Advanced ColdFusion Development

ColdFusion 4.0 for Windows® NT, Windows 95/98, and Solaris

Allaire Corporation

Copyright Notice

Part Number: AA-ADDEV-RK

Contents

Chapter 1: Advanced ColdFusion Development 1

About this Manual 2
Developer Resources 2
Developing Applications in ColdFusion Studio 3
About ColdFusion Documentation 3
    Documentation distribution 4
    Reading online documentation 4
Contacting Allaire 5

Chapter 2: Functions and Expressions 7

About ColdFusion Expressions 8
    What’s in an expression? 8
    Expression examples 8
Creating Expressions in ColdFusion Studio 9
The Structure of Expressions 9
    Numbers 9
    Strings 10
    Boolean values 10
    Date-and-time values 11
    Lists 12
    Structures 13
    Arrays 13
    Queries 14
    COM objects 14
    Variables 15
Operators 16
    Arithmetic operators 16
    String operators 17
    Decision, or comparison, operators 18
    Boolean operators 19
    Operator precedence 20
Functions 21
    Function types 21
    Function usage 22

Optional arguments in functions 23
Functions that return a Boolean 23
Using Pound Signs 24
Pound signs inside CFOUTPUT tags 24
Pound signs inside strings 25
Pound signs inside tag attribute values 26
Nested pound signs 26
Pound signs in general expressions 27
Typeless Expression Evaluation 27
Operation-driven evaluation 27
Conversion between types 28
Examples of Typeless Expression Evaluation 29
Debugging and Troubleshooting Expressions 30
Setting watches in the debugger 30
Testing expressions using CFSET and CFOUTPUT 31

Chapter 3: Dynamic Expression Evaluation 33

About Dynamic Expression Evaluation 34
String expressions 34
Evaluating String Expressions 35
Pound Signs Inside String Expressions 36
Setting Variables Dynamically 37
Using the SetVariable and Evaluate functions together 37

Chapter 4: Regular Expressions 39

Regular Expressions 40
About regular expressions 40
Single-Character Regular Expressions 40
Character Classes 41
Multi-Character Regular Expressions 42
Using Backreferences 42
Returning Matched Sub-Expressions 43
Anchoring a Regular Expression to a String 44
Regular expression examples 44
Regular expressions in CFML 45

Chapter 5: Working with Arrays 47

About Arrays 48
Creating an array 48
Array terms 48
Dynamic arrays 49
Multidimensional Arrays 50
Basic Array Techniques 50
Adding elements to an array 51
Shifting indexes in a dynamic array 51
Referencing Elements in Dynamic Arrays 51

Additional referencing methods 52
Calculating an array index 52
Populating Arrays with Data 52
Populating an array with ArraySet 53
Populating an array with CFLOOP 53
Using Nested Loops for 2D and 3D Arrays 53
Nesting CFLOOPs for a 2D array 54
Nesting CFLOOPs for a 3D array 54
Populating an Array from a Query 54
Array Functions 56

Chapter 6: Working with Structures 57

About Structures 58
Structure notation 58
Creating and Using Structures 58
Creating structures 59
Adding data to structures 59
Finding information in Structures 59
Getting information about structures 60
Copying structures 60
Deleting structures 61
Structure Example 61
Using Structures as Associative Arrays 63
Looping through structures 63
Structure Functions 64

Chapter 7: Exchanging Data via XML 67

An Overview of Distributed Data for the Web 68
WDDX Components 68
Working With Application-Level Data 69
Data Exchange Across Application Servers 69
How WDDX Works 69
Converting CFML Data to a JavaScript Object 70
Transferring Data From Browser to Server 72

Chapter 8: Using CFML Scripting 75

About CFScript 76
CFScript example 76
Supported statements 76
The CFScript Language 77
Statements 77
Expressions 80
Variables 80
Comments 80
Differences from JavaScript 80
Reserved words 81

Interaction of CFScript with CFML 81

Chapter 9: Structured Exception Handling 83

Overview of Exception Handling in ColdFusion 84
Types of recoverable exceptions supported 84
Exception-Handling Strategies 85
Exception Handling Example 86
CFTHROW syntax 87
CFTRY syntax 88
CFCATCH syntax 88
Order of evaluation 88
Exception Information in CFCATCH 88
Database exceptions 89
Locking exceptions 89
MissingInclude exceptions 89

Chapter 10: Accessing the Registry 91

Overview of Registry Access in ColdFusion 92
Getting Registry Values 92
Setting Registry Values 93
Deleting Registry Values 94

Chapter 11: Building ColdFusion Extensions 95

About ColdFusion Extensions 96
Building ColdFusion Extensions in CFML 96
Allaire Tag Gallery 96
Allaire Alive 96
Custom Tag Editors 97
Installing Custom Tags 97
Local tags 97
Shared tags 97
Writing Custom Tags 97
Naming Custom Tags 98
Tag scope 98
Defining attributes 98
Example Tags 98
Utility tags 98
Function tags 99
User interface tags 99
CFML 4.0 Custom Tag Enhancements 102
Main features of CFML 4.0 custom tags 102
Tag nesting 102
Associating sub-tags with the base tag 102
Tag instance data 103
Pattern of execution 103
Modes of execution 103

Specifying execution modes 103
CFEXIT 104
Access to generated content 105
Inter-tag data exchange 105
High-level data exchange 108
Managing Custom Tags 109
Resolving file name conflicts 109
Securing Custom Tags 110
Encrypting Custom Tags 110
Building Extensions in C++ 111
Sample C++ tags 112
Implementing CFX Tags 112
Debugging CFX tags 113
Registering CFX tags 113
Managing CFX Tags 114
Adding a CFX tag 114
Changing CFX tag settings 115
Deleting a CFX tag 115
Using COM and CORBA Objects 115
About CORBA 115
Getting Started with COM Components 116
Getting set up: COM 116
Register the object 117
Find the component ProgID and methods 117
Using the OLE/COM Object Viewer 117
Creating and Using COM Objects 118
Two ways to create objects with CFOBJECT 119
Setting Properties and Invoking Methods 120
Getting Started with CORBA Objects 121
Using CFOBJECT to create a CORBA object 121

Chapter 12: The ColdFusion Extension API 123

The ColdFusion Extension (CFX) API 124
The CCFXException Class 125
The CCFXQuery Class 125
Class members 125
CCFXQuery::AddRow 125
CCFXQuery::GetColumns 126
CCFXQuery::GetData 127
CCFXQuery::GetName 127
CCFXQuery::GetRowCount 128
CCFXQuery::SetData 128
CCFXQuery::SetQueryString 129
CCFXQuery::SetTotalTime 129
The CCFXRequest Class 130
Overview 130
Class Members 130
CCFXRequest::AddQuery 131

CCFXRequest::AttributeExists 132
CCFXRequest::CreateStringSet 132
CCFXRequest::Debug 132
CCFXRequest::GetAttribute 133
CCFXRequest::GetAttributeList 133
CCFXRequest::GetCustomData 134
CCFXRequest::GetQuery 134
CCFXRequest::GetSetting 135
CCFXRequest::ReThrowException 135
CCFXRequest::SetCustomData 136
CCFXRequest::SetVariable 136
CCFXRequest::ThrowException 137
CCFXRequest::Write 138
CCFXRequest::WriteDebug 138
The CCFXStringSet Class 139
    Overview 139
    Class members 139
CCFXStringSet::AddString 139
CCFXStringSet::GetCount 139
CCFXStringSet::GetIndexForString 140
CCFXStringSet::GetString 140

Chapter 13: Connecting to LDAP Directories 143

ColdFusion Support for LDAP 144
Directory Structures 144
    Viewing directory schema 145
    LDAP attributes 146
Key Terms 146
Operations 147
Search Filters 148
Examples 148

Chapter 14: Application Security 153

ColdFusion Security Features 154
Remote Development Services (RDS) Security 154
    RDS and Basic security 154
    Configuring RDS security 155
Overview of User Security 155
    Implementation summary 156
Using Advanced Security in Application Pages 156
    CFAUTHENTICATE syntax 157
    Authentication and Authorization functions 158
    Catching security exceptions 159
Example of User Authentication and Authorization 160
    Authenticating users in Application.cfm 160
    Checking for Authentication and Authorization 162

CHAPTER 1

Advanced ColdFusion Development

The programming techniques and server features presented here are "advanced" in the sense that they give ColdFusion developers tools to build and deploy Web applications at the highest levels of current technology. The skills you need to work with these tools are extensions of the skills you have already mastered in ColdFusion and in other development environments.

The strengths of ColdFusion’s tag-based language and open architecture are evident in both the depth of its programming capabilities and in the rapid adoption of new technologies to meet the demands of today’s Web solutions.

Allaire’s goal of encapsulating complexity for developers applies equally to data manipulation, language extensibility, supporting technologies, and server options.

This preface gives a brief overview of the topics covered, a description of the product documentation set, and a listing of additional ColdFusion resources.

Contents

- About this Manual.. 2
- Developer Resources... 2
- Developing Applications in ColdFusion Studio.. 3
- About ColdFusion Documentation 3
- Contacting Allaire... 5

About this Manual

Advanced ColdFusion Development describes a wide range of features, including:

- Data manipulation using expressions and CFML functions
- Exchanging structured data via XML
- CFML scripting
- Custom tags
- CFML Extensions
- Incorporating object technologies via COM/DCOM and CORBA
- Connecting to LDAP directories
- Regular expressions
- Using arrays and structures
- Application security

Developer Resources

Allaire Corporation is committed to setting the standard for customer support in developer education, technical support, and professional services. Our Web site is designed to give you quick access to the entire range of online resources.

Allaire Developer Services	
Resource	**Description**
Allaire Web site www.allaire.com	General information about Allaire products and services.
Technical Support www.allaire.com/support	Allaire offers a wide range of professional support programs. This page explains all of the available options.
Professional Education www.allaire.com/education	Information about classes, on-site training, and online courses offered by Allaire.
Developer Community www.allaire.com/developer	All of the resources you need to stay on the cutting edge of ColdFusion development, including online discussion groups, Knowledge Base, Component Exchange, Resource Library, technical papers and more.
Allaire Alliance www.allaire.com/partners	The growing network of solution providers, application developers, resellers, and hosting services creating solutions with ColdFusion.

Developing Applications in ColdFusion Studio

ColdFusion Studio is a special version of HomeSite, Allaire's award-winning HTML editor. HomeSite's strengths in Web page creation have been enhanced with powerful tools specifically designed for ColdFusion development.

All of the components of dynamic page creation and site management are accessible from Studio.

- View your data sources.
- Quickly build SQL statements to insert in CFQUERY.
- Access the complete HTML and CFML tag sets from the Tag Chooser.
- Edit code from tag-specific editors or from the Tag Inspector.
- Render pages with internal or external browsers and visually edit page elements in Design view.
- Create projects to group your application pages and support files for easy maintenance and uploading.
- Quickly make global changes to files using extended search and replace.
- Save code blocks for re-use as snippets.
- Build ColdFusion expressions from the complete set of ColdFusion functions, constants, operators, and variables available in the Expression Builder.
- Debug dynamic pages against ColdFusion Server.
- View your site's structure in the Visualizer.
- Validate HTML and CFML code.
- Verify links for individual files or entire projects.
- Enable version source control of your files for team development.

About ColdFusion Documentation

The documentation set is designed to provide support for all components of the ColdFusion development system. Both the print and online versions are organized to allow you to quickly locate the information you need.

The documentation set contains:

Getting Started with ColdFusion — Covers system installation and basic configuration, describes the components of the ColdFusion development system, and introduces the ColdFusion Markup Language (CFML).

Administering ColdFusion Server — Describes configuration options for maximizing performance, managing data sources, setting security levels, and a range of development and site management tasks.

Developing Web Applications with ColdFusion — Presents the fundamentals of ColdFusion application development and deployment, including data sources, user interfaces, and Web technologies. The development tools in ColdFusion Studio are covered in detail.

Advanced ColdFusion Development — Gives an overview of CFML elements such as functions, expressions, arrays, scripting, and XML data exchange. It also discusses custom tags, ColdFusion API tags, integrating object technologies, and site management.

CFML Language Reference — Provides the complete syntax, with example code, of all CFML tags and functions.

Quick Reference Card — An online (Acrobat) guide to CFML.

Documentation distribution

The ColdFusion CD-ROM contains the complete document set. The setup program installs the document set by default.

The print manuals are available in Adobe Acrobat (PDF) format from the dochome.htm page in the `/cfdocs` directory of your Web root. If the files are not available locally, you get them from our Web site at http://www.allaire.com/products/COLDFUSION/Documentation.cfm.

You can also access the documentation in HTML from both of these locations.

Documentation conventions

When reading, please be aware of these formatting cues:

- Code samples, filenames, and URLs are set in a distinct font.
- Notes and tips are identified by bold type in the margin.
- Bulleted lists present options and features.
- Numbered steps indicate procedures.
- Tool button icons are generally shown with procedure steps.
- Menu levels are separated by the greater than (>) sign.
- Text for you to type in is set in *italics*.

Reading online documentation

You can open the online documents in a number of ways:

- From your browser, click the ColdFusion Documentation link on the Welcome to ColdFusion page. Each page contains links to other documents and a search window.
- In ColdFusion Studio, click the Help tab in the Resources area to open the help tree. You can expand the list to select topics by title.

Tip You can extend the online documentation in Studio by adding your own HTML files. Just copy a folder to the Help directory under the ColdFusion Studio directory. Press F5 to refresh the Help reference list. You can now browse and search these files in the Help References.

Contacting Allaire

Corporate headquarters

Allaire Corporation
One Alewife Center
Cambridge, MA 02140

Tel: 617.761.2000 voice
Fax: 617.761.2001 fax

http://www.allaire.com

Technical support

Telephone support is available Monday through Friday 8 A.M. to 8 P.M. Eastern time (except holidays).

Toll Free: 888.939.2545 (U.S. and Canada)
Tel: 617.761.2100 (outside U.S. and Canada)

Postings to the ColdFusion Support Forum can be made at any time.

Sales

Toll Free: 888.939.2545
Tel: 617.761.2100
Fax: 617.761.2101

Email: sales@allaire.com
Web: http://www.allaire.com/store

CHAPTER 2

Functions and Expressions

This chapter describes ColdFusion expressions — powerful, easy-to-learn language constructs that allow you to create more sophisticated applications.

Contents

- About ColdFusion Expressions 8
- Creating Expressions in ColdFusion Studio 9
- The Structure of Expressions 9
- Operators 16
- Functions 21
- Using Pound Signs 24
- Typeless Expression Evaluation 27
- Examples of Typeless Expression Evaluation 29
- Debugging and Troubleshooting Expressions 30

About ColdFusion Expressions

At a basic level, you use expressions in ColdFusion application pages to perform operations on data. Since you can embed expressions in ColdFusion Markup Language (CFML) tags and CFScript blocks, you can create standard programming logic to manipulate data.

For example, you can use expressions in the ColdFusion tags CFSET and CFIF to create standard IF-ELSE statements:

```
<CFIF 1 + 1 is 2>
    The world is rational.<BR>
<CFELSE>
    Go home, today's a bad day.<BR>
</CFIF>
```

Some of the uses for expressions are to:

- Perform mathematical calculations.
- Manipulate strings.
- Execute date-and-time operations.
- Format dates, times, and numbers.
- Add data to arrays and structures.

What's in an expression?

Expressions can contain a wide variety of objects or elements. Expressions can be built using the following components:

- Basic terms: numbers, strings, Boolean (logical) values, date-and-time objects, lists, and complex objects like arrays, structures, queries, and COM objects
- Variables that store some previously computed data
- Functions that manipulate data in predefined ways
- Operators that combine simple expressions to create more complex ones

Expression examples

For example:

- `1 + 1` is a mathematical expression that evaluates to `2`.
- `1 is 2` is a logical expression that evaluates to the string `NO` (also the Boolean value `FALSE`).
- `Left("Monkey", 4)` is a string expression that evaluates to the string `Monk`.
- DateFormat(CreateDate(1998, 9, 11), "dddd, mmmm d, yyyy") is a date-and-time formatting operation that evaluates to Friday, September 11, 1998.

