 - caching a connection 843

- CFML Java exception
 - handling 777
 - CFScript 130
 - declaring CORBA structures 805
 - exception-throwing class 776
 - Java objects 771
 - JSP pages 766
 - JSP tags 763
 - LDAP security 369
 - locking CFX tags 346
 - regular expressions 640
 - request error page 296
 - setting default values 62
 - synchronizing file system
 - access 346
 - testing for variables 61
 - user authentication and
 - authorization 357
 - user security 360
 - user-defined functions 182
 - using Java objects 771, 772
 - using StructInsert 109
 - using structures 111
 - validating an e-mail address 643
 - validation error page 296
 - variable locking 343
 - web services, consuming 738
 - web services, publishing 747
- Excel spreadsheet
- from cfcontent tag 860
- exception handling
- cfcatch tag 299
 - cftry tag 299
 - CORBA objects 804
 - example 304, 310
 - in CFScript 129
 - in ColdFusion MX 299
 - Java 776
 - Java example 777
 - Java objects 776
 - nesting cftry tags 310
 - rules 300
 - tags 299
- exception types
- advanced 285
 - basic 285
 - custom 285, 286
 - Java 286
 - Java class 285
 - missing include file 285
- exceptions
- database 303
 - debugging output 399
 - expressions 303
 - handling 291
 - in user-defined functions 195
 - information returned 301
 - Java 776
 - locking 303
 - missing files 304
 - naming custom 308
 - types 285
- exclusive locks
- about 339
 - avoiding deadlocks 341
- execution time
- format 395
 - of ColdFusion pages 395
 - tree format 396
 - using 396
- explicit queries 558
- modifiers 558
 - operators 558
 - special characters 560
 - wildcards 559
- exporting client variable
- database 327
- expression exceptions 285, 303
- expressions 21
- CFScript 118
 - dynamic 74
 - operands 21
 - operator types 66
 - operators 21, 66
 - pound signs in 74
- extending CFML 244
- F**
- field searches 574
- fields 416
- fields, database 416
- file operations
- cfftp actions 844
 - using cffile 846
 - using cfftp 841
- file scope 208
- file types, supported for
- searching 523
- files
- appending 855
 - character encodings 387
 - controlling type uploaded 849
 - copying 852
 - deleting 852
 - downloading 858
 - locking access to 340, 346
 - moving 852
 - name conflicts 849
 - on server 846
 - reading 853
 - renaming 852
 - updating 336
 - uploading 846
 - writing 854, 855
- Find CFML function 134
- finding
- a structure key 104
 - component ProgID and
 - methods 790
 - with regular expressions 134
- Flash
- ColdFusion connectivity 7
 - Remoting service 7
- Flash Remoting 7
- ColdFusion Java objects 683
 - web services and 740
- Flash Remoting service
- arrays and structures 676
 - components 680
 - data types 675
 - Flash variable scope 675
 - handling errors 684
 - returning records in
 - increments 678
 - separating display code from
 - business logic 674
 - server-side ActionScript 682
 - Service Browser 681
 - using with ColdFusion
 - overview 674
- Flash scope 23, 56
- flow control
- tags 27
- for loop, CFScript 124
- for-in loop, CFScript 127
- form controls
- cfform 608
 - description 581
- form field validation errors 287
- form fields, required 603
- Form scope 22, 55