See the *CFML Language Reference*for a full catalog of ColdFusion tags and functions.

Creating Expressions in ColdFusion Studio

In ColdFusion Studio, you can use the Expression Builder to create CFML expressions. You can use this visual tool to combine functions, operators, and values into CFML expressions.

To build expressions:

1. Place the cursor at the point in the document where you want to insert the expression.
2. Right-click and select Insert Expression or choose Tools > Expression Builder. You can open and close the list of Expression Elements to show or hide the functions, constants, operators, and variables.
3. In the Functions list, choose an expression type to display the expression elements in the adjoining pane. For example, select Date and Time to see all the ColdFusion functions for manipulating date and time values.
4. Double-click an element to add it to the element list.
5. Add operators by clicking on them in the operator toolbar.
6. Click Insert to add the expression in the current document.

The Structure of Expressions

This section is about the objects that can be used to build expressions. The objects and their properties are described in detail. Knowledge of the material in this section will enable the construction of powerful and flexible ColdFusion expressions.

The basic terms — numbers, strings, Boolean (logical) values, date-and-time objects, lists, and complex objects such as arrays, structures, queries, and COM objects — are the simplest expressions that exist. This section defines the meaning and properties of each of these terms.

Numbers

ColdFusion supports both integer numbers (numbers with no decimal part, for example, 14) and real numbers (numbers with a decimal part, such as 3.127). Real numbers are also known as floating-point numbers. Integer and real numbers can be freely intermixed in expressions, so, for example, `1.2 + 3` evaluates to `4.2`.

ColdFusion can work with both very large and very small real numbers. The range of ColdFusion numbers is approximately $\pm 10^{300}$, or ±1 with 300 zeros after it. Most operations are accurate to 12 digits after the decimal point.

Scientific notation

In ColdFusion, numbers can also be represented in what is known as engineering, or scientific notation. The format of such numbers is `xEy`, where `x` is a positive real number in the range 1.0 (inclusive) to 10 (exclusive) and `y` is an integer number. The value of a number in the engineering notation is x times 10^y, so, for example, 4.0E2 is 4.0 times 10^2 which is equal to 400. Similarly, 2.5E-2 is equal to 2.5 times 10^{-2}, which is equal to 0.025. Engineering notation is very useful for writing very large and very small numbers.

Strings

In ColdFusion, text values are stored in strings. Strings are pieces of text delimited on both ends by either single or double quotes. For example, the two strings below are equivalent:

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

An empty string can be written as `""` (a pair of double quotes with nothing in between), or as `''` (a pair of single quotes with nothing in between). Strings can have arbitrary size, limited only by the amount of available memory on the ColdFusion server.

Using quote marks and pound signs

Strings can use either single or double quotes. To use a single quote inside a string that is single quoted, use two single quotes. This is known as *escaping* the single quote:

```
'Single Quote: '' Double Quote: "'
```

To use a double quote inside a double-quoted string, use two double quotes. This is known as *escaping* the double quote:

```
"Single Quote: ' Double Quote: """
```

Because strings can be in either double quotes or single quotes, both of these strings are equivalent.

To insert a pound sign in a string, the pound sign must be escaped, or doubled, as in:

```
"This is a pound sign ##"
```

Boolean values

Boolean values store the result of a logical operation. Thus their value can be one of truth, or falsity. ColdFusion has two special constants — TRUE and FALSE — for each of these values. For example:

- `1 IS 1` is an expression that evaluates to TRUE.
- `"Monkey" CONTAINS "Money"` is an expression that evaluates to FALSE.

The two Boolean values can be used directly in expressions, as in:

```
<CFSET UserHasBeenHere=TRUE>
```

The numerical value of TRUE is 1. The numerical value of FALSE is 0. When converted to a string, TRUE becomes “YES” and FALSE becomes “NO”.

Date-and-time values

ColdFusion can perform a variety of operations on date-and-time values. Date-and-time values identify a date and time in the range 100AD to 9999AD.

There are a variety of ways in which a date-and-time value can be entered in ColdFusion. You can use the functions that create date-and-time objects using various criteria. (See the *CFML Language Reference* for information about date-and-time functions.)

You can also directly enter a date-and-time object in a familiar format:

```
"October 30, 1998"
"Oct 30, 1998"
"Oct. 30, 1998"
"10/30/98"
"1998-30-10"
```

21st century dates

Two-digit years from 00 to 29 are treated as 21st century dates; 30 to 99 are treated as 20th century dates.

```
"October 30, 2015"
"October 30, 15"
```

Time formats

If no time part is specified, time is set to 12:00am. Times can be added in a variety of common formats as well:

```
"October 30, 1998 02:34:12"
"October 30, 1998 2:34a"
"October 30, 1998 2:34am"
"October 30, 1998 02:34am"
"October 30, 1998 2am"
```

The time part of the object is accurate to the second.

Note Internally to ColdFusion, date-and-time values are represented on a time line as a subset of the real numbers. This is done for efficiency in evaluation and because it directly mimics the method used by many popular database systems, including Microsoft Access. 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.

Thus, arithmetic operations can be used to manipulate date-and-time values. For example, Now() + 1 will evaluate to tomorrow at the same time. However, we strongly discourage ColdFusion developers from using this potentially troublesome method of manipulating date-and-time objects. Date-and-time manipulation routines should be used instead.

Lists

Lists are objects that enable ColdFusion developers to easily perform sophisticated manipulation operations on collections of elements returned by Web browsers and some ColdFusion functions such as ValueList, QuotedValueList.

Lists are a special kind of string. When a string is viewed as a list, all characters inside it are divided into two types—delimiters and non-delimiters. Elements of the list consist of the text between (one or more) delimiters. Elements consist entirely of non-delimiter characters. In general, the structure of a string viewed as a list can be described as:

```
"TextElementDelimiterTextElementTextDelimiterText…"
```

Here's an example of how this might look:

```
“Biology,Chemistry,Geology,Physics”
```

Examples

- “1,2,3” is a three-element list with “,” as the delimiting character. The first element is “1”, the second is “2”, and the third is “3”.
- “First;Second” is a two-element list with “;” as the delimiting character. The first element is “First”, the second is “Second”.
- “A,B;C,D” is a three-element list with “,” as the delimiting character. The first element is “A”, the second is “B;C”, and the third is “D”.
- “A,B;C,D” is a two-element list with “;” as the delimiting character. The first element is “A,B”, the second is “C,D”. “A,B;C,D” is a four-element list with both “,” and “;” as the delimiting characters. The first element is “A”, the second is “B”, the third is “C,” and the fourth is “D”.

As the examples show, a list can have more than one delimiting character. The default delimiting character, used by all list processing functions is a comma: “,”.

Elements in lists can be separated by more than one delimiter. For example, the list “1xx2xyz3” has the three elements “1”, “2”, and “3” as long as the delimiters include all of “x”, “y”, and “z”.

Delimiters before the first element and after the last element will be ignored. Thus the list “1xx2xyz3” from the previous example will be processed in the same way as the list “zzy1xx2xyz3yz”.

Note that the structure of lists is flat – that is, lists cannot be nested into one another. Also, lists can contain no “empty” elements. A list can be empty, however. The empty list is equivalent to the empty string `""`.

A word of caution: white space is not considered a delimiter. When using lists where elements may be separated by white space as well as other delimiters, be sure to add the white space characters to the delimiters. For example, "1, 2, 3" should probably be processed with both the comma and the space as delimiters.

Structures

You can use structures to create and maintain key-value pairs. You can also use structures to refer to related string values as a unit rather than individually. For example, a structure lets you build a collection of related variables that are grouped under a single name. You can also use structures to create associative arrays.

You create structures by assigning a variable name to the structure with the StructNew function. For example, to create a structure named *employee*, use this syntax:

```
<CFSET employee=StructNew()>
```

You can add key-value pairs to the structure using the StructInsert function:

```
<CFSET value=StructInsert(structure_name, key, value)>
```

For more information about structures, see the Working with Structures chapter in this book.

Arrays

Arrays are essentially tables of objects or data that can be indexed. Although the ArrayNew function only supports creating up to three-dimensional arrays, there is no limit on array size or maximum dimension. You can piece together arrays of dimension greater than three by adding arrays to array indexes.

Array objects can be created by assigning an existing array to a new variable:

```
<CFSET myarray2=myarray>
```

In this case, a separate copy of the data in myarray is copied to myarray2. Changes made in myarray are not reflected in myarray2. It is very important to understand that such assignments are very resource-intensive since the entire array is copied from one variable to the other. This operation can significantly affect performance when large arrays are involved.

Elements stored in an array are referenced as follows:

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

For more information about arrays, see the Working with Arrays chapter in this book.

Note Complex objects, such as arrays, structures, queries, and COM objects, are passed to custom tags surrounded by pound signs (#).

Queries

Like arrays, ColdFusion queries can be referenced as objects by assigning a query to a variable:

```
<CFQUERY NAME=myquery
    DATASOURCE=mydata
    SELECT * FROM CUSTOMERS
</CFQUERY>

<CFSET myquery2=myquery>
```

In this case (unlike the same operation with arrays) the query is *not* copied. Instead, both names point to the record set data so that if you make changes to the table referenced in the query, the original query and the query object myquery2 will both reflect those changes.

Query columns can be referenced as if they were one-dimensional arrays using cursor-style processing. However, they are not dynamically scalable.

```
<CFSET myvar=queryname.columnname[index]>
```

Multiple queries of the same name can now be run in the same application page, so dynamic expression evaluation doesn't need to be used to run queries inside a loop. And queries can be passed as objects to custom tags.

However, queries and variables cannot have the same name at the same time in the same application page. You can make a query available to all ColdFusion applications on a specified server by assigning a query to a server variable:

```
<CFSET Server.query=myquery>
```

When you want to clear the server scope query, you reassign the query object:

```
<CFSET Server.query=0>
```

Note Complex objects, such as arrays, structures, queries, and COM objects are passed to custom tags surrounded by pound signs (#).

COM objects

COM (Component Object Model) objects are non-visual components that encapsulate specific functionality you can invoke in your application pages. ActiveX, OCX, CORBA, and ADO objects are examples of COM objects.

COM objects are created using CFOBJECT and CFSET (when the right-hand side of the expression evaluates to an object):

```
<CFOBJECT ACTION="Create"
    NAME="Mailer"
    CLASS=SMTP.Mailer>

<CFSET myvar=Mailer>
<CFSET MessageObj=Mailer.GetCurrentMessageObject()>
```

COM objects generally contain methods, like functions, you can use to execute certain operations:

```
<CFSET temp=Mailer.SendMail()>
```

COM objects also generally contain properties you can read and write using ColdFusion variables:

```
<CFSET Mailer.FromName=Form.fromname>
```

Properties can be invoked on either side of an assignment:

```
<!--- To set the Mailer.Subject property --->
<CFSET Mailer.Subject=form.subject>

<!--- To get the Mailer.Subject property --->
<CFSET subject=Mailer.Subject>
```

Methods and properties can return objects such as arrays, queries, and other COM objects.

For a COM object to be used by ColdFusion, it needs to be registered and it needs to expose the IDispatch interface. For more information about COM objects, see the Using Objects chapter in this book.

Note Complex objects, such as arrays, structures, queries, and COM objects are passed to custom tags surrounded by pound signs (#).

Variables

When variables are used in ColdFusion expressions, the value stored in the variable is returned. The values can be one of the previously described basic objects: numbers, strings, Boolean values, date-and-time objects, or lists.

Variable names must begin with a letter that can be followed by any number of letters, numbers, or the underscore character "_". For example, TheVariable_1 and TheVariable_2 are valid variable names, while 1stVariable and WhatAVariable! are not.

Sometimes variable names can begin with a qualifier that itself is a variable name. The qualifier name of a variable is separated from the qualified name with a period character (.). For example, Form.MyVariable is a valid qualified variable name. The qualifier, in this case "Form," signifies that we are interested in the form variable MyVariable, as opposed to, for example, the client variable MyVariable (Client.MyVariable). Qualifiers are also known as scopes. Thus MyVariable is said to belong to the Form scope.

In some cases, a variable must have pounds signs around it to allow ColdFusion to distinguish it from string or HTML text and to insert its value as opposed to its name. For more information on how to use pound signs in expressions see Using Pound Signs.

Functions

Because ColdFusion functions return basic objects, such as numbers, strings, Boolean values, date-and-time objects, lists, arrays, structures, queries, and COM objects, their results are basic expression terms.

Operators

Operators combine sub-expressions to create more complex expressions. The general syntax for using operators is:

```
Expression Operator Expression
```

For example, 2 * (3 + 4) is a valid expression in which the plus (+) operator is used to combine two numbers into an expression (which evaluates to 7), while the multiplication (*) operator is used to combine the number 2 with the expression (3 + 4) to produce the final result of 14.

ColdFusion has four types of operators:

- Arithmetic operators
- String operators
- Decision, or comparison, operators
- Boolean operators

Arithmetic operators

ColdFusion has nine arithmetic operators for addition, subtraction, multiplication, division, remainder calculation, integer division, exponentiation, and sign changing.

Arithmetic Operators

Operator	Description
+, -, *, /	The basic arithmetic operators: addition, subtraction, multiplication, and division. In the case of division, the right operand cannot be zero.
MOD	Returns the remainder (modulus) after a number is divided by a divisor. The result has the same sign as the divisor. The right operand cannot be zero. For example, 11 MOD 4 is 3.
\	Divides two integer values. Use the \ (trailing slash) to separate the integers. The right operand cannot be zero. For example, 9 \ 4 is 2.
^	Returns the result of a number raised to a power (exponent). Use the ^ (caret) to separate the number from the power. The left operand cannot be zero. For example, 2 ^ 3 is 8.

Unary arithmetic operators

There are two unary arithmetic operators for setting the sign of a number either positive or negative (+ or -). They modify the value as you would expect. For example:

- +2 is 2
- -2 is (-1)*2

Examples

```
<CFSET DoubleNumber=2 * Form.Number>
<CFSET Number=(2 + 3) * 2>
<CFIF Form.Number IS DoubleNumber / 2>
...CFML tags...
</CFIF>
```

String operators

In ColdFusion multiple strings can be concatenated with the & (ampersand) operator.

ColdFusion also supports the automatic concatenation of variable values and function return results delimited by pounds inside strings. For more information, see Using Pound Signs.

Examples

```
<CFSET Text1="Jack is not " & (Form.Height * 2)>
<CFSET Text2="This text " & "continues…">
```

You can also output strings using variables, as in this example:

```
<CFSET Text1=”John “>
<CFOUTPUT>#Text1#Smith
</CFOUTPUT>
```

Decision, or comparison, operators

ColdFusion has eight decision, or comparison, operators that produce a Boolean TRUE/FALSE result based on the result of the test they perform on their two arguments.

Decision Operators	
Operator	**Description**
IS	Performs a case-insensitive comparison of the two values and returns true if the values are identical.
IS NOT	Opposite behavior of *is*.
CONTAINS	Checks to see if the value on the left is contained in the value on the right and returns true if it is.
DOES NOT CONTAIN	Opposite behavior of *contains*.
GREATER THAN	Checks to see if the value on the left is greater than the value on the right and returns true if it is.
LESS THAN	Opposite behavior of *greater than*.
GREATER THAN OR EQUAL TO	Checks to see if the value on the left is greater than or equal to the value on the right and returns true if it is.
LESS THAN OR EQUAL TO	Checks to see if the value on the left is less than or equal to the value on the right and returns true if it is.

Shorthand notation for Boolean operators

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

Shorthand Notation for Boolean Operators	
Operator	**Alternative name(s)**
IS	EQUAL, EQ
IS NOT	NOT EQUAL, NEQ
CONTAINS	Not available
DOES NOT CONTAIN	Not available

Shorthand Notation for Boolean Operators (Continued)	
Operator	**Alternative name(s)**
GREATER THAN	GT
LESS THAN	LT
GREATER THAN OR EQUAL TO	GTE, GE
LESS THAN OR EQUAL TO	LTE, LE

Example

```
<CFSET ResultValue=4 IS NOT Form.Number>
```

If Form.Number is not 4, ResultValue would be TRUE, which would be output as the string "TRUE".

Boolean operators

Boolean, or Logical, operators perform logical connective and negation operations. The operands of Boolean operators are Boolean (TRUE/FALSE) values. ColdFusion has the following six Boolean operators:

Boolean Operators	
Operator	**Description**
NOT	Reverses the value of an argument. For example, NOT TRUE is FALSE and vice versa.
AND	Returns TRUE if both arguments are TRUE; returns FALSE otherwise. For example, TRUE AND TRUE is TRUE, but TRUE AND FALSE is FALSE.
OR	Returns TRUE if any of the arguments is TRUE; returns FALSE otherwise. For example, TRUE OR FALSE is TRUE, but FALSE OR FALSE is FALSE.
XOR	Exclusive or—either, or, but not both. Returns TRUE if the truth values of both arguments are different; returns FALSE otherwise. For example, TRUE XOR TRUE is FALSE, but TRUE XOR FALSE is TRUE.

Boolean Operators (Continued)	
Operator	**Description**
EQV	Equivalence both true or both 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. A IMP B is the truth value of the logical statement "If A Then B." A IMP B is FALSE only when A is TRUE and B is FALSE.

The Boolean operators are most often used in CFIF and CFELSEIF tags to control the execution of CFML tags in an application page. They are also used in the CONDITION attribute of the CFLOOP tag. Boolean operators are also used in the IIf function.

Examples

```
<CFIF IsDefined("Form.FName") AND IsDefined("Form.LName")>
... CFML tags...
</CFIF>

<CFLOOP CONDITION="NOT (IndexValue LTE 1 OR IndexValue GTE 10)">
... CFML tags...
</CFLOOP>
```

Operator precedence

The order of precedence controls which operator is evaluated first in an expression. Operators on the same line have the same precedence.

Operator precedence, highest to lowest

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

To enforce a specific 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

Parenthesized expressions can be arbitrarily nested. When in doubt about the order in which operators in an expression will be evaluated, always use parentheses.

Functions

Functions are predefined operations you can use to manipulate data. In ColdFusion, functions let you perform decision-making, date/time-formatting, and other common operations automatically.

ColdFusion provides a variety of functions that perform many types of tasks. For example, you can use a mathematical function to take the square root of a number, or a string function to find the first letter of a word.

See the *CFML Language Reference* for a full catalog of ColdFusion functions.

Function types

In ColdFusion, functions are organized by category, as shown in the following table:

Function Categories

Category	Purpose
Administrative functions	Perform actions on client variables.
Array functions	Create, edit, and manage arrays.
Date and Time functions	Perform date-and-time actions.
Decision functions	Test for arrays, queries, and simple values.
Display and Formatting functions	Control the display of dates, times, and numbers.
Dynamic Evaluation functions	Evaluate dynamic expressions, define variable values.
International locale functions	Perform date, time, and currency formatting based on a specified locale, such as French (Standard) and French (Canadian).
List functions	Perform actions on lists.
Mathematical and Trigonometric functions	Perform mathematical operations on values.

Function Categories (Continued)

Category	Purpose
Query functions	Perform actions on queries.
Strings functions	Perform actions on text values.
Structure functions	Create, edit, and manage structures.
System-level functions	Perform actions on directories and paths.

Function usage

All functions return a value, which is one of the basic expression objects:

- Numbers
- Strings
- Boolean values
- Date-and-time objects
- Lists
- Arrays
- Structures
- Queries
- COM objects

Often, the returned value is computed based on some data passed to the function from the ColdFusion application page. Data is passed to functions via their arguments (also known as parameters).

Although some functions take no parameters, such as, Now() which returns the current date and time, most functions take at least one argument. Any valid ColdFusion expression can be an argument to a function. This means that functions can take functions as arguments. (This is known as nesting functions.)

The following table illustrates function syntax and usage guidelines.

Function Syntax Guidelines

Usage	Example
The exception — no arguments	`Function()`
Basic format	`Function(Data)`
Nesting functions	`Function1(Function2(Data))`

Function Syntax Guidelines (Continued)

Usage	Example
Separate multiple arguments with commas	`Function(data1, data2, data3)`
Enclose string arguments in single or double quotes	`Function('This is a demo') Function("This is a demo")`
Arguments are expressions	`Function1(X*Y, Function2("Text"))`

To learn how to insert functions in various types of expressions see Using Pound Signs.

Optional arguments in functions

Some functions may take optional arguments after their required arguments. If omitted, optional arguments take some default value. For example:

```
Replace("FooFoo", "Foo", "Boo") returns "BooFoo"
Replace("FooFoo", "Foo", "Boo", "ALL") returns "BooBoo"
```

The difference in behavior is explained by the fact that the Replace function takes an optional fourth argument which specifies the scope of replacement. The default value is “ONE” which explains why only the first occurrence of “Foo” was replaced with “Boo”. In the second example, a fourth argument is provided that forces the function to replace all occurrences of “Foo” with “Boo”.

Functions that return a Boolean

When you test the return of any function that returns a Boolean value, the output will always appear to be YES or NO. The actual Boolean return is TRUE or FALSE, but when tested in a CFIF statement, the Boolean return is converted to its string equivalent, YES or NO.

Each of the following examples demonstrates a valid test for a Boolean return:

```
<CFIF #booleanfunction(arg1, arg2)# IS TRUE>
<CFIF #booleanfunction(arg1, arg2)# IS "YES">
<CFIF #booleanfunction(arg1, arg2)# IS 1>
<CFIF #booleanfunction(arg1, arg2)#>
```

Note in the second example that YES must be enclosed in quotation marks since it is resolved as a string. This is an important point to keep in mind for an example such as:

```
<CFSET ReturnValue= #booleanfunction(arg1, arg2)#>
<CFIF ReturnValue IS "YES">
```

Preferred method

The preferred method from the previous examples omits the explicit condition from the expression, as shown below:

```
<CFIF #booleanfunction(arg1, arg2)#>
```

Using Pound Signs

Pound signs (#) have special meaning in ColdFusion. When a CFML application page is processed, ColdFusion treats text delimited by pound signs differently from plain text.

Two simple and very important points about pound signs in CFML are:

- Use pound signs to distinguish expressions from plain text.
- When expressions are evaluated, the resulting value is substituted for the expression text.

For example, to output the current value of a variable named “Form.MyFormVariable,” you must delimit the variable name with pound signs:

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

When ColdFusion processes an expression, it replaces the text of the expression and the two pound signs around it with its resulting value. In the example above, the expression #Form.MyFormVariable# is replaced with whatever value has been assigned to it.

While the guidelines for using pound signs in CFML are simple, there is still some possibility for confusion to arise. This is particularly true in cases where expressions and plain text are mixed together. The following sections provide more details on how pound signs should be used in CFML.

Pound signs inside CFOUTPUT tags

Expressions containing a single variable or a single function can be used freely inside CFOUTPUT tags as long as they are enclosed in pound signs.

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

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

<CFOUTPUT>
Cos(0) is #Cos(0)#
</CFOUTPUT>
```

If pounds are not used around these expressions, the expression text rather than the expression value will appear in the output generated by the CFOUTPUT statement.

Note that two expressions inside pound signs can be adjacent to one another, as in

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

Complex expressions

Complex expressions involving one or more operators cannot be inserted inside CFOUTPUT tags. The following example will produce an error.

```
<CFOUTPUT>1 + 1 is #1 + 1#</CFOUTPUT>
```

To insert the value of a complex expression in the output generated by a CFOUTPUT statement, use CFSET to set a variable to the value of the expression and use that variable inside the CFOUTPUT statement, as is shown below:

```
<CFSET Result=1 + 1>
<CFOUTPUT>1 + 1 is #Result#</CFOUTPUT>
```

Pound signs inside strings

Expressions containing a single variable or a single function can be used inside strings as long as they are enclosed in pound signs.

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

ColdFusion automatically replaces the expression 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 not used around these expressions, the expression text as opposed to the expression value will appear 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 in the case of the CFOUTPUT statement, in strings two expressions can be adjacent to each other, as in

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

Note that the double quotes around “Moon” and “Monkey” need not be escaped (or doubled, as in ""Moon"" and ""Monkey""). This is because the text between the pound signs is treated as an expression that is evaluated first before its value is inserted inside the string.

Inserting complex expressions in strings

Complex expressions involving one or more operators cannot be inserted inside strings. The following example produces an error:

```
<CFSET TheString="1 + 1 is #1 + 1#">
```

To insert the value of a complex expression inside a string, use CFSET to set some variable to the value of the expression and use that variable inside the string, or use the string concatenation operator:

```
<CFSET Result=1 + 1>
<CFSET TheString="1 + 1 is #Result#">
<CFSET TheString="1 + 1 is " & (1 + 1)>
```

To insert the pound character inside a string, use two pound signs as shown below:

```
<CFSET TheString="This is a pound sign ##.">
```

Pound signs inside tag attribute values

The rules for using pound signs inside strings apply to the use of pound signs inside tag attribute values. The following example demonstrates the point:

```
<CFCOOKIE NAME="TestCookie"
    VALUE="The value is #CookieValue#">
```

If the value of a tag attribute is a variable, function, or array element, use the following syntax:

```
<CFCOOKIE NAME="TestCookie"
    VALUE=#CookieValue#>
<CFCOOKIE NAME="TestCookie"
    VALUE=#CookieValueArray[Index]#>
```

This usage is more efficient than VALUE="#CookieValue#".

Nested pound signs

There are very few cases in which pound signs can be nested inside the same expression. Usually, the need for nested pound signs arises because of the high degree of complexity of the expression. The following example shows a valid use of nested pound signs:

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

The pound signs need to be nested so that the values of the variables FirstName and LastName are inserted in the string whose length the Len function will calculate. Generally, the existence of nested pounds implies the presence of a complicated expression. For example, the above piece of CFML could be rewritten to improve its 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)#)#">
```

All of the above statements result in errors. As a general rule, *never* put pound signs around function arguments.

Pound signs in general expressions

Allaire recommends that pound signs be used only where necessary. The following example demonstrates the preferred method for referencing variables.

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

It is a cleaner and more efficient method.

In contrast, note the following example, which uses pound signs unnecessarily:

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

Typeless Expression Evaluation

Typelessness in the evaluation of expressions refers to a system's ability to automatically convert between data types in order to satisfy the requirements of the operations in expressions.

ColdFusion has a typeless expression evaluation system that simplifies data manipulation for Web developers. Instead of worrying about compatibility between data types and the conversions from one data type to another, ColdFusion developers can focus on the operations they would like to perform on the data.

Operation-driven evaluation

Traditional programming languages enforce strict rules about mixing different types of objects in expressions. For example, in a language such as Fortran, Pascal, C/C++, or Basic, the expression ("8" * 10) produces some form of compile or run-time error because the multiplication operator expects two numerical operands and "8" is a string. Developers using such languages must constantly worry about converting between data types to ensure error-free program execution. For example, the above expression may 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 tries to convert its operands to numbers. Since “8” can be successfully converted to the number 8, the expression evaluates to 80.

How ColdFusion processes expressions

In general, ColdFusion processes an expression using the following steps:

1. In the case of operators, ColdFusion determines the required operands. For example, the multiplication operator requires its operands to be numbers and the CONTAINS operator requires its operands to be strings. In the case of functions, the required function arguments are determined. For example, the Min function expects two numbers as arguments, and the Len function expects a string.
2. In the case of operators, ColdFusion evaluates all operands. In the case of functions, all arguments are evaluated.
3. In the case of operators, ColdFusion converts all operands that are of a different type from the required type to the required type. In the case of functions, all arguments that are of a different type from the required type are converted to the required type. (If a conversion fails, and ColdFusion reports an error.)

Because ColdFusion performs automatic conversions based on the operations that are involved in the evaluation of an expression, its typeless expression evaluation mechanism is essentially operation-driven evaluation. Operation-driven evaluation lets ColdFusion developers focus on what they want to do with data, not on the details of ensuring error-free expression evaluation.

Conversion between types

While the typeless expression evaluation mechanism in ColdFusion is very powerful, it cannot perform miracles – not all conversions that seem obvious to a ColdFusion developer can be performed automatically. For example, "eight" * 10 will produce an error since ColdFusion does not convert the string “eight” to the number 8.

While typeless expression evaluation does provide developers with a lot of flexibility at practically no cost, it has its intricacies. It can be helpful to understand the way in which some special values and types — such as Boolean — are converted.

The following table explains how conversions are performed. The first column lists the value to be converted. The last four columns list the result of the conversion to a Boolean, a number, a date-and-time value, and a string. Note that complex types, such as arrays, structures, queries, and COM objects, cannot be converted.

Examples of Data Type Conversions

Value	As Boolean	As Number	As Date-and-Time	As String
“YES”	TRUE	1	Error	“YES”
“NO”	FALSE	0	Error	“NO”
TRUE	TRUE	1	Error	“YES”
FALSE	FALSE	0	Error	“NO”
Number	TRUE if Number is not 0, FALSE otherwise	Number	See Date-and-time values.	Number is converted using a default format.

Examples of Data Type Conversions (Continued)

Value	As Boolean	As Number	As Date-and-Time	As String
String	If “YES” or “NO” or if the string can be converted to a number, it is treated just like "Number."	If it can be converted to a number, it is.	A date-and-time value if String is an ODBC date, time, or timestamp; or if it is expressed in a standard US date or time format, including the use of full or abbreviated month names. Days of week or unusual punctuation result in error. Dashes, forward-slashes, and spaces are generally allowed.	String
Date	Error	See Date-and-time values.	Date	Automatic conversion is to ODBC timestamp type.

Examples of Typeless Expression Evaluation

The following examples demonstrate ColdFusion’s typeless expression evaluation.

Example 1

```
2 * TRUE + "YES" - (’y’ & "es")
Value as string: "2"
```

Explanation: (2*TRUE) is equal to 2; ("YES"-"yes") is equal to 0 because “Yes” converts to 1. And, of course, 2 * 0 equals 0.

Example 2

```
TRUE AND 2 * 3
Value as string: "YES"
```

Explanation: 6 is TRUE as a Boolean; TRUE AND TRUE is TRUE.

Example 3

```
"Five is " & 5
Value as string: "Five is 5"
```

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

Example 4

```
DateFormat("October 30, 1998" + 1)
Value as string: "31-Oct-98"
```

Explanation: The addition operator forces the string “October 30, 1998” first to be converted to a date-and-time object and then again converted to a number. The number is incremented by one. The DateFormat function requires its argument to be a date-and-time object; thus the result of the addition is converted back to a date-and-time object. The addition of 1 has moved the date one day ahead.

Debugging and Troubleshooting Expressions

There are several ways to test expressions in ColdFusion application pages. If you’re using ColdFusion Studio to develop your application, you can use the interactive debugger to do the following:

- Set watches.
- Use the evaluator to troubleshoot your expressions.

Or, you can also test expressions by assigning the expression to a variable and outputting its contents using the CFOUTPUT tag.

Setting watches in the debugger

The interactive debugger in ColdFusion Studio lets you set breakpoints and watches to evaluate ColdFusion expressions. You can evaluate ColdFusion expressions at breakpoints using the Watches pane of the debugger.

You can use the evaluator box at the top of the Watches pane of the Debug window to evaluate arbitrary expressions when the debugger is suspended at a breakpoint. Use the evaluator when you want to know how an expression evaluates as you step through code, line by line.