- form tag syntax 580
- form variables
 - considerations 588
 - in queries 586
 - naming 585
 - processing 585
 - referring to 585
 - scope of 585, 588
- formatting
 - data items 590
 - query results 590
- forms
 - about 580
 - action pages 585
 - character encodings 386
 - check boxes 599
 - considerations for 584
 - creating with cform 608
 - data encoding 385
 - deleting data 459
 - designing 584
 - drop-down list boxes 619
 - dynamically populating 597
 - HTML 580
 - inserting data 446
 - Java applets in 633
 - preserving data 609
 - requiring entries 588
 - slider bars 621
 - text entry boxes 620
 - tree controls 611
 - updating data 452
 - validating data in 603
- FROM SQL clause
 - description 421
- FTP 830
 - actions and attributes 844
 - caching connections 843
 - using cftp 841
- function local scope 23
- function variable, definition 175
- function, CFScript statement 174
- function-only variables 181
- Enterprise Java Beans
 - See* EJB
- functions
 - built in 19
 - calling 177
 - example custom 182
 - for arrays 98
 - introduction 19
 - JavaScript, for validation 642
 - structures 113
 - syntax 70
 - user defined 19
 - See also* ColdFusion functions, user-defined functions
- JavaServer Pages
 - See* JSP
- G**
 - generated content 210
 - Get method, cfhttp 830
 - GetAuthUser CFML function 353
 - GetClientVariablesList CFML function 326
 - GetLocale CFML functions 378
 - GetPageContext 762
 - globalization 373
 - applications 374
 - character encodings 377
 - character sets 375
 - currency functions 383
 - date functions 383
 - functions 382
 - input data 385
 - locales 375
 - numeric functions 383
 - request processing 379
 - string functions 382
 - tags 382
 - time functions 383
 - graphing
 - queries 650
 - See also* charts
 - grids
 - navigating 622
 - See also* cfgrid tag
- GROUP BY, SQL clause 421
- H**
 - handling
 - applet form variables 636
 - exceptions 299
 - failed validation 642
 - POP Mail 821
 - hidden fields 603
 - horizontal bar charts 660
 - HTML
 - using tables 589
 - using with cform 608
 - HTMLEditFormat CFML function 726, 822
 - HTTP 830
 - HTTP/URL problems 411
- I**
 - if-else, CFScript statements 122
 - IIF CFML function 80
 - implementing
 - C++ CFX tags 256
 - Java CFX tags 248
 - IN SQL operator 421
 - including ColdFusion pages 158
 - index, updating 530
 - indexing
 - cfldap query results 549
 - database query results 545
 - directories 522
 - e-mail 522, 550
 - external Verity collections 530
 - LDAP query results 549
 - query results 522
 - websites 522
 - indexing collections
 - about 530
 - with Administrator 540
 - with cfindex 539
 - infix notation, search string 562
 - inout parameters 740
 - input validation
 - cftree 614
 - with cform Controls 637
 - with JavaScript 642
 - INSERT SQL statement 421, 426
 - inserting data
 - description 446
 - with cfinsert 448
 - with cfquery 449
 - installation, support xxiv
 - instance data, custom tag 208
 - integer variables 37
 - international languages, search support 526
 - internationalization
 - applications 374
 - Internet
 - applications 2
 - ColdFusion and 2
 - dynamic applications 2
 - HTML and 2

- invoking
 - COM methods 794
 - component objects 788
 - methods in cobject 794
 - objects 769
 - IP address, debugging and 392
 - IsCustomFunction CFML function 188
 - IsDebugMode CFML function
 - debugging with 403
 - IsDefined CFML function 60, 104, 587, 603
 - IsStruct CFML function 103
 - IsUserInRole CFML function 353
 - IsXmlDoc CFML function 695
 - IsXmlElem CFML function 695
 - IsXMLRoot CFML function 695
- J**
- J2EE
 - about 760
 - benefits 9
 - ColdFusion and 8
 - GetPageContext 762
 - infrastructure 8
 - introduction 8
 - PageContext 762
 - J2EE application server 8
 - Java
 - about 760
 - alternate constructor 773
 - and ColdFusion data 774
 - and WDDX 717
 - class loading mechanism 761
 - class reloading 250
 - considerations 773
 - custom class 780
 - customizing and configuring 246
 - data-type conversions 774
 - data-type conversions with 53
 - development environment 245
 - EJB 778
 - exceptions 776
 - getting started 771
 - JavaCast function 776
 - objects 761
 - user-defined functions 777
 - Java applets 760
 - embedding 633, 635
 - form variables 636
 - overriding default values 635
 - registering 633
 - Java CFX tags
 - cfx.jar 247
 - class loading 250
 - debugging 253, 254
 - example 251
 - life cycle of 251
 - registering 247
 - writing 247
 - Java classes
 - custom 780
 - loading 761
 - Java exception classes 286
 - Java exceptions 286
 - handling 776
 - tags for 776
 - Java objects 761
 - calling 767
 - considerations 773
 - example 771
 - exception handling 776
 - invoking 769
 - JavaBeans and 770
 - methods, calling 770
 - nested 771
 - properties 769
 - using 769
 - JavaBeans, calling 770
 - JavaCast 53
 - JavaCast CFML function 776
 - JavaScript
 - ColdFusion MX and 760
 - differences from CFScript 120
 - in charts 670
 - validating with 642
 - joins
 - queries of queries 474
 - JSP pages
 - accessing 764
 - Application scope 765
 - calling from ColdFusion 767
 - example 766
 - Session scope 765
 - sharing data with 765
 - JSP tags
 - ColdFusion and 760
 - example 763
 - in ColdFusion applications 764
 - standard 762
 - tag libraries 762
 - using 762, 763
- K**
- keys, listing structure 104
- L**
- Latin-1 377
 - LD_LIBRARY_PATH
 - about 256
 - C++ CFX tags 256
 - LDAP
 - adding attributes 512
 - asymmetric directory structure 491
 - attribute values 514
 - attributes 492, 514
 - character encodings 388
 - deleting attributes 512
 - deleting entries 509
 - description of 490
 - directory attributes 512
 - directory DN 513
 - distinguished name 493
 - DN 513
 - entry 492
 - object classes 493
 - querying directories 496
 - referrals 519
 - schema 493
 - schema attribute type 494
 - scope 496
 - search filters 496
 - security 369
 - security and 369
 - symmetrical directory structure 490
 - updating directories 503, 510
 - LDAP query results
 - indexing 549
 - searching 549
 - LIKE SQL operator 421
 - linking from charts 667
 - list boxes
 - populating 619
 - populating dynamically 597
 - listing
 - Application variables 335
 - Client variables 326
 - ListQualify CFML function 600, 602