Watches allow you to evaluate the same expression or variable every time you stop execution. When you set a watch, the debugger evaluates the watched expression. A hand pops up when the expression’s value changes from one line to the next as you step through code.

To set watches:

1. Choose Debug > Watches or click the Watches button on the Debug toolbar. The Watches pane appears.
2. Cut and paste the expression or variable you want to watch into the list box at the top of the pane.
3. Choose Evaluate to find the value of the expression at the next breakpoint or line where the Debugger stops.

The Evaluator window shows the results of the evaluation at the current point in processing the page.

4. Choose Watch to add the expression in the evaluator list box to the list of watched expressions.

   The Watch area shows the values of watched expressions and any error messages in resolving these parameters.

5. Press the Start/Continue button to continue debugging.
6. When you are finished debugging, press End.

Note You can use the evaluator to change values of variables, create new variables, or practice using ColdFusion functions in your expressions.

Testing expressions using CFSET and CFOUTPUT

One of the simplest ways to test an expression is to write a piece of code that assigns an expression to a variable using the CFSET tag and then display the contents of the variable using CFOUTPUT. You can use this technique directly in the application page in which you are experiencing the problem.

The following example illustrates how to debug an expression using CFSET and CFOUTPUT. Imagine we are using a complex expression inside the DESTINATION attribute of the CFFILE tag.

```
<CFFILE ACTION="Upload"
    FILEFIELD="FileFormField"
    DESTINATION="ExpandPath('text.txt') & '\text.txt'">
```

To debug the expression, insert a simple CFSET and CFOUTPUT statement before the CFFILE tag:

```
<CFSET TempVariable=ExpandPath('text.txt') & '\text.txt'>
<CFOUTPUT>Destination="#TempVariable#"</CFOUTPUT>
```

This CFML code assigns the expression to a temporary variable (*TempVariable*) and displays it enclosed in quotes. The quotes are very useful when debugging string expressions in which unwanted spaces need to be detected.

The above test code results in the following page output.

```
"d:\webdocs\text.txt\text.txt"
```

The CFOUTPUT produces results that are easily interpreted. In this case, the error occurs because of the duplication of the filename at the end of the expanded path. Using the simple test script, the problem in the expression was resolved and the correct CFFILE tag was specified as:

```
<CFFILE ACTION="Upload"
    FILEFIELD="FileFormField"
    DESTINATION=ExpandPath("text.txt")>
```

For more information

For more information on debugging and troubleshooting your applications, see the Debugging and Troubleshooting chapter of the *Developing Web Applications with ColdFusion* book.

See the *CFML Language Reference* for a full catalog of ColdFusion tags and functions.

CHAPTER 3

Dynamic Expression Evaluation

This chapter examines dynamic expression evaluation in ColdFusion. As a prerequisite to understanding this material, a CFML developer should be thoroughly familiar with ColdFusion expressions.

See the Functions and Expressions chapter of this book for information on ColdFusion expressions. See the *CFML Language Reference* for descriptions of all ColdFusion functions.

Contents

- About Dynamic Expression Evaluation 34
- Evaluating String Expressions 35
- Pound Signs Inside String Expressions 36
- Setting Variables Dynamically 37

About Dynamic Expression Evaluation

Dynamic expression evaluation is an advanced technique in ColdFusion application development that allows CFML expressions to be created dynamically using string operations and evaluated as needed. Expressions are evaluated using one or more of the four dynamic expression evaluation functions — Evaluate, SetVariable, IIf, and DE.

To appreciate the power and flexibility of dynamic expressions in CFML, you must first understand what dynamic expressions are and how are they used.

String expressions

Central to the notion of dynamic expression evaluation is the concept of a string expression. A string expression is nothing more than a CFML expression inside a string, for example, “1+1”.

Any CFML expression can be converted to a string expression by following these simple steps:

- Start with the expression text.
- Escape any double quotes in it.
- Put double quotes around it.

The following table shows several examples of the process:

Sample Conversions to String Expressions

Step 1	Step 2	Step 3
`2`	`2`	`"2"`
`2 * (11 MOD 3)`	`2 * (11 MOD 3)`	`"2 * (11 MOD 3)"`
`Form.MyFormVariable`	`Form.MyFormVariable`	`"Form.MyFormVariable"`
`Min(X, Y)`	`Min(X, Y)`	`"Min(X, Y)"`
`"Some text"`	`""Some text""`	`"""Some text"""`
`"A double quote """`	`""A double quote """"""`	`"""A double quote """""""`

The last two examples demonstrate one of the difficulties with string expressions — the proliferation of quotes. In general, if some expression text has N double quote characters in it, its string expression equivalent will have 2*N+2 double quotes in it (every double quote must be escaped and two more are added one on either end of the expression text). In the last example, the expression text has 4 double quotes to start with, therefore its string expression equivalent has 10!

This explosion of double quotes makes string expressions harder to read. There are two things you can do to improve the situation. The simplest thing to do is to mix single and double quotes. The last two examples from above then could take the following form:

Simplifying String Expressions		
Step 1	**Step 2**	**Step 3**
`'Some text'`	`'Some text'`	`"'Some text'"`
`"Some text"`	`"Some text"`	`'"Some text"'`
`'A double quote "'`	`'A double quote ""'`	`"'A double quote ""'"`

You can also use the DE (Delay Evaluation) function to convert strings to string expressions. It has a specific use within dynamic expression evaluation, but it comes in handy for this particular task. Start by storing the text of the expression you want to convert to a string expression in a variable. Then apply the DE function to that variable to get the desired result.

For example, the following two pieces of CFML code are equivalent:

```
<CFSET TheStringExpression = "'A double quote ""'">

<CFSET TheExpression = 'A double quote "'>
<CFSET TheStringExpression = DE(TheExpression)>
```

See the *CFML Language Reference* for details on the DE function.

Evaluating String Expressions

String expressions (which are often called dynamic expressions since their structure can easily change) can be evaluated using the Evaluate function. The Evaluate function takes a string expression, evaluates it, and returns the result. When ColdFusion is asked to evaluate the string expression "1+1", it produces the expected result of 2.

Examples

The following table illustrates the use of the Evaluate function:

Using the Evaluate Function	
CFML expression	**Result**
`Evaluate("2")`	2
`Evaluate("2 * (11 MOD 3)")`	4
`Evaluate("Form.MyFormVariable")`	the value of Form.MyFormVariable

Using the Evaluate Function (Continued)

CFML expression	Result
`Evaluate("Min(X, Y)")`	the smaller of the values of X and Y
`Evaluate(DE("Some text"))`	Some text
`Evaluate(DE('A double quote "'))`	A double quote "

The guidelines for calling the Evaluate function are shown below:

Guidelines for Calling the Evaluate Function

To get	Use
the result of (1 + A)	`Evaluate("1 + A")`
the value of variable A	`Evaluate("A")`
the number 2	`Evaluate("2")`
the string A	`Evaluate(DE("A"))`

Note the difference between the second and the fourth example. In the second example, we want to evaluate the expression A which should resolve to the value of the variable A (or an error if A does not exist). In the last example, we want to evaluate the expression “A” which should resolve to the text inside the string, or simply A. To inform ColdFusion of the difference (without worrying about escaping quotes), you call the DE function.

Pound Signs Inside String Expressions

Developers must be careful when using pound signs inside string expressions. Consider the following example as a guideline on when to use pound signs inside string expressions:

```
<CFSET A=2>
<CFSET Expression1="1 + #A#">
<CFSET Expression2="1 + A"> <BR>
<!--- This will produce a 3 --->

<CFOUTPUT>#Evaluate(Expression1)#</CFOUTPUT> <BR>

<!--- This will produce a 3 also --->

<CFOUTPUT>#Evaluate(Expression2)#</CFOUTPUT> <BR>
```

```
<!--- Now change the value of A --->

<CFSET A=5> <BR>

<!--- This will produce a 3 again, because Expression1
is equal to the string "1 + 2". The value of A,
which was 2 at the time Expression1 got its value
was directly inserted into the expression text. --->

<CFOUTPUT>#Evaluate(Expression1)#</CFOUTPUT> <BR>

<!--- This will produce a 6, because Expression2
is equal to "1 + A". The name, rather than the
value of the variable A was inserted into the
text of Expression2. --->

<CFOUTPUT>#Evaluate(Expression2)#</CFOUTPUT> <BR>
```

Note To build dynamic expressions with variables inside them, do not use pound signs around the variable names when you build the expression text.

Setting Variables Dynamically

In addition to evaluating expressions whose text is generated dynamically, ColdFusion allows developers to assign values to variables whose names are dynamic using the SetVariable function. SetVariable returns the value that was set. The following simple example shows how SetVariable can be used:

```
<CFSET VariableName="MyVariable">
<CFSET Value=2 ^ 10>
<CFSET ValueJustSet=SetVariable(VariableName, Value)>
```

After the third CFSET statement, the resulting value of the MyVariable variable is 1024.

Using the SetVariable and Evaluate functions together

The following example demonstrates how the SetVariable and Evaluate functions can work together. The first loop sets ten variables with names MyVar1, MyVar2, etc. to the values 100, 200, etc. up to 1000. The second loop outputs the names and values of these variables.

```
<CFLOOP INDEX="Counter" FROM="1" TO="10">

<CFSET Result=SetVariable("MyVar#Counter#", Counter*100)>

</CFLOOP>

<CFLOOP INDEX="Counter"
    FROM="1" TO="10">
    <CFOUTPUT>
    MyVar#Counter#=#Evaluate("MyVar#Counter#")#<BR>
    </CFOUTPUT>

</CFLOOP>
```

Example

The final example shows how you might implement the `<CFPARAM NAME=”VariableName” DEFAULT=”DefaultValue”>` tag in CFML:

```
<!--- Initialize the necessary variables --->

<CFSET VariableName="SomeVariable">
<CFSET DefaultValue="Default value">

<!--- Use Evaluate to call ParameterExists with
the name of the variable whose existence we
want to check for --->

<CFIF NOT Evaluate("ParameterExists(#VariableName#)")>

<!--- If the variable does not exist, use
SetVariable to set its value to the default --->

<CFSET Result=SetVariable(VariableName, DefaultValue)>

</CFIF>

<!--- Check to see that the variable
has been created --->

<CFOUTPUT>
    #VariableName#=#Evaluate(VariableName)#
</CFOUTPUT>
```

The CFML above executes as if the condition inside the CFIF tag were:

```
NOT ParameterExists(SomeVariable)
```

Advanced dynamic expressions

There are two additional dynamic expression evaluation functions, IIf and DE. For more information on these functions, refer to the Functions and Expressions chapter in this book.

See the *CFML Language Reference* for details on specific CFML functions.

CHAPTER 4

Regular Expressions

This chapter describes how regular expressions work in ColdFusion. As a prerequisite to understanding this material, a CFML developer should be thoroughly familiar with ColdFusion expressions.

See Chapter 2, “Functions and Expressions,” on page 7 for information on ColdFusion expressions. See the *CFML Language Reference* for descriptions of all ColdFusion functions.

Contents

- Regular Expressions 40
- Single-Character Regular Expressions 40
- Character Classes 41
- Multi-Character Regular Expressions 42
- Using Backreferences 42
- Returning Matched Sub-Expressions 43
- Anchoring a Regular Expression to a String 44

Regular Expressions

Most people who have worked with the UNIX operating system or have done Web development using the Perl language will be familiar with regular expressions. ColdFusion supports regular expressions in functions whose names begin with the letters "RE", for example, REFind, REReplace, REFindNoCase, and REReplaceNoCase.

About regular expressions

Regular expressions allow for very powerful and flexible string search and replace operations. In traditional search and replace operations, as in the Find and Replace functions of ColdFusion, developers must provide the exact text to be searched for.

This makes searches for dynamic data very difficult, if not impossible. For example, how can you find the first occurrence in a string of any word that consists entirely of capital letters that has spaces around it? Using regular expressions, the tasks is trivial:

```
<CFSET IndexOfOccurrence=REFind(" [A-Z]+ ",
    "Some BIG string")>
<!--- The value of IndexOfOccurrence is 5 --->
```

Web developers often have to process large amounts of dynamic textual data. Regular expressions can be invaluable to the developer writing complex ColdFusion applications.

Use the case-insensitive functions, REFindNoCase and REReplaceNoCase, for expressions where the search string is likely to be mixed case.

Single-Character Regular Expressions

This section describes the rules for creating regular expressions (REs). Regular expressions can be used to match complex string patterns.

The following rules determine one-character REs that match a single character:

- Special characters are: `+ * ? . [ ^ $ ( ) { | \`
- Any character that is not a special character 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, for example, “.umpty” matches either “Humpty” or “Dumpty.”
- A set of characters enclosed in brackets ([]) is a one-character RE that matches any of the characters in that set. For example, "[akm]" matches an "a", "k", or "m".
- Any regular expression can be followed by one of the following suffixes: {m,n} forces a match of m through n (inclusive) occurrences of the preceding regular expression. The suffix {m,} forces a match of at least m occurrences of the preceding regular expression. The syntax {,n} is not allowed.

- A range of characters can be indicated with a dash. For example, "[a-z]" matches any lowercase letter. However, if the first character of the set is the caret (^), the RE 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.
- All regular expressions can be made case insensitive by substituting individual characters with character sets, for example, [Nn][Ii][Cc][Kk].

Character Classes

In ColdFusion regular expressions, you can specify a character using one of the POSIX character classes. You enclose the character class name inside two square brackets, as in this example:

```
REReplace ("Allaire's Web Site","[[:space:]]","*","ALL")
```

This code replaces all the spaces with *, producing this string:

```
Allaire's*Web*Site
```

The following table shows the POSIX character classes that ColdFusion supports.

Supported Character Classes

Character Class	Matches
alpha	Matches any letter. Same as [A-Za-z].
upper	Matches any upper-case letter. Same as [A-Z].
lower	Matches any lower-case letter. Same as [a-z].
digit	Matches any digit. Same as [0-9].
alnum	Matches any alphanumeric character. Same as [A-Za-z0-9].
xdigit	Matches any hexadecimal digit. Same as [0-9A-Fa-f].
space	Matches a tab, new line, vertical tab, form feed, carriage return, or space.
print	Matches any printable character.
punct	Matches any punctuation character, that is, one of ! ' # S % & ' ( ) * + , - . / : ; < = > ? @ [/] ^ _ { | } ~
graph	Matches any of the characters defined as a printable character except those defined to be part of the *space* character class.
cntrl	Matches any character not part of the character classes [:upper:], [:lower:], [:alpha:], [:digit:], [:punct:], [:graph:], [:print:], or [:xdigit:].

Multi-Character Regular Expressions

You can use the following rules to build a multi-character regular expressions:

- Parentheses group parts of regular expressions together into grouped sub-expressions 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 sub-expressions followed by an asterisk (*) matches zero or more occurrences of the regular expression. For example, [a-z]* matches zero or more lower-case characters.
- A one-character regular expression or grouped sub-expressions followed by a plus (+) matches one or more occurrences of the regular expression. For example, [a-z]+ matches one or more lower-case characters.
- A one-character regular expression or grouped sub-expressions followed by a question mark (?) matches zero or one occurrences of the regular expression. For example, xy?z matches either "xyz" or "xz".
- 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".
- An excellent reference on regular expressions is *Mastering Regular Expressions*, Jeffrey E. F. Friedl. O'Reilly & Associates, Inc., 1997. ISBN: 1-56592-257-3, http://www.oreilly.com.

Using Backreferences

ColdFusion Server supports backreferencing, which allows you to match text in previously matched sets of parentheses. A slash followed by a digit n (\n) is used to refer to the nth parenthesized sub-expression.