- ListSort CFML function 104
 - loading, Java CFX classes 250
 - locales
 - introduction 375
 - language 378
 - regional variation 378
 - setting 378
 - variant 378
 - localization
 - applications 374
 - dates 40
 - lock management 341
 - locking
 - avoiding deadlocks 341
 - CFX tags 346
 - exceptions 303
 - file access 346
 - granularity 341
 - scopes 339
 - with cflock 336
 - write-once variables 338
 - locking exceptions 285
 - locks
 - controlling time-outs 340
 - exclusive 339
 - naming 340
 - read-only 339
 - scopes and names 339
 - types 339
 - log files
 - example 297
 - using 297
 - logging errors 297
 - login
 - applicationToken 355
 - browser support for 368
 - internet domains 355
 - structure 356
 - tags 354
 - tokens 354
 - logout, performing 355
 - looping through structures 107
- M**
- Macromedia ColdFusion MX. *See* ColdFusion MX
 - Macromedia Dreamweaver MX. *See* Dreamweaver MX
 - Macromedia Flash Remoting. *See* Flash Remoting
 - Macromedia HomeSite+, SQL editor 430
 - mail servers, and ColdFusion MX 810
 - managing
 - client state 318
 - client state, in clusters 322
 - custom tags 207
 - mapping, application framework 265
 - matched subexpressions
 - len array 147
 - minimal matching 149
 - pos array 147
 - result arrays 147
 - matches, pattern 640
 - method attribute, cfhttp tag 830, 837
 - migration
 - Code Compatibility Analyzer 409
 - MIME type 858
 - missing files, exceptions 304
 - missing template errors 287
 - modifiers, searching 572
 - MonthAsString CFML function 95
 - moving, data across the web 717
 - multicharacter regular expressions
 - for searching 137
 - for validation 639
 - multiple selection lists 601
- N**
- naming
 - applications 268
 - variables 203
 - naming conventions, for custom exceptions 308
 - navigating grids 622
 - nested pound signs in expressions 73
 - nesting
 - cflock tags 341
 - cfloops for arrays 96
 - custom tags 212
 - object calls 771
 - tags, using Request scope 213
 - NOT SQL operator 421
 - numeric variables 36
 - converting 51
- O**
- object data type 35
 - object exceptions 285
 - objects
 - calling methods 770, 788
 - calling nested 771, 789
 - COM 786
 - CORBA 786
 - DCOM 786
 - invoking 769
 - Java 761, 769
 - nesting object calls 789
 - query 249
 - Request 248
 - Response 248
 - using properties 769, 788
 - OLE/COM Object Viewer 791
 - OnRequestEnd.cfm 265
 - opening, SQL Builder 430
 - operands 21
 - operators 21
 - alternative notation 68
 - arithmetic 66
 - Boolean 67
 - comparison 67
 - concept 564
 - decision, or comparison 67
 - evidence 568
 - precedence 69
 - proximity 569
 - relational 565
 - score 571
 - search 563
 - SQL 421
 - string operators 69
 - types 66
 - optimizing
 - applications 272
 - caching 272
 - database access 277
 - optional arguments
 - about 175, 179
 - in functions 70
 - OR SQL operator 421
 - ORDER BY SQL clause 421, 424
 - out parameters 740
 - outputting
 - debug information 253
 - query data 438
 - overriding default Java applet values 635