One example of how backreferencing can be used 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 in ColdFusion:

```
REReplace("There is is coffee in the the kitchen",
"([A-Za-z]+)[ ]+\1","*","ALL")
```

This code searches for words that are all letters ([A-Za-z]+) followed by one or more spaces [ ]+ followed by the first matched sub-expression in parentheses. The parser detects the two occurrences of *is* as well as the two occurrences of *the* and replaces them with an asterisk, resulting in the following text:

```
There * coffee in * kitchen
```

Using backreferences in replace strings

You can now use backreferences in replace strings. Backreferences in the replace string refer to parenthesized matched sub-expressions in the regular expression search. For example, to replace all repeated words in a text string with a single word, you can 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 in REReplace, ReturnSubExpression, 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"
```

Note To use backreferences in either the search string or the replace string, you must use parentheses around the sub-expression. Otherwise, ColdFusion throws an exception.

Returning Matched Sub-Expressions

Regular expressions in ColdFusion allow you to access matched sub-expressions using the REFind and REFindNoCase functions. If you set the fourth parameter, ReturnSubExpression, to TRUE, the function returns a CFML structure with two arrays containing the positions and lengths of the matched sub-expressions, if any.

You can find the structure's contents using the keys "pos" and "len". If there are no occurrences of the regular expression, the "pos" and the "len" arrays each contain 1 element with a value of 0.

Example

```
<CFSET subExprs=REFind("([A-Za-z]+)[ ]+\1",
    "There is is a cat in in the kitchen",1,"TRUE")>

<CFSET posarray = subExprs.pos>

<CFSET lenarray=subExprs.len>
```

After these statements, posarray[1]=7, lenarray[1]=6, posarray[2]=7, and lenarray[2]=2.

Note that `posarray[1]` and `lenarray[1]` refer to the entire matched expression ("is is"), while `posarray[2]` and `lenarray[2]` refer to the first parenthesized sub-expression. This is always the case, because the complete matched expression is returned in the first element and the parenthesized elements are returned sequentially from indices 2 onwards. `Posarray[1]` and `lenarray[1]` are both 0 if there are no matches.

Anchoring a Regular Expression to a String

All or part of a regular expression can be anchored to either the beginning or end of the string being searched:

- If the caret (^) is at the beginning of a (sub)expression, the matched string must be at the beginning of the string being searched.
- If the dollar sign ($) is at the end of a (sub)expression, the matched string must be at the end of the string being searched.

Regular expression examples

The following examples show some regular expressions and describe what they match.

Regular Expression Examples	
Expression	**Description**
`[\?&]value=`	A URL parameter value in a URL.
`[A-Z]:(\\[A-Z0-9_]+)+`	An uppercase DOS/Windows full path that (a) is not the root of a drive, and (b) has only letters, numbers, and underscores in its text.
`[A-Za-z][A-Za-z0-9_]*`	A ColdFusion variable with no qualifier.
`([A-Za-z][A-Za-z0-9_]*)(\.[A-Za-z][A-Za-z0-9_]*)?`	A ColdFusion variable with no more than one qualifier, for example, Form.VarName, but not Form.Image.VarName.
`(\+|-)?[1-9][0-9]*`	An integer that does not begin with a zero and has an optional sign.
`(\+|-)?[1-9][0-9]*(\.[0-9]*)?`	A real number.
`(\+|-)?[1-9]\.[0-9]*E(\+|-)?[0-9]+`	A real number in engineering notation.

Regular Expression Examples (Continued)	
Expression	**Description**
`a{2,4}`	Two to four occurrences of 'a': aa, aaa, aaaa.
`(ba){3,}`	At least three 'ba' pairs: bababa, babababa, ...

Regular expressions in CFML

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

Examples of Regular Expression Functions	
Expression	**Description**
`REReplace (CGI.Query_String, "CFID=[0-9]+[&]*", "")`	Returns the query string with parameter CFID and its numeric value stripped out.
`REReplace("I Love Jellies", "[[:lower:]]","x","ALL"`	I Lxxx Jxxxxxx
`REReplaceNoCase("cabaret","[A-Z]", "G","ALL")`	GGGGGGG
`REReplace (Report, "\$[0-9,]*\.[0-9]*", "$***.**")", "")`	Returns the string value of the variable Report with all positive numbers in the dollar format changed to "$***.**".
`REFind ("[Uu]\.?[Ss]\.?[Aa}\.?", Report )`	Finds the position of the first occurrence of the abbreviation USA in the variable Report.
`REFindNoCase("a+c","ABCAACCDD")`	4
`REReplace("There is is coffee in the the kitchen","([A-Za-z]+) [ ]+\1","*","ALL")`	There * coffee in * kitchen
`REReplace(report, "<[^>]*>", "", "All")`	Removes all HTML tags from a string value of the report variable.

CHAPTER 5

Working with Arrays

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

Contents

- About Arrays 48
- Multidimensional Arrays 50
- Basic Array Techniques 50
- Referencing Elements in Dynamic Arrays 51
- Populating Arrays with Data 52
- Using Nested Loops for 2D and 3D Arrays 53
- Populating an Array from a Query 54
- Array Functions 56

About Arrays

If you've worked with arrays in the C programming language, or even read about working with them in C, you might regard an array as a tabular structure used to hold data, much like a spreadsheet table with clearly defined limits and dimension. A 2-dimensional (2D) array would be like a simple table; a 3-dimensional array would be like a cube made up of individual cells.

ColdFusion arrays aren't quite that simple because they are dynamic. So in a 2D array, for example, you might have what you could think of as columns of differing lengths based on the data that has been added or removed, whereas in a conventional array, array size is constant and symmetrical.

Creating an array

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

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

where *x* is the number of dimensions (from 1 to 3) in the array you want to create. You can visualize a one-dimensional (1D) array as a string of cells, like a single row from a table.

Once created, you can add data to the array, in this case using a form variable:

```
<CFSET mynewarray[3]=Form.emailaddress>
```

Data in an array is referenced by index number, in the following manner:

```
#My1DArray[index1]#<BR>
#My2DArray[index1][index2]#<BR>
#My3DArray[index1][index2][index3]#
```

Array terms

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

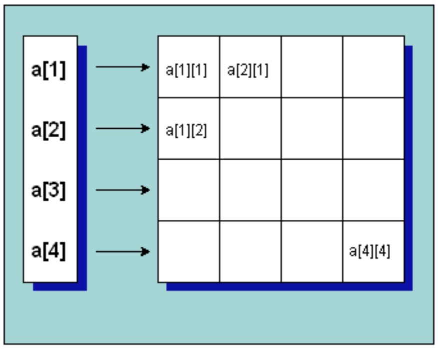
- Array dimension – The relative complexity of the array structure.
- Index – The position of an element in a dimension, ordinarily surrounded by square brackets: my1Darray[1], my2Darray[1][1], my3Darray[1][1][1].
- Array element – Data stored in an array index.

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".'

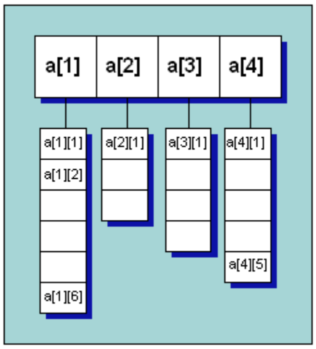
Dynamic arrays

Dynamic arrays expand to accept data you add to them and contract as you remove data from them. This diagram shows the difference between a static 2D array and a ColdFusion dynamic 2D array.

Conventional fixed-size 2D array



ColdFusion dynamic 2D array



A ColdFusion 2D array is actually a 1D array that contains a series of additional 1D arrays. Each of the arrays that make up a column can expand and contract independently of any other column.

Multidimensional Arrays

ColdFusion supports dynamic multidimensional arrays. When you declare an array with the ArrayNew function, you can specify up to three dimensions. However, if you're feeling particularly adventurous, you can increase an array's dimensions by nesting arrays as array elements:

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

<CFSET biggerarray[1][1][1]=myarray>
<CFSET biggerarray[1][1][1][10]=some_value>
<CFSET biggerarray[2][1][1]=myotherarray>
<CFSET biggerarray[2][1][1][4][2]=some_value>

<CFSET biggestarray=ArrayNew(3)>
<CFSET biggestarray[3][1][1]=biggerarray>
<CFSET biggestarray[2][1][1][2][3][1]=some_value>
```

Basic Array Techniques

To use arrays in ColdFusion, as in other languages, you need to first declare the array, specifying its dimension. Once it's declared, you can add array elements, which you can then reference by index.

As an example, say you declare a one-dimensional array called "firstname":

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

At first, the array firstname holds no data and is of an unspecified length. Now you want to add data to the array:

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

Once you've added these names to the array, it has a length of 3:

```
<CFSET temp=ArrayLen(firstname)>
<!--- temp=3 --->
```

If you remove data from an index, the array resizes dynamically:

```
<CFSET temp=ArrayDeleteAt(firstname, 2)>
<!--- "Charlie" has been removed from the array --->

<CFOUTPUT>
    The firstname array is #ArrayLen(firstname)#
    indexes in length
</CFOUTPUT>

<!--- Now the array has a length of 2, not 3 --->
```

The array now contains:

```
firstname[1]=Coleman
firstname[2]=Dexter
```

Adding elements to an array

You can add elements to an array by simply defining the value of an array element:

```
<CFSET myarray[1]=form.variable>
```

But you can also employ a number of array functions to add data to an array. You can use ArrayAppend to create a new array index at the end of the array. You can use ArrayPrepend to create a new array index at the beginning of the array. You can also use ArrayInsertAt to insert an array index and data. When you insert an array index with ArrayInsertAt, as with ArrayDeleteAt, all indexes to the right of the new index are recalculated to reflect the new index count.

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

Shifting indexes in a dynamic array

Because ColdFusion arrays are dynamic, be careful when referencing array indexes. If you add or delete an element from the middle of an array, subsequent index positions all change.

When an array index is deleted, index positions in the array are recalculated. For example, in a 1D array containing the months of the year, deleting index position [5] removes the entry for May. If you then want to delete the entry for November, you delete index position [10], not [11], since the index positions were recalculated after index position [5] was removed.

Referencing Elements in Dynamic Arrays

Unlike the C language, in ColdFusion, array indexes are counted starting with position 1. In C, array indexes start with zero, so position 1 would be referenced as firstname[0]. In ColdFusion, position one is referenced as firstname[1].

Let's add to the current firstname array example. For 2D arrays, you reference an index by specifying two coordinates: myarray[1][1].

```
<!--- This example adds a 1D array to a 1D array --->

<CFSET firstname=ArrayNew(1)>

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

<!--- First, declare the array --->
```

```
<CFSET fullname=ArrayNew(1)>

<!--- Then, add the firstname array to
index 1 of the fullname array --->

<CFSET fullname[1]=firstname>

<!--- Now we’ll add the last names for symmetry --->

<CFSET fullname[2][1]="Hawkins">
<CFSET fullname[2][2]="Parker">
<CFSET fullname[2][3]="Gordon">

<CFOUTPUT>
    #fullname[1][1]# #fullname[2][1]#<BR>
    #fullname[1][2]# #fullname[2][2]#<BR>
    #fullname[1][3]# #fullname[2][3]#<BR>
</CFOUTPUT>
```

Additional referencing methods

You can reference array indexes in the standard way: myarray[*x*] where *x* is the index you want to reference. You can also use ColdFusion expressions inside the square brackets to reference an index. The following are valid ways of referencing an array index:

```
<CFSET myarray[1]=expression>
<CFSET myarray[1 + 1]=expression>
<CFSET myarray[arrayindex]=expression>
```

Calculating an array index

As described earlier in Shifting indexes in a dynamic array, array indexes are recalculated whenever data is added, removed, appended, or prepended to an array. Keep this in mind when building an array that will be manipulated in any of these ways.

Populating Arrays with Data

One-dimensional arrays can store any values, including queries 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 addition, several basic techniques are important to master:

- Populating an array with ArraySet
- Populating an array with CFLOOP
- Populating an array from a query

Populating an array with ArraySet

You can use the ArraySet function to populate a 1D array, or one dimension of a multi-dimensional array, with some initial value such as an empty string or 0 (zero). This can be useful if you need to create an array of a certain size, but don't need to add data to it right away. Array indexes need to contain some value, such as an empty string, in order to be referenced.

Use ArraySet to initialize all elements of an array to some value:

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

This example initializes the array myarray, indexes 1 to 100, with an empty string.

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

Populating an array with CFLOOP

A common and very efficient method for populating an array is by creating a looping structure that adds data to an array based on some condition using CFLOOP.

In the following example, a simple one-dimensional array is populated with the names of the months using a CFLOOP. A second CFLOOP is used to output 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 need to employ nested loops to return array data. With a 1D array, a single CFLOOP is sufficient to output data, as in the example just above. With arrays of dimension greater than one, you need to maintain separate loop counters for each array level.

Nesting CFLOOPs for a 2D array

The following example shows how to handle nested CFLOOPs to output data from a 2D array:

```
<P>The values in my2darray are currently:

<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 CFLOOPs for a 3D array

For 3D arrays, you simply nest an additional CFLOOP:

```
<P>My3darray’s values are currently:

<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:

- Query data cannot be added to an array all at once. A looping structure is generally required to populate an array from a query.
- Query column data can be referenced using array-like syntax. For example, myquery.col_name[1] references data in the first row in the column col_name.

You can use a CFSET tag to define values for array indexes, as in the following example:

```
<CFSET arrayname[x]=queryname.column[row]>
```

In the following example, a CFLOOP is used to place four columns of data from a sample data source into an array, "myarray."

```
<!--- Do the query --->

<CFQUERY NAME="test" DATASOURCE="cfsnippets">
    SELECT EMPLOYEE_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]=test.employee_id[CurrentRow]>
<CFSET myarray[CurrentRow][2]=test.LASTNAME[CurrentRow]>
<CFSET myarray[CurrentRow][3]=test.FIRSTNAME[CurrentRow]>
<CFSET myarray[CurrentRow][4]=test.EMAIL[CurrentRow]>

</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>
```

Array Functions

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

Array Functions	
Function	**Description**
ArrayAppend	Appends an array index 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 data from a specified array at the specified index.
ArrayInsertAt	Inserts data in a specified array at the specified index.
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 one dimensional 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 the Array Functions section of the *CFML Language Reference*.

CHAPTER 6

Working with Structures

ColdFusion supports structures for managing lists of key-value pairs. This section explains the basics of creating and working with structures.

Contents

- About Structures 58
- Creating and Using Structures 58
- Structure Example 61
- Using Structures as Associative Arrays 63
- Structure Functions 64

About Structures

ColdFusion supports the creation and handling of Structures, which enable developers to create and maintain key-value pairs. A structure lets you build a collection of related variables that are grouped under a single name. Structures are also known as associative arrays. You can define ColdFusion structures dynamically.

You can use structures to refer to related string 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 number, etc. Then you can refer to this collection of information as a structure called *employee* rather than as a collection of individual variables.

Structure notation

Developers can use three types of notation for structures:

Objects.property

You can use the *object.property* notation to refer to values in a structure. So a property, *prop*, of an object, *obj*, can be referred to as *obj.prop*. This notation is useful for simple assignments, as in this example:

```
depts.John="Sales"
```

Use this notation only when the property names (keys) are known in advance and they are strings, with no special characters, numbers, or spaces. You cannot use the dot notation when the property, or key, is dynamic.

Associative arrays

If the key name is not known in advance, or contains spaces, numbers or special characters, you can use associative array notation. This uses structures as arrays with string indexes, for example, depts[“John”] or depts[“John Doe”]=”Sales”.

See Using Structures as Associative Arrays for more information.

Structure functions

The Structure functions should be used when the simpler syntax styles described above cannot be used, for example when dynamic keys are required. The sections in this chapter describe how to use the Structure functions.

Creating and Using Structures

This section explains how to use the structure functions to create and use structures in ColdFusion. We use as our example a sample structure called *employee,* which is used to add new employees to a corporate information system.

Creating structures

You create structures by assigning a variable name to the structure with the StructNew function:

```
<CFSET mystructure=StructNew()>
```

For example, to create a structure named *employee*, use this syntax:

```
<CFSET employee=StructNew()>
```

Now the structure exists and you can add data to it.