- P**
- page encoding
 - BOM 380
 - default 380
 - determining 380
 - setting 380
 - page execution time 395
 - tree format 396
 - page settings 269
 - PageContext 762
 - pages
 - cache flushing 273
 - caching 272
 - parent tags 212
 - passing
 - arguments 180
 - arrays to user-defined functions 189
 - custom tag attributes 202, 203
 - custom tag data 212
 - queries to user-defined functions 187
 - passthrough attribute 608
 - paths
 - custom tags 199
 - performing a query on a query 465
 - Perl
 - regular expression compliance 154
 - WDDX and 717
 - persistent scope variables 316
 - persistent variables
 - in clustered system 317
 - scopes 317
 - using 317
 - pie charts
 - example 662
 - setting appearance 662
 - populating
 - arrays from queries 97
 - arrays with ArraySet 95
 - arrays with cfloop 95
 - arrays with nested loops 96
 - Post method, cfhttp 830, 837
 - pound signs
 - in cfoutput tags 72
 - in general expressions 74
 - inside strings 72
 - inside tag attributes 71
 - nested 73
 - using 71
 - precedence rules, search 563
 - prefix notation, search strings 562
 - preservedata cfform attribute 609
 - problems, troubleshooting 410
 - processing
 - Application.cfm 265
 - form variables on action pages 585
 - Java CFX requests 248
 - OnRequestEnd.cfm 265
 - protecting data 336
 - proximity operators 569
 - punctuation, searching 560
 - Python, WDDX and 717
- Q**
- queries 650
 - as function parameters 187
 - as variables 43
 - building 420, 436
 - charting 650
 - converting to XML 709
 - creating from text files 835
 - graphing 650
 - grouping output 612
 - guidelines for outputting 439
 - outputting 438
 - referencing 44
 - scopes 44
 - syntax 435
 - troubleshooting 411
 - using form variables 586
 - web services, consuming 742
 - web services, publishing 756, 757
 - XML and 708
 - queries of queries
 - aggregate functions 480
 - aliases 475, 482
 - benefits 465
 - BNF syntax 486
 - case sensitivity 479
 - cfdump tag and 470
 - combining record sets 472
 - conditional operators 477
 - displaying record sets 468
 - escaping reserved words 483
 - escaping wildcards 480
 - evaluation order 476
 - example 466
 - joins 474
 - non-SQL record sets and 470
 - null support 483
 - ORDER BY clause 481
 - performing 465
 - reserved words 483
 - syntax 486
 - unions 474
 - user guide 474
 - using 462
 - Query CFX object 249
 - query columns 44
 - query functions 462
 - Query objects 249
 - query objects 43, 462
 - query properties, guidelines for 442
 - query results
 - about 441
 - cfpop 550
 - columns in 441
 - current row 441
 - displaying 438
 - indexing 522
 - LDAP 549
 - no records 593
 - records returned 441
 - returning 593
 - returning incrementally 594
 - variables 441
 - query variables 43
 - querying, LDAP directories 496
 - queryNew() CFML function 463
 - quotes
 - for IsDefined CFML function 60
 - using 60, 436
- R**
- Rand CFML function 596
 - RandRange CFML function 596
 - RDN (Relative Distinguished Names) 493
 - reading, a text file 853
 - read-only locks 339
 - real number variables 37
 - receiving e-mail 819
 - record sets 423
 - combining 472
 - creating 462
 - displaying 468
 - example 463
 - queries of queries 462

- searching 545
 - with functions 463
- records 416
 - definition 416
- recoverable expressions 284
- recursion
 - with user-defined functions 190
- referencing array elements 90
- referrals, LDAP 519
- REFind CFML function 147
- REFindNoCase CFML
 - function 147
- registering
 - CFX tags 257
 - COM objects 790
 - CORBA objects 798
 - Java applets 633
- regular expressions
 - backreferences 144, 640
 - basic syntax 135
 - case sensitivity 138
 - character classes 143
 - character sets 136
 - common uses 152
 - escape sequences 141
 - examples 152, 640
 - for form validation 637
 - for searching and replacing
 - text 133
 - hyphens in 137
 - minimal matching 149
 - partial matches 640
 - Perl compliance 154
 - repeating characters 137
 - replacing with 134
 - returning matched
 - subexpressions 147
 - single-character 136, 638
 - special characters 136, 138
 - technologies 154
- relational operators 565
- remote servers 830
- renaming server files 852
- Replace CFML function 134
- replacing using regular
 - expressions 134
- Request object 248, 249
- Request scope
 - about 22, 56, 213
 - user-defined functions and 186

- requests
 - globalization and 379
 - processing 379
- requiring form entries 588
- reserved words
 - in CFML 32
 - list of 32
- reserved words, CFScript 120
- reset buttons 581
- resolving
 - custom tag file conflicts 201, 207
 - filename conflicts 849
- resource security
 - resources 349
 - using 349
- resources, regular expressions 641
- Response object 248, 249
- results, returning incrementally 594
- retrieving
 - binary files 830
 - e-mail attachments 824
 - e-mail headers 821
 - e-mail messages 823
 - files 841
 - query data 435
 - text 830
- return CFScript statement 175
- returning
 - file information 856
 - query results 593
 - results incrementally 594
 - subexpressions 147
- reusing code
 - cfinclude 198
 - custom tags 198
 - method comparison 166
 - methods 157
 - options 158
 - techniques 158
- roles
 - checking 357
 - defined 351
 - setting 357
 - source for 351
 - using 351
 - web services and 751
- rollbacks 418
- rows in tables 416

S

- sample CFX tags
 - C++ 256
 - Java 245
- sandbox security, resource security
 - and 349
- saving
 - binary files 833
 - web pages 832
- schema
 - LDAP directory 515
- scopes
 - about 55
 - and user-defined functions 180
 - Application 56, 264, 316, 333
 - Arguments 56
 - as structures 59
 - Attributes 55
 - Caller 55
 - CFX tags 59
 - CGI 56
 - Client 56, 264, 316, 319, 323
 - Cookie 56
 - debug output 400
 - evaluating 59
 - File 208
 - Flash 56
 - Form 55, 585
 - function local 57
 - LDAP 496
 - locking 339
 - managing locking of 342
 - of Form variables 588
 - persistent variables 316
 - Request 56, 213
 - Server 56, 264, 316, 335
 - Session 56, 264, 316, 319, 328
 - This 57
 - ThisTag 55
 - types 55
 - URL 55
 - using 59
 - Variables 55
 - variables 44, 57
- score search operators 571
- search criteria, multiple 591
- search expressions
 - case sensitivity 562
 - commas in 562
 - composing 562
 - delimiters 563