Adding data to structures

After you’ve created a structure, you add key-value pairs to the structure using the StructInsert function:

```
<CFSET value=StructInsert(structure_name, key, value [, AllowOverwrite])>
```

The AllowOverwrite parameter is optional and can be either TRUE or FALSE. It can be used to specify whether an existing key should be overwritten or not. The default is FALSE.

When adding string values to a structure, enclose the string in quotation marks. For example, to add a key, *John*, with a value, *Sales*, to an existing structure called *Departments*, use this syntax:

```
<CFSET value=StructInsert(Departments, “John”, “Sales”)>
```

To change the value associated with a specific key, use the StructUpdate function. For example, if John moves from the Sales department to the Marketing department, you would use this syntax to update the Departments associative array:

```
<CFOUTPUT>
Personnel moves: #StructUpdate(Departments, “John”, “Marketing”)#
</CFOUTPUT>
```

Example of adding data to a structure

The following example shows how to add content to a sample structure named *employee*, building the content of the value fields dynamically using form variables:

```
<CFSET rc=StructInsert(employee, "firstname", "#FORM.firstname#")>
<CFSET rc=StructInsert(employee, "lastname", "#FORM.lastname#")>
<CFSET rc=StructInsert(employee, "email", "#FORM.email#")>
<CFSET rc=StructInsert(employee, "phone", "#FORM.phone#")>
<CFSET rc=StructInsert(employee, "department", "#FORM.department#")>
```

Finding information in Structures

To find the value associated with a specific key, use the StructFind function:

```
StructFind(structure_name, key)
```

Example

The following example shows how to generate a list of keys defined for a structure.

```
<CFLOOP COLLECTION=#department# item=person>
    <CFOUTPUT>
    Key - #person#<BR>
    Value - #StructFind(department,person)#<BR>
    </CFOUTPUT>
```

Note that the StructFind function is case-insensitive. When you enumerate key-value pairs using a loop, the keys appear in upper-case.

Getting information about structures

To find out if a given value represents a structure, use the IsStruct function:

`IsStruct(`*`variable`*`)`

This function returns TRUE if *variable* is a structure.

Structures are not indexed numerically, so to find out how many name-value pairs exist in a structure, use the StructCount function, as in this example:

```
StructCount(employee)
```

To discover whether a specific Structure contains data, use the StructIsEmpty function:

```
StructIsEmpty(structure_name)
```

This function returns TRUE if the structure is empty and FALSE if it contains data.

Finding a specific key

To learn whether a specific key exists in a structure, use the StructKeyExists function.

```
StructKeyExists(structure_name, key)
```

If the name of the key is known in advance, you can use the ColdFusion function IsDefined, as in this example:

`<CFSET temp=IsDefined(“`*`structure_name.key`*`“)>`

But if the key is dynamic, or contains special characters, you must use the StructKeyExists function:

```
<CFSET temp=StructKeyExists(structure_name, key)>
```

Copying structures

To copy a structure, use the StructCopy function. This function takes the name of the structure you want to copy and returns a new structure with all the keys and values of the named structure.

```
StructCopy(structure)
```

This function throws an exception if *structure* doesn’t exist.

Use the StructCopy function when you want to create a physical copy of a structure. You can also use assignment to create a copy by reference.

Deleting structures

To delete an individual name-value pair in a structure, use the StructDelete function:

```
StructDelete(structure_name, key)
```

This deletes the named key and its associated value.

You can also use the StructClear function, to delete all the data in a structure but keep the structure instance itself:

```
StructClear(structure_name)
```

Structure Example

Structures are particularly useful for grouping together a set of variables under a single name. In the following example files, structures are used to collect information from a form, `structure.cfm`, and submit that information to a custom tag at `addemployee.cfm`.

These example files show how you can use a structure to pass information to a custom tag, named CF_ADDEMPLOYEE.

Example file structure.cfm

```
<!--- This example shows how to use the StructInsert
      function. It calls the CF_ADDEMPLOYEE custom tag,
      which uses the addemployee.cfm file. --->
<HTML>
<HEAD>
<TITLE>Add New Employees</TITLE>
</HEAD>

<BODY>
<H1>Add New Employees</H1>

<!--- Establish parms 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 all form fields are passed, create structure
    named employee and add values --->

<CFIF #FORM.FIRSTNAME# EQ "">
 <P>Please fill out the form.
```

```
<CFELSE>
  <CFOUTPUT>
   <CFSCRIPT>
     employee=StructNew();
     StructInsert(employee, "firstname", "#FORM.firstname#");
     StructInsert(employee, "lastname", "#FORM.lastname#");
     StructInsert(employee, "email", "#FORM.email#");
     StructInsert(employee, "phone", "#FORM.phone#");
     StructInsert(employee, "department", "#FORM.department#");
  </CFSCRIPT>

  <P>First name is #StructFind(employee, "firstname")#</P>
  <P>Last name is #StructFind(employee, "lastname")#</P>
  <P>EMail is #StructFind(employee, "email")#</P>
  <P>Phone is #StructFind(employee, "phone")#</P>
  <P>Department is #StructFind(employee, "department")#</P>
  </CFOUTPUT>

  <!--- Call the custom tag that adds employees --->

  <CF_ADDEMPLOYEE EMPINFO="#employee#">
</CFIF>

<HR>
<FORM ACTION="structinsert.cfm" METHOD="Post">
<P>First Name:&nbsp;
<INPUT NAME="firstname" TYPE="text" HSPACE="30" MAXLENGTH="30">
<P>Last Name:&nbsp;
<INPUT NAME="lastname" TYPE="text" HSPACE="30" MAXLENGTH="30">
<P>EMail:&nbsp;
<INPUT NAME="email" TYPE="text" HSPACE="30" MAXLENGTH="30">
<P>Phone:&nbsp;
<INPUT NAME="phone" TYPE="text" HSPACE="20" MAXLENGTH="20">
<P>Department:&nbsp;
<INPUT NAME="department" TYPE="text" HSPACE="30" MAXLENGTH="30">

<P>
<INPUT TYPE="Submit" VALUE="OK">
</FORM>

</BODY>
</HTML>
```

Example file addemployee.cfm

```
<P>This file is an example of a custom tag used
to add employees. Employee information is passed
through the employee structure (the EMPINFO attribute).
In UNIX, you must also add the Emp_ID.

<CFSWITCH EXPRESSION="#ThisTag.ExecutionMode#">
    <CFCASE VALUE="start">
        <CFIF StructIsEmpty(attributes.EMPINFO)>
        <CFOUTPUT>Error. No employee data was passed.</CFOUTPUT>
            <CFEXIT METHOD="ExitTag">
        <CFELSE>
        <!--- Add the employee --->
        <!--- In UNIX, you must also add the Emp_ID --->

        <CFQUERY NAME="AddEmployee" DATASOURCE="cfsnippets">
            INSERT INTO Employees
            (FirstName, LastName, Email, Phone, Department)
            VALUES
            <CFOUTPUT>
            (
            '#StructFind(attributes.EMPINFO, "firstname")#' ,
            '#StructFind(attributes.EMPINFO, "lastname")#' ,
            '#StructFind(attributes.EMPINFO, "email")#' ,
            '#StructFind(attributes.EMPINFO, "phone")#' ,
            '#StructFind(attributes.EMPINFO, "department")#'
             )
            </CFOUTPUT>
        </CFQUERY>
        </CFIF>
    <CFOUTPUT><HR>Employee Add Complete</CFOUTPUT>
    </CFCASE>
</CFSWITCH>
```

Using Structures as Associative Arrays

You can also use structures as associative arrays. Structures index data by string keys rather than by integers.

You might use structures to create an associative array that matches people's names with their departments. In this example, a structure named *Departments* includes an employee named John, listed in the Sales department. To access John's department, you would use the syntax, `Departments["John"]`.

A structure's key must be a string. The values associated with the key can be anything:

- a string
- an integer
- an array
- another structure

Looping through structures

The following example shows how you can loop through a structure to output its contents. Note that when you enumerate key-value pairs using a loop, the keys appear in upper-case.

```
<!--- Create a structure and loop through 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>Dept.</B></TD>
    </TR>

<!--- In CFLOOP, use ITEM to create a variable
    called person to hold value of key as loop runs --->

<CFLOOP COLLECTION=#Departments# ITEM=”person”>

    <TR>
    <TD>#person#</TD>
    <TD>#Departments[person]#</TD>
    </TR>

</CFLOOP>

</TABLE>
</CFOUTPUT>
```

Structure Functions

There are several new functions that help you create and manage structures in ColdFusion applications.

Structure Functions

Function	Description
IsStruct	Returns TRUE if the specified variable is a structure.
StructClear	Removes all data from the specified structure.
StructCopy	Returns a new structure with all the keys and values of the specified structure.
StructCount	Returns the number of keys in the specified structure.

Structure Functions	
Function	**Description**
StructDelete	Removes the specified item from the specified structure.
StructFind	Returns the value associated with the specified key in the specified structure.
StructInsert	Inserts the specified key-value pair into the specified structure.
StructIsEmpty	Indicates whether the specified structure contains data. Returns TRUE if the structure contains no data, and FALSE if it does contain data.
StructKeyExists	Returns TRUE if the specified key is in the specified structure.
StructNew	Returns a new structure.
StructUpdate	Updates the specified key with the specified value.

Note that in all cases, except StructDelete, an exception will be thrown if the referenced key or structure does not exist.

For more information on these functions, see the *CFML Language Reference*.

CHAPTER 7

Exchanging Data via XML

You can now move complex CFML data structures across the Web using Web Distributed Data Exchange (WDDX). This new capability is based on XML 1.0 and is used to exchange data between CFML applications and other applications.

Additionally, CFML data structures can be instantiated as WDDX elements for access by JavaScript statements on the browser.

This functionality is encapsulated in the CFWDDX tag.

Contents

- An Overview of Distributed Data for the Web.. 68
- WDDX Components 68
- Working With Application-Level Data... 69
- Data Exchange Across Application Servers... 69
- How WDDX Works 69
- Converting CFML Data to a JavaScript Object.. 70
- Transferring Data From Browser to Server... 72

An Overview of Distributed Data for the Web

Web Distributed Data Exchange (WDDX) is an Extensible Markup Language (XML) vocabulary for describing complex data structures such as arrays, associative arrays, and recordsets in a generic fashion so they can be moved between different application server platforms and between application servers and browsers using only HTTP. Target platforms for WDDX include ColdFusion, Active Server Pages, JavaScript, and Perl.

Unlike other approaches to creating XML-based generic distributed object systems for the Web, WDDX is not designed as an analog of traditional object programming languages. These approaches use XML as a generic descriptor for initiating remote procedure calls between different object frameworks. This is a valuable approach to the problem of using traditional object-based applications to the Internet, but it is more useful as a bridge between different programming paradigms than it is as a Web-native methodology for distributing structured data between application frameworks.

There are several problems with merging the distributed object model of computing with the Internet. Primarily, this model was designed with a completely different vision of what general internetworking would look like. Instead of the "dumb and disconnected" model of HTTP, distributed computing was built on the assumption of rich network services that would allow resources on remote machines to act like local components. These services allow an application on one system to find, invoke, and maintain state with objects on a remote system. Communication between objects on remote systems uses an efficient, special-purpose wire protocol.

But these services are a barrier to development in the disconnected world. At the most fundamental level, the wire protocols of Distributed COM and CORBA are blocked by most Web firewall software. But the largest barrier is that client-server oriented distributed computing frameworks impose a development methodology that is radically different from that of the Web. This methodology excludes the vast majority of developers building Web applications whose main tools are tag-based markup languages and scripting. While WDDX will work with systems that support component object development paradigms, there is a large set of applications that can benefit from the general characteristics of a distributed data system without the client-server overhead.

A business scenario for using ColdFusion's XML implementation is available at http://www.microsoft.com/xml/scenario/allaire.asp.

WDDX Components

The core of WDDX is the XML vocabulary, and a set of components for each of the target platforms to serialize and de-serialize data into the appropriate data structure and a document type definition (DTD) that describes the structure of standard data types. Functionally, this 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.

The first version of WDDX is based on XML 1.0, which is a W3C Recommendation. Other W3C efforts now in the works will have obvious application to WDDX when they are completed, including the XML-Data proposal and metadata formats such as the Resource Description Framework (RDF). The WDDX DTD supports versioning, allowing these and other enhancements to be folded into the specification as they become available without disrupting working applications.

Working With Application-Level Data

The real strength of WDDX is clear if the client and server are seen as a unified platform for applications. This is a subtle, but profound, distinction from the traditional view of an application where services are partitioned between the client and server.

In client-server, a client might query a database and get a recordset that can be browsed, updated and returned to the server without requiring a persistent connection. In this scenario, data is highly-structured and that structure is baked into the client side of the application ahead of time.

While this style of databinding relies on the presence of data sources that expose well-structured data of known types, WDDX is designed to transport application-level data structures to facilitate seamless computing between the client and the server side of a web application. Application-level data structures generally differ from data exposed via traditional data sources, e.g., databases. They are generally more complex and ad hoc, with dynamic structure. WDDX allows developers to work with this data without the overhead of setting up a datasource for every type of data needed. Therefore, it integrates nicely with and complements other approaches that rely on existing data sources.

Data Exchange Across Application Servers

The other common use of WDDX is expected to be sending complex, structured data seamlessly between different application server platforms. This will allow an application based on ColdFusion at one business to send a purchase order, for instance, to a supplier running a CGI-based system. The supplier could then extract information from the order and pass it to a shipping company running an application based on ASP. Unlike traditional client-server approaches (including distributed object systems) minimal to no prior knowledge of the source or target systems is required by any of the others.

How WDDX Works

The WDDX vocabulary describes a data object with a high level of abstraction. For instance, a simple object with two string properties might take the following form after it is serialized into a WDDX XML representation for delivery via HTTP:

```
<var name='x'>

<struct>
<var name='a'>

    <string>Property a</string>
    </var>
    <var name='b'>
    <string>Property b</string>
    </var>

</struct>
</var>
```

The deserialization of this XML by the WDDX Serializer object would create a structure similar to what would be created directly by this JavaScript object declaration:

```
x = new Object();
x.a = "Property a";
x.b = "Property b";
```

See the *CFML Language Reference* for more information on JavaScript objects.

Converting CFML Data to a JavaScript Object

The following example demonstrates the transfer of a CFQUERY result set from a CFML template executing on the server to a JavaScript object that is processed by the browser.

The application consists of five principal sections:

- Running a data query
- Calling the WDDX JavaScript utility
- Specifying the conversion type and the input and output variables
- Calling the conversion function
- Outputting the object data in HTML

This example uses a registered ColdFusion 4.0 datasource and can be run from ColdFusion Server.

```
<!--- Create a simple query  --->
<CFQUERY NAME = 'q' DATASOURCE ='snippets'>
    SELECT Message_Id, Thread_id,
    Username, Posted from messages
</CFQUERY>

<script language=javascript>

<!--- Bring in WDDX JS support objects
     A <script src=></script> can be used instead
     wddx.js is part of the ColdFusion distribution --->
    <CFINCLUDE template='/CFIDE/scripts/wddx.js'>
    <!--- Use WDDX to move from CFML data to JS --->
    <CFWDDX ACTION='cfml2js' input=#q# topLevelVariable='q'>

    <!--- Recordset dumping routine --->
    function dumpWddxRecordset(r)
    {
    // Get row count
    nRows = r.getRowCount();

    // Determine column names
    colNames = new Array();
    i = 0;
    for (col in r)
    {
        if (typeof(r[col]) == "object")
        {
            colNames[i++] = col;
        }
    }

    // Dump the recordset data

    o = "Dumping recordset...<p>";

    o += "<table cellpadding=3pt><tr>";
    for (i = 0; i < colNames.length; ++i)
    {
        o += "<td>" + colNames[i] + "</td>";
    }
    o += "</tr>";

    for (row = 0; row < nRows; ++row)
    {
        o += "<tr>";
        for (i = 0; i < colNames.length; ++i)
        {
            o += "<td>" + r.getField(row, colNames[i]) + "</td>";
        }
        o += "</tr>";
    }
        o += "</table>";

    // Write the table to the HTML stream

    document.write(o);
    }

    <!--- Dump the recordset --->
    dumpWddxRecordset(q);

</script>
```

Transferring Data From Browser to Server

This example serializes form field data, posts it to the server, deserializes it, and outputs the data. For simplicity, only a small amount of data is collected. In applications where complex JavaScript data collections are generated, this basic approach can be extended very effectively.

```
<!--- Get WDDX JS utility objects --->
<script language="JavaScript"
    src="/CFIDE/scripts/wddx.js"></script>

<!--- Add 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
    var personInfo = new WddxRecordset(new Array("firstName",
    "lastName"));

    // Add next record
    function doNext()
    {
        nRows = personInfo.getRowCount();
        personInfo.firstName[nRows] =
        document.personForm.firstName.value;
        personInfo.lastName[nRows] = document.personForm.lastName.value;
        document.personForm.firstName.value = "";
        document.personForm.lastName.value = "";
    }

</script>

<!--- Data collection form --->
<form action="wddx_browser_2_server.cfm" method="post"
name="personForm">

    <!--- Input fields --->
    Personal information<p>
    First name: <input type=text name=firstName><br>
    Last name: <input type=text name=lastName><br>
    <p>

    <!--- 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">
    <p>

    <!--- This is where the WDDX packet will be stored --->
    WDDX packet display:<p>
    <textarea name="wddxPacket" rows="10" cols="80" wrap="Virtual"><
    /textarea>

</form>

<!--- Server-side processing --->
<hr>
<p><b>Server-side processing</b><p>
<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:<p>
        <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>
```

CHAPTER 8

Using CFML Scripting

ColdFusion now offers a server-side scripting language, CFScript, that provides ColdFusion functionality in script syntax. This JavaScript-like language gives developers the same control flow, but without tags.

This chapter describes the CFScript language's functionality and syntax.

Contents

- About CFScript 76
- The CFScript Language........ 77
- Interaction of CFScript with CFML........ 81

About CFScript

ColdFusion now has a server-side scripting language, CFScript, that offers ColdFusion functionality in script syntax.

This JavaScript-like language offers the same control flow, but without tags. CFScript regions are bounded by <CFSCRIPT> and </CFSCRIPT>. You can use ColdFusion expressions, but not CFML tags, inside a CFScript region.

See Chapter 2, "Functions and Expressions," on page 7 for more on CFML expressions.

CFScript example

The following example shows how a block of CFSET tags can be rewritten in CFScript:

Using CFML tags

```
<CFSET employee=StructNew()>
    <CFSET employee.firstname=FORM.firstname>
    <CFSET employee.lastname=FORM.lastname>
    <CFSET employee.email=FORM.email>
    <CFSET employee.phone=FORM.phone>
    <CFSET employee.department=FORM.department>
<CFOUTPUT>About to add #FORM.firstname# #FORM.lastname#
</CFOUTPUT>
```

Using CFScript

```
<CFSCRIPT>
    employee=StructNew();
    employee.firstname=FORM.firstname;
    employee.lastname=FORM.lastname;
    employee.email=FORM.email;
    employee.phone=FORM.phone;
    employee.department=FORM.department;
    WriteOutput("About to add " & FORM.firstname & " " &
FORM.lastname);
</CFSCRIPT>
```

The WriteOutput function appends text to the page output stream. Although you can call this function anywhere within a page, it is most useful inside a CFSCRIPT block. See the *CFML Language Reference* for information on the WriteOutput function.

Supported statements

CFScript supports the following statements:

- if-else
- while
- do-while

- for
- break
- continue
- for-in
- switch-case

For more information

The following JavaScript references may be useful in understanding the concepts and control flow statements in CFScript:

- Netscape's JavaScript Guide
- Netscape's JavaScript Reference
- David Flanagan's *JavaScript: The Definitive Guide*, published by O'Reilly & Associates, 1996, 1998, http://www.oreilly.com.

The CFScript Language

This section explains the syntax of the CFScript language.

Statements

Note that in CFScript semicolons define the end of a statement. Line breaks in your source are insignificant. You can enclose multiple statements in curly braces:

```
{ statement; statement; statement; }
```

The following statements are supported in CFScript:

Assignment: *lval* = *expr* ;

Note that *lval* can be a simple variable, an array reference, or a member of a structure.

```
x = "positive"; /y = x; a[3]=5;/ structure.member=10;
```

CFML expression: *expr* ;

```
StructInsert(employee,"lastname",FORM.lastname);
```

For more information on ColdFusion expressions see Chapter 2, “Functions and Expressions,” on page 7 in this book.

if-else: if(*expr) statement* [else *statement*] ;

```
if(score GT 1)
    result = "positive";
else
    result = "negative";
```

for loop: for (*init-expr* ; *test-expr* ; *final-expr) statement* ;

Note that *init-expr* and *final-expr* 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-expr* can be one of the following:

- any ColdFusion expression, for example, A LT 5, loop LE x, or Y EQ "not found" AND loop LT end
- empty

Here are some examples of *for* loops:

```
// Multiline for statement
for(Loop1=1;
    Loop1 LT 10;
    Loop1 = Loop1 + 1);
    a[loop1]=loop1;

// Complete for loop in a single line.
for(loop=0; loop LT 10; loop=loop+1)arr[loop]=loop;

// Uses braces to note the code to loop over
for( ; ; )
{
    indx=indx+1;
    if(Find("key",strings[indx],1))
        break;
}
```

while loop: while (*expr*) *statement* ;

```
// Use braces to note the code to loop over
a = ArrayNew(1);
while (loop1 LT 10)
{
 a[loop1] = loop1 + 5;
 loop1 = loop1 + 1;
}


a = ArrayNew(1);
while (loop1 LT 10)
{
    a[loop1] = loop1 + 5;
    loop1 = loop1 +1;
}
```

do-while loop: do *statement* while (*expr*) ;

```
// Complete do-while loop on a single line
a = ArrayNew(1);
do {a[loop1] = loop1 + 5; loop1 = loop1 + 1;} while (loop1 LT 10);


// Multiline do-while loop
a = ArrayNew(1);
do
{
  a[loop1] = loop1 + 5;
  loop1 = loop1 + 1;
}
while (loop1 LT 10);
```

switch-case: switch (*expr*) {case *const-expr* : *statement* break ; default : *statement* }

In this syntax, *const-expr* must be a constant (i.e., not a variable, a function, or other expression). Only one default statement is allowed. There can be multiple case statements. You cannot mix Boolean and numeric case values in a *switch* statement.

No two constants may be the same inside a *switch* statement.

```
switch(name)
{
    case “John”:
    {
        male=true;
        found=true;
        break;
    }
    case “Mary”:
    {
        male=false;
        found=true;
        break;
    }
    default:
    {
        found=false;
        break;
    }
} //end switch
```

for-in loop: for (*variable* in *collection*) *statement* ;

Note that *variable* can be any ColdFusion identifier, and *collection* must be the name of an existing ColdFusion structure.

```
for (x in mystruct) mystruct[x]=0;
```

continue: skip to next loop iteration

```
for ( loop=1; loop LT 10; loop = loop+1)
{
    if(a[loop]=0) continue;
    a[loop]=1;
}
```

break: break out of the current switch statement or loop

```
for( ; ; )
{
    indx=indx+1;
    if(Find("key",strings[indx],1))
        break;
}
```

Expressions

CFScript supports all CFML expressions. CFML expressions include operators (such as +, -, EQ, etc.) as well as all CFML functions.

See the Functions and Expressions chapter for information about CFML operators and functions.

Note You cannot use CFML tags in CFScript.

Variables

Variables can be of any ColdFusion type, such as numbers, strings, arrays, queries, and COM objects. You can read and write variables within the script region.

Comments

Comments in CFScript blocks begin with two forward slashes (//) and end at the line end. You can also enclose CFScript comments between /* and */. Note that you cannot nest /* and */ inside other comment lines.

Differences from JavaScript

While CFScript is based on JavaScript, there are some key differences you'll want to note:

- CFScript uses ColdFusion expressions, which are neither a subset nor a superset of JavaScript expressions. For example, there is no < operator in CFScript.

- No user-defined functions or variable declarations are available.
- CFScript is case-insensitive.
- All statements end in a semi-colon, and line breaks in your code are insignificant.
- In CFScript, assignments are statements, not expressions.
- Some implicit objects are not available, such as Window and Document.

Note CFScript is not directly exportable to JavaScript. Only a limited subset of JavaScript can run inside CFScript.

Reserved words

In addition to the names of ColdFusion functions and words reserved by ColdFusion expressions (such as NOT, AND, IS, and so on), the following words are reserved in CFScript. Do not use these words as variables or identifiers in your scripting code:

- for
- while
- do
- if
- else
- switch
- case
- break
- default
- in
- continue

Interaction of CFScript with CFML

You enclose CFScript regions inside <CFSCRIPT> and </CFSCRIPT> tags. No other CFML tags are allowed inside a CFSCRIPT region.

A CFSCRIPT tag block must contain at least one CFScript statement, and comments are not considered statements. If there are no statements, you should comment out the entire CFSCRIPT block (including its enclosing <CFSCRIPT> and </CFSCRIPT> blocks) with CFML comment tags.

You can read and write ColdFusion variables inside CFScript, as shown in this example:

```
<CFOUTPUT QUERY="employees">

    <CFSCRIPT>
    //‘testres’ is a column in the "employees" query

    if( testres EQ 1 )
        result="positive";
    else
        result="negative";

    </CFSCRIPT>

<!--- The variable result takes its
    value from the script region --->

Test for #name# is #result#.

</CFOUTPUT>
```

CHAPTER 9

Structured Exception Handling

The ColdFusion Server offers a means for developers to catch and process exceptions in ColdFusion application pages, through the CFTRY, CFCATCH, and CFTHROW tags.

Contents

- Overview of Exception Handling in ColdFusion 84
- Exception-Handling Strategies 85
- Exception Handling Example.. 86
- Exception Information in CFCATCH 88

Overview of Exception Handling in ColdFusion

Used with one or more CFCATCH tags, the CFTRY tag allows developers to catch and process exceptions in ColdFusion pages. 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.

You use the following syntax for CFTRY/CFCATCH blocks:

```
<CFTRY>
... Add code here ...

    <CFCATCH TYPE="exception type">
    ... Add exception processing code here ...
    </CFCATCH>

    ... Additional CFCATCH blocks go here ...
</CFTRY>
```

In order for ColdFusion to handle an exception, it must appear within a CFTRY block. You might enclose an entire application page in a CFTRY block, using a CFCATCH block around a potential error.

To catch errors in a single problematic SQL statement, for example, you might narrow the focus by using a CFTRY block with a CFCATCH TYPE=”Database” tag, outputting the CFCATCH.State information as well.

See the *CFML Language Reference* for information on the CFTRY, CFCATCH, and CFTHROW tags.

Types of recoverable exceptions supported

The ColdFusion Server supports several types of recoverable exceptions. Use the TYPE attribute in the CFCATCH tag to determine which type of exception to catch.

Application-defined exception events

ColdFusion applications can raise exceptions using the CFTHROW tag, with an optional diagnostic message. CFTHROW raises an exception that can be caught by a CFCATCH TYPE="Application" tag, or a CFCATCH TYPE="Any" tag. This exception can also be caught by a CFCATCH block that has no TYPE attribute.

Database failures

Use the CFCATCH tag with TYPE="Database" or CFCATCH TYPE="Any" to catch failed database operations, such as failed SQL statements, ODBC problems, and so on.

Template errors

Use the CFCATCH tag with TYPE="Template" or TYPE="Any" to catch general application page errors.

Missing included file errors

Use the CFCATCH tag with TYPE="MissingInclude" or TYPE="Any" to catch errors for missing included files.

Object exceptions

Use the CFCATCH TYPE="Object" tag to catch exceptions in ColdFusion code that works with objects.

Security exceptions

Use the CFCATCH TYPE="Security" tag to raise catchable exceptions in ColdFusion code that works with security.

Expression exceptions

Use the CFCATCH TYPE=”Expression” tag to catch exceptions when an expression fails evaluation.

Locking exceptions

Use the CFCATCH tag with TYPE="Lock" to catch failed locking operations, such as when a CFLOCK critical section times out or fails at runtime.

Unexpected internal exceptions

You can catch unexpected exceptions in the ColdFusion Server with the CFCATCH TYPE="Any" tag.

Note Attempting to handle unexpected exceptions in CFML code can cause unpredictable results, and may seriously degrade or crash the ColdFusion Server.

Exception-Handling Strategies

Developers can use CFTRY with CFCATCH to handle exceptions based on their point of origin within an application page, or based on diagnostic information.

Handling exceptions based on point of origin

Use the CFTRY tag with one or more CFCATCH blocks to define a ColdFusion block for exception handling. When an application page raises an error condition, the ColdFusion server checks the stack of currently active blocks for a corresponding CFCATCH handler. At extremes, an exception-prone tag might be enclosed in a specialized combination of CFTRY and CFCATCH to immediately isolate the tag's exceptions, or to use CFTRY with CFCATCH TYPE="Any" at a main processing level to gracefully terminate a subsystem's processing in case of an unexpected error.

Handling exceptions based on diagnostic information

Use CFCATCH with the attribute TYPE="*exception type*" to catch specific types of exceptions. A CFCATCH handler can further analyze the exception's diagnostic information, and re-throw the exception if the exceptional condition requires further handling.

Exception Handling Example

The following example shows CFTRY and CFCATCH, using a sample data source called *company* and a sample included file, `includeme.cfm`.

If the data access driver raises an exception during the CFQUERY statement's execution, the application page flow continues 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. Any unknown, but possibly recoverable, exceptions are handled by the CFCATCH TYPE="Any" block.

```
<!--- Wrap code you want to check in a CFTRY block --->

<CFTRY>
    <CFQUERY NAME="test" DATASOURCE="company">
        SELECT DepartmentID, FirstName, LastName
        FROM employees
        WHERE employeeID=#EmpID#
    </CFQUERY>

    <HTML>
    <HEAD>
        <TITLE>Test CFTRY/CFCATCH</TITLE>
    </HEAD>

    <BODY>
    <HR>
    <CFINCLUDE TEMPLATE="includeme.cfm">
    <CFOUTPUT QUERY="test">
    <P>Department: #DepartmentID#
    <P>Last Name: #LastName#
    <P>First Name: #FirstName#
    </CFOUTPUT>

    <HR>

<!--- Use CFCATCH to test for missing included files.
        Print Message and Detail error messages. --->

    <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>
    </CFCATCH>

<!--- Use CFCATCH to test for database errors.
        Print error messages. --->

    <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>
    </CFCATCH>

<!--- Use CFCATCH with TYPE="Any"
    to find unexpected exceptions. --->

    <CFCATCH TYPE="Any">
    <H1>Other Error: #CFCATCH.Type#</H1>

    <CFOUTPUT>
    <UL>
    <LI><b>Message:</b> #CFCATCH.message#
    <LI><b>Detail:</b> #CFCATCH.Detail#
    </UL>
    </CFOUTPUT>
    </CFCATCH>
</CFTRY>
    </BODY>
    </HTML>
```

CFTHROW syntax

Use CFTHROW within a CFTRY block to raise an error condition. The CFCATCH block can access this message through CFCATCH.message.

```
<CFTHROW MESSAGE="...diagnostic message...">
```

This form of the CFTHROW tag throws a new CFML-recoverable exception with the specified diagnostic message.

Using CFTHROW without the MESSAGE attribute throws a new CFML-recoverable exception with an empty diagnostic message.