- operators 563
- with wildcards 559
- search, character encodings 388
- searching
 - case sensitivity 562
 - cfsearch tag 542
 - collections 522
 - collections, creating 528
 - creating index summaries 544
 - database records 545
 - external Verity collections 530
 - fields 573
 - file types 523
 - for special characters 560
 - full-text 522
 - international languages 526
 - LDAP query results 549
 - modifiers 572
 - numeric values 599, 601
 - operators 563
 - performing 542
 - prefix and infix notation 562
 - punctuation 560
 - query results 549
 - record sets 545
 - refining 573
 - results of 542
 - search expressions 562
 - special characters 560
 - string values 600, 601
 - wildcards for 559
 - zones 573
- searching e-mail 550
- securing, custom tags 201, 207
- security
 - application 264
 - application security 347
 - basic authentication 353
 - cookies and 356
 - functions 353
 - IsAuthenticated CFML function 357
 - LDAP and 369
 - login 354
 - logout 355
 - resource types 348
 - resources 349
 - roles 351
 - sandbox security 349
 - scenarios 357
 - tags 353
 - types 348
 - user 351
 - web servers and 353, 750
 - web services 749, 750
- security exceptions 285
- SELECT SQL statement 421, 422
- selection lists, multiple 601
- sending
 - e-mail 811
 - e-mail to multiple recipients 814
 - form-based e-mail 813
 - mail as HTML 818
 - query-based e-mail 813
- Server scope 23, 56, 264, 316
- server variables
 - about 264, 316
 - built-in 335
 - using 335
- servers
 - remote 830, 841
 - retrieving files from 830
 - uploading files 846
- server-side ActionScript 6
- servlets
 - ColdFusion and 761
 - in ColdFusion applications 764
- Session scope
 - about 23, 56, 264, 316
 - JSP pages 765
- Session variables
 - about 23, 56, 264, 316, 319, 330
 - built-in 330
 - enabling 329
 - using 328
- session, definition of 328
- setEncoding CFML function 386
- SetLocale CFML function 378
- setting
 - application defaults 269
 - bar chart characteristics 660
 - Client variable options 323
 - file and directory attributes 850
 - pie chart characteristics 662
- setting up
 - C++ development environment 256
 - Java development environment 245
- settings, application-level 263
- SetVariable CFML functions 80
- Shift-JIS 377
- SHLIB_PATH
 - about 256
 - C++ CFX tags 256
- shorthand notation, for Boolean operators 68
- simple queries 555
- simple queries, stemming 555
- simple variables 35
- single quotes, in SQL 436, 602
- single-character regular expressions 136, 638
- slider bar controls 621
- SMTP 811
- SOAP
 - about web services and 731
 - defined 731
- special characters 560, 638
 - entering 31
 - list 31
- specifying
 - Client variable storage 325
 - tree items in URLs 618
- SQL
 - AVG function 652
 - case sensitivity 422
 - column aliases 424
 - debugging output 397
 - DELETE statement 427, 459
 - Dreamweaver MX for 428
 - example 420
 - filtering 423
 - generating dynamically 585
 - guidelines 422
 - INSERT statement 426, 449
 - introduction 415, 420
 - nonstandard 422
 - operators 421
 - ORDER BY clause 424
 - ordering results 424
 - query editors 428
 - record sets 423
 - results 423
 - SELECT statement 422
 - single quotes in 436, 602
 - sorting 424
 - statement clauses 421
 - statements 421
 - SUM function 663

- syntax 421
 - text literals in 436
 - UPDATE statement 426, 452
 - use in cfquery 435
 - WHERE clause 423, 585
 - writing 420
 - standard variables. *See* built-in variables
 - statement clauses, SQL 421
 - statements
 - CFScript 118
 - SQL 421
 - status output
 - with user-defined functions 192
 - stemming
 - preventing 557
 - simple queries 555
 - stored procedures 277
 - string operators 69
 - string variables 37
 - strings
 - a variables 37
 - empty 37
 - escaping 37
 - evaluating in functions 188
 - quoting 37
 - storing complex data in 726
 - StructClear CFML function 107
 - StructCount CFML function 103
 - StructDelete CFML function 107
 - StructIsEmpty CFML function 103
 - StructKeyArray CFML
 - function 104
 - StructKeyExists CFML
 - function 104
 - StructKeyList CFML function 104
 - StructNew CFML function 102
 - structures
 - about 99
 - adding data to 102
 - as variables 42
 - copying 105
 - creating 102
 - custom tag 203
 - deleting 107
 - example 109
 - finding keys 104
 - functions 113
 - getting information on 103
 - in dynamic expressions 76
 - listing keys in 104
 - looping through 107
 - notation for 99
 - passing tag arguments 205
 - referencing 44
 - scopes 44
 - scopes and 59
 - sorting keys 104
 - updating 102
 - web services, consuming 753
 - web services, publishing 756
 - sub tags, definition 212
 - submit buttons 581
 - SUM SQL function 663
 - summaries, search 544
 - switch-case, CFScript 123
 - syntax, errors in CFML 410
- T**
- tables 416
 - displaying queries 589
 - using HTML 589
 - tag libraries 762
 - tags 694
 - built in 17
 - custom 18
 - syntax 17
 - TCP network directory services 495
 - template errors 285
 - testing, a variable's existence 587
 - text control 581
 - text files
 - column headings 835
 - creating queries from 835
 - delimiters 835
 - This scope 23, 57
 - ThisTag scope 22, 55
 - throwOnTimeout, cflock
 - attribute 340
 - time zone processing, WDDX 721
 - time-out attribute, cflock 340
 - ToString CFML function 695
 - tracing 404
 - cftrace tag 404
 - considerations for 406
 - enabling 390
 - format 405
 - messages 405
 - options 404
 - output 399, 404
 - transactions 418
 - transferring data, from browser to server 723
 - cfree tagcfree tag
 - See also* tree controls
 - tree controls, structuring 614
 - troubleshooting 410
 - CFML syntax 410
 - common problems 410
 - data sources 411
 - HTTP 411
- U**
- UCS-2 377
 - UDDI
 - about 731
 - defined 732
 - UDF. *See* user-defined functions
 - Unicode
 - and ColdFusion 377
 - character encoding 377
 - unions, queries of queries 474
 - Universal Description, Discovery and Integration 732
 - UNIX, permissions 850
 - UPDATE SQL statement 426
 - updating
 - a database with cfgridupdate 629
 - a database with cfquery 630
 - data using forms 452
 - files 336
 - values in structures 102
 - uploading files 846
 - uploads, controlling file type 849
 - URL scope 22, 55
 - URLEncodedFormat CFML
 - function 411
 - URLs
 - character sets 385
 - encoding 385
 - user authentication
 - example 363
 - IsAuthenticated CFML
 - function 357
 - user edits, returning 626
 - user roles 351
 - user security 351
 - application based 362
 - basic authentication 360
 - example 360
 - flow of control 351

- implementing 360
 - overview 351
 - user-defined functions
 - argument naming 174
 - arguments 180, 186
 - Arguments scope and 171, 172
 - array arguments 189
 - calling 161, 169, 177
 - CFML tags in 179
 - CFSyntax syntax 174
 - creating 169
 - creating with tags 170
 - creation rules 170
 - defining 174
 - described 168
 - effective use of 184
 - error handling 191
 - evaluating strings 188
 - example 177, 182
 - exception handling 195
 - function-only variables 174
 - generating exceptions 196
 - identifying 188
 - in Application.cfm 184
 - Java and 777
 - passing arrays 189
 - queries as arguments 187
 - recommendations for 161
 - recursion 190
 - Request scope and 186
 - status output 192
 - using with queries 184
 - variables 180
 - users, keeping track of 318
 - UTF-8 377
- V**
- validating
 - data types 62
 - form attributes 637
 - form field data types 603
 - form input 614
 - JavaScript functions 642
 - user input 603
 - using regular expressions 637
 - validation, error handling 642
 - var, CFSyntax statement 175
 - variable names, periods in 45, 47
 - variable naming 34
 - variable scopes 22
 - Application 23
 - Arguments 23
 - Attributes 22
 - Caller 22
 - CGI 22
 - Client 23
 - Cookie 22
 - Flash 23
 - Form 22
 - function local scope 23
 - Request 22
 - Server 23
 - Sessions 23
 - This 23
 - ThisTag 22
 - URL 22
 - Variables 22
 - variables
 - Application 333
 - Application scope 264, 316, 333
 - array 41
 - Base64 40
 - binary 35, 40
 - Boolean 38
 - caching 327
 - CFSyntax 118
 - cfset tag and 34
 - client 48
 - Client scope 264, 316, 319
 - complex 35, 41
 - configuring Client 323
 - cookie 48
 - creating 34
 - data types 22
 - datatypes 24
 - date-time 39
 - default 61, 62, 269
 - dynamic naming 74
 - ensuring existence of 60
 - evaluating 49
 - Form 585
 - formatting 590
 - forms 580
 - getting 46
 - in user-defined functions 180
 - integer 37
 - kinds of 22
 - locking example 343
 - naming 203
 - naming rules 34
 - numeric 36
 - objects 35
 - passing 830
 - persistent 316
 - processing 580
 - queries 43
 - real numbers 37
 - Request scope 213
 - scopes 22, 44, 57
 - scopes for custom pages 213
 - sending 837
 - Server 335
 - Server scope 264, 316
 - Session scope 264, 316, 319, 328, 330
 - setting 46
 - setting default values 62
 - shared 264
 - simple 35
 - string 37
 - structures 42
 - testing for existence 60, 61, 587
 - validating 603
 - See also* built-in variables
 - Variables scope 22, 55
 - verbs, SQL 421
 - Verity
 - case sensitivity 562
 - explicit queries 558
 - query types 554
 - refining search 573
 - searching with 523
 - simple queries 555
 - wizard 530
 - zone filter 573
 - Verity Search engine exception 286
 - Verity Wizard 530
- W**
- WDDX
 - character encodings 388
 - components 717
 - converting CFML to
 - JavaScript 723
 - exchanging data 718
 - operation of 718
 - purpose of 717
 - storing data in strings 726
 - time zone processing 721
 - transferring data 723

- web
 - accessing with cfhttp 717, 830
 - application framework 318
- web application servers
 - request handling 3
 - tasks 3
 - web servers and 3
- web pages
 - dynamic 434
 - saving 832
 - static 434
- web servers
 - Apache 2
 - authorization 353
 - basic authorization 353
 - IIS 2
 - overview 2
 - security 353
- web services
 - accessing 730
 - basic authentication and 750
 - CFScript and 739
 - ColdFusion MX
 - Administrator 741
 - complex data types 753
 - components for 744
 - concepts 731
 - consuming 730, 736
 - Dreamweaver MX and 733
 - error handling 740
 - Flash Remoting and 740
 - introduction 730
 - parameter passing 736
 - publishing 730, 744
 - return values 737
 - securing 749
 - SOAP and 731
 - type conversions 741
 - UDDI and 731
 - WSDL file
- Web Services Description Language
 - file 730
 - See also* WSDL
- web services, consuming 730
 - cfinvoke tag 736, 737
 - CFScript for 739
 - ColdFusion MX 742
 - ColdFusion MX
 - Administrator 741
 - complex data types 753
 - CreateObject function 739
 - error handling 740
 - example 738
 - inout parameters 740
 - methods for 736
 - out parameters 740
 - parameter passing 736
 - queries 742
 - return values 737
 - structures 742, 753
 - type conversions 741
- web services, publishing 730, 744
 - best practices for 752
 - complex data types 756
 - components and 744
 - components as data types 748
 - data types for 744
 - example 747
 - queries 756
 - requirements 744
 - securing 749
 - structures 756
 - WSDL files 745
- web services, security
 - about 750
 - example 750
 - in ColdFusion 750
 - programmatic 751
 - roles for 751
 - using web servers 750
- websites, indexing 522
- WHERE SQL clause 423
 - comparing with 585
 - description 421
- while loop, CFScript 126
- wildcards, in searches 559
- Windows file attributes 850
- Windows NT, debugging C++ CFX
 - tags 257
- writing SQL statements 431
- WSDL files
 - components 735
 - creating 733
 - defined
 - described 732
 - reading 734
 - viewing in Dreamweaver
 - MX 733
 - web services, publishing 745
- X**
- XML 694
 - basic document view 690
 - ColdFusion and 688
 - converting to query 708
 - DOM node view 690
 - elements 703
 - example 712
 - functions 694
 - queries and 708
 - using 688
 - XML document object 689
- XML document object
 - assigning data to 697
 - basic view 690
 - changing 703
 - child elements 701
 - converting to query 708
 - creating 698, 699
 - definition 689
 - deleting 702
 - DOM node view 690
 - example 712
 - exporting 699
 - extracting data with XPath 711
 - modifying 700
 - referencing summary 702
 - referencing syntax 697
 - saving 698
 - structure 691, 692
 - syntax for referencing 696
 - transforming, XSLT 710
 - using 696
 - XmlComment 692
 - XmlDocType 692
 - XmlRoot 692
 - XPath 711
 - XSLT 710
- XML elements
 - adding 704, 705
 - attributes 707
 - child elements 703
 - children of 701
 - copying 706
 - counting 703
 - deleting 706
 - deleting multiple elements 707
 - finding 703
 - properties 707
 - properties, modifying 707
 - replacing 708

- XmlAttributes 692
- XmlChildren 692
- XmlComment 692
- XmlName 692
- XmlNodes 692
- XmlNsPrefix 692
- XmlNsURI 692
- XmlParent 692
- XmlText 692
- XmlAttributes 692
- XmlChildPos CFML function 694
- XmlChildren 692
- XmlComment 692
- XmlDocType 692
- XmlElemNew CFML function 694
- XmlName 692
- XmlNew CFML function 694
- XmlNodes 692
- XmlNsPrefix 692
- XmlNsURI 692
- XmlParent 692
- XmlParse CFML function 694
- XmlRoot 692
- XMLSearch CFML function 695
- XmlText 692
- XMLTransform CFML
function 695
- XmlType 693
- XmlValue 693
- XPath
 - extracting XML data 711
 - XSL transformation with 711
- XSLT
 - example 710
 - transforming XML
documents 710

Z

- zone searches 573