CFTRY syntax

The CFTRY tag starts a ColdFusion exception-handling block. One or more CFCATCH tags must be included within a CFTRY block.

```
<CFTRY>
    ...Other CFML tags...

    <CFCATCH TYPE="Any">
</CFTRY>
```

Note A CFCATCH block must be the last set of tags within a CFTRY block.

CFCATCH syntax

The CFCATCH tag catches exceptions of the type specified in the TYPE attribute, such as database, application, missing include, or application page.

`<CFCATCH TYPE=”`*`exception type`*`”>`

The following form of the CFCATCH tag catches all CFML-recoverable exceptions generated within the preceding CFTRY block, or within any of the CFTRY block’s children.

```
<CFCATCH TYPE="Any">
```

A CFCATCH tag without a TYPE attribute is equivalent to CFCATCH TYPE="Any".

Order of evaluation

For a given CFTRY block, CFCATCH tags are tested in the order in which they appear in the application page.

Note An exception raised within a CFCATCH block cannot be handled by the CFTRY block that immediately encloses the CFCATCH tag.

See the *CFML Language Reference*for information on the syntax of the exception handling tags, CFTRY, CFCATCH, and CFTHROW.

Exception Information in CFCATCH

Within a CFCATCH block, the active exception’s properties can be accessed as variables:

CFCATCH.TYPE -- The exception’s type, returned as a string:

- Application
- Database
- Template

- MissingInclude
- Object
- Security
- Expression
- Lock
- Any

CFCATCH.MESSAGE — The exception's diagnostic message, if one was provided. If no diagnostic message is available, this is an empty string.

CFCATCH.DETAIL — A detailed message from the CFML interpreter. This message, which contains HTML formatting, can help to determine which tag threw the exception.

Database exceptions

For database exceptions, ColdFusion supplies some additional diagnostic information. The following variables are available whenever the exception type is database:

CFCATCH.NATIVEERRORCODE — The native error code associated with this exception. Database drivers typically provide error codes to assist in the diagnosis of failing database operations. The values assumed by CFCATCH.NATIVEERRORCODE are driver-dependent. If no error code is provided, the value of NativeErrorCode is -1.

CFCATCH.SQLSTATE — The SQLSTATE code associated with this exception. Database drivers typically provide error codes to assist in the diagnosis of failing database operations. The values assumed by CFCATCH.SQLSTATE are driver-dependent. If no SQLSTATE value was provided, the value of SQLSTATE is -1.

Locking exceptions

For exceptions related to CFLOCK sections, there is additional information available within the CFCATCH block:

CFCATCH.LOCKNAME — The name of the affected lock. This is set to “anonymous” if the lock name is not known.

CFCATCH.LOCKOPERATION — The operation that failed. This is set to “unknown” if the failed operation is unknown.

MissingInclude exceptions

For exceptions related to missing files, where the type of exception is MissingInclude, the following variable is available:

CFCATCH.MISSINGFILENAME — The name of the file missing in an exception of type MissingInclude.

CHAPTER 10

Accessing the Registry

ColdFusion includes the CFREGISTRY tag, which allows you to get, set, and delete registry values.

Contents

- Overview of Registry Access in ColdFusion 92
- Getting Registry Values 92
- Setting Registry Values 93
- Deleting Registry Values 94

Overview of Registry Access in ColdFusion

ColdFusion includes the CFREGISTRY tag, which allows you to get, set, and delete registry values. The registry is a database that Windows NT uses to maintain hierarchical information about users, hardware, and software. It includes keys and values:

- Keys can contain either values or other keys. A key and the keys/values below it are referred to as a *branch*.
- Values are conceptually split into two parts: *value name* and *value data*.

To maintain consistency with other CFML tags, the CFREGISTRY tag refers to keys and value names as *entries*. Additionally, the CFREGISTRY SET action uses the *value* attribute to refer to value data.

Solaris note: ColdFusion for Solaris includes functionality that emulates the registry.

The registry contains information critical to your system. Be very careful when modifying and deleting registry values. Depending on expected usage, you might consider using the Basic Security tab of the ColdFusion Administrator to implement a tag restriction on the CFREGISTRY tag (this is especially true for ISPs, whose server may host a large and diverse set of applications).

Getting Registry Values

You can use CFREGISTRY with either the GET or GETALL actions to retrieve multiple keys and values from the registry.

Getting all keys and values

Use CFREGISTRY with the GETALL action to return all registry keys and values defined in a branch. You can access these values as follows:

- CFREGISTRY creates a record set that contains #Entry#, #Type#, and #Value#. You can access through tags such as CFOUTPUT. To fully qualify these variables use the record set name, as specified in the NAME attribute of the CFREGISTRY tag.
- If #Type# is a key, #Value# is an empty string.
- If you specify Any for TYPE, GetAll also returns any binary registry values. For binary values, the #Type# variable contains UNSUPPORTED and #Value# is blank.

  You can optionally specify the SORT attribute to sort the record set based on the contents of the Entry, Type, and Value columns. Specify any combination of columns in a comma separated list. ASC (ascending) or DESC (descending) can be specified as qualifiers for column names. ASC is the default. For example:

  ```
  Sort="type ASC, entry ASC"
  ```

To get all values for a specified registry key:

1. Code a CFREGISTRY tag with the GETALL action, specifying the branch, type, and record set name.

```
<CFREGISTRY ACTION="GetAll"
    BRANCH="HKEY_LOCAL_MACHINE\Software\Microsoft\Java VM"
    TYPE="Any" NAME="RegQuery">
```

2. Access the record set (this example uses the CFTABLE tag):

```
<H1>CFREGISTRY ACTION="GetAll"</H1>
<CFTABLE QUERY="RegQuery" COLHEADERS
    HTMLTABLE BORDER="Yes">
<CFCOL HEADER="<B>Entry</b>" WIDTH="35"
    TEXT="#RegQuery.Entry#">
<CFCOL HEADER="<B>Type</b>" WIDTH="10"
    TEXT="#RegQuery.Type#">
<CFCOL HEADER="<B>Value</b>" WIDTH="35"
    TEXT="#RegQuery.Value#">
</CFTABLE>
```

Getting a specific value

Use CFREGISTRY with the GET action to access a single registry value and store it in a ColdFusion variable.

To get a specific registry value:

1. Code a CFREGISTRY tag with the GET action, specifying the branch, the entry to be accessed, the type (optional), and a variable in which to return the value.

```
<CFREGISTRY ACTION="Get"
    BRANCH="HKEY_LOCAL_MACHINE\Software\Microsoft\Java VM"
    ENTRY="ClassPath" TYPE="String" Variable="RegValue">
```

2. Access the variable:

```
<H1>CFREGISTRY ACTION="Get"</H1>
<CFOUTPUT>
<P>
Java ClassPath value is #RegValue#
</CFOUTPUT>
```

Setting Registry Values

Use CFREGISTRY with the SET action to add a registry key, add a new value, or update value data. CFREGISTRY creates the key or value if it does not exist.

To set a registry value:

Call the CFREGISTRY tag with the SET action, specifying the branch, the entry to set, the type of data contained in the value, and the value data. This example assumes a session variable named LastFileName:

```
<CFREGISTRY ACTION="Set"
    BRANCH="HKEY_LOCAL_MACHINE\Software\cflangref"
    ENTRY="LastCFM01" TYPE="String"
    VALUE="#SESSION.LastFileName#">
```

If the specified value does not exist, ColdFusion creates it. If the value already exists, ColdFusion updates the value data.

To set a registry key:

Call the CFREGISTRY tag with the SET action, specifying the branch, the entry to set, specifying KEY for the TYPE attribute:

```
<CFREGISTRY ACTION="Set"
    BRANCH="HKEY_LOCAL_MACHINE\Software\cflangref"
    ENTRY="Temp" TYPE="Key">
```

Deleting Registry Values

You can use CFREGISTRY with the DELETE action to delete registry keys and values.

To delete a registry value:

Call the CFREGISTRY tag with the DELETE action, specifying the branch and value name:

```
<CFREGISTRY ACTION="Delete"
    BRANCH="HKEY_LOCAL_MACHINE\Software\cflangref"
    ENTRY="LastCFM01">
```

To delete a registry key:

Call the CFREGISTRY tag with the DELETE action, specifying the branch of the key to be deleted (including the key name):

```
<CFREGISTRY ACTION="Delete"
    BRANCH="HKEY_LOCAL_MACHINE\Software\cflangref">
```

Be careful when using the DELETE action; if you delete a key, CFREGISTRY also deletes values and subkeys defined beneath the key.

CHAPTER 11

Building ColdFusion Extensions

This chapter provides information about building and deploying ColdFusion Extensions or CFXs. In this release of ColdFusion, we're consolidating our approach to ColdFusion extensions (CFXs). In addition to custom tags built in CFML, CFXs can be built using C/C++, JavaScript/VBScript, COM/CORBA, or Java, VTML, and WIZML.

Contents

- About ColdFusion Extensions.. 96
- Building ColdFusion Extensions in CFML 96
- Installing Custom Tags.. 97
- Writing Custom Tags 97
- Example Tags 98
- CFML 4.0 Custom Tag Enhancements 102
- Managing Custom Tags 109
- Securing Custom Tags.. 110
- Encrypting Custom Tags.. 110
- Building Extensions in C++ 111
- Implementing CFX Tags 112
- Managing CFX Tags.. 114
- Using COM and CORBA Objects... 115
- Getting Started with COM Components 116
- Using the OLE/COM Object Viewer 117
- Creating and Using COM Objects... 118
- Setting Properties and Invoking Methods.. 120
- Getting Started with CORBA Objects.. 121

About ColdFusion Extensions

ColdFusion Extensions (CFXs) are an open XML-based framework for extending ColdFusion with new server components and connectivity to enterprise systems using COM, CORBA, C/C++, VBScript, JavaScript, or CFML.

In addition, ColdFusion Studio supports two built-in languages, VTML and WIZML. VTML is an XML based language used to define visual tool components and dialogs. WIZML, also an XML based language is used to create application wizards that can be distributed to developers working in ColdFusion Studio. For more information see *Customizing the Development Environment.*

Building ColdFusion Extensions in CFML

ColdFusion custom tags built in CFML, a technology introduced in ColdFusion 3.0, are an essential part of the ColdFusion support for rapid application development and code re-use. Custom tags are now a valued resource for the ColdFusion developer community and demonstrate the rich variety of solutions — utilitarian, sophisticated, and even whimsical — that can be built in ColdFusion.

Custom tags extend the ColdFusion development model of encapsulating complexity by enabling you to wrap functionality in a page that can be called from a ColdFusion application page.

Allaire Tag Gallery

The success of CFML custom tags is best seen by a visit to the Tag Gallery at http://www.allaire.com/taggallery. Tags are grouped in several broad categories and are downloadable as freeware, shareware, or commercial software. You can quickly 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 API, see Chapter 12, "The ColdFusion Extension API," on page 123.

Allaire Alive

An online RealVideo title called "Using Custom Tags" is available at the Allaire alive section of our Web site. It presents an overview of custom tags as a component architecture for the emerging Web platform and outlines the creation and use of CFML custom tags.

The video is part of Allaire Alive, an educational service that offers Web videos on topics specific to ColdFusion development and application deployment as well as broader industry issues. The titles are available free for online viewing or download.

Custom Tag Editors

As you scroll through the Gallery listings, you will notice a number of the tags are marked with <VTM> after the tag name. These tags include a special file written in Allaire's Visual Tool Markup Language. Click on the <VTM> link to read about how VTML is used to create custom interfaces in ColdFusion Studio.

When you download and unzip a custom tag that includes a vtm file, copy that file to the /Templates/TagEditors folder under your Studio root directory. When you insert the custom tag into a page, a tag editor (Ctrl + F4) is available for the selected tag. Many of these editors contain embedded help for their syntax and usage.

See *Customizing the Development Environment* for more information.

Installing Custom Tags

Custom tags are cfm files with a difference. They are created just like any other ColdFusion page, but they must be installed in a specific location to be accessible from the calling template. ColdFusion loads the first instance it finds of the custom tag called by a template, so avoid placing copies of a custom tag in different locations. Custom tags written in CFML are typically named using the CF_* convention to distinguish them from CFXs written in C/C++, which use the CFX_* convention.

Local tags

The ColdFusion engine responds to a request for a custom tag by first searching the directory of the calling template. This allows you to keep a custom tag file in the same directory as the page that uses it.

Shared tags

To share a custom tag among applications in multiple directories, place it in the Custom Tags folder under your ColdFusion installation directory. You can create sub-folders to organize custom tags — ColdFusion searches recursively for the Custom Tags directory, stepping down through any existing subdirectories until the custom tag is found.

Writing Custom Tags

All CFML constructs can be used in custom tags and HTML can be included, too. You only need to be aware of a few requirements when creating custom tags.

Naming Custom Tags

Custom tags are identified by the CF_ prefix. Beyond that, 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. For example, the tag name CF_MyTag invokes the file MyTag.cfm

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, the CFMODULE element provides a solution. See Resolving file name conflicts.

Tag scope

Because custom tags are individual templates, there is no automatic exposure of variables and other data between a custom tag and the calling template. To pass data, you define attributes for the custom tag just as in standard CFML coding.

Data pertaining to the HTTP request or to current application is visible, however. This includes the variables in Form, URL, CGI, Cookies, Server, Application, Session, and Client.

Defining attributes

As the creator of the custom tag, you have the responsibility to specify a syntax for the tag's functionality. CFML custom tags support both required and optional attributes. Attributes are defined as name-value pairs. Custom tag attributes conform to CFML coding standards:

- Attributes are case-insensitive.
- Attributes may be listed in any order within a tag.
- Attribute=value pairs for a tag must be separated by a space.
- Passed values that contain spaces must be enclosed in double-quotes.

Example Tags

You may have already jumped to the Tag Gallery to peruse the selections; your own interests and development needs will guide you there. A few samples will be presented here to illustrate their rich variety. These samples are taken from some of the Gallery's major tag categories.

Utility tags

CF_MERGEQUERY, written by Michael Dinowitz, performs a single task that can be very useful when outputting multiple query result sets. It is distributed with

ColdFusion and can be selected from the Custom Tags folder in the Tag Chooser (CTRL+E) in ColdFusion Studio.

```
<CF_MERGEQUERY Query1 = "query1" Query2="query2">
```

Function tags

Custom tags are often developed to perform a specific operation within an application. They can be written to supply interface elements and data for ColdFusion tags and functions.

CF_COUNTWORD, written by Rob Bilson, performs a well-defined function using just three CFSET tags to handle input, processing, and output. This tag replicates the function of an existing CFX tag written in C++ but it is easier to implement because it does not have to be explicitly registered. The code is well-commented and the author created a custom editor in VTML for the tag, making it easy for others to use.

```
<!--- set local variable to the passed attribute --->
<CFSET MyString = attributes.Mystring>

<!--- Get the number of words in the string by treating
the string as a list and using the space character as the
delimiter. Note that the tag assumes a single line string
where words are separated by one or more spaces --->

<CFSET WordsInString = ListLen(MyString, " ")>

<!--- return the count back to the calling template --->
<CFSET Caller.NumberOfWords = WordsInString>
```

User interface tags

CF_COOLLINK, written by Brian Shin, allows you to generate links that change color on mouseover and then change color again when you click the link. You can also specify a custom message to appear in the browser’s status bar.

As you read through the code you will notice that it utilizes a number of familiar CFML and HTML elements:

- An easily-configurable set of attributes
- An embedded style tag to specify output formatting
- JavaScript that initiates mouseover actions based on results of a browser test
- Conditional logic
- Output of returned values from variables

The CoolLink code

```
<--- CoolLink.cfm to change link appearance
    and properties --->
<--- This tag should be called as follows:
    CF_CoolLink
        LinkName = “Allaire”
        LinkHref = “http://www.allaire.com”
        OnColor = “Purple”
        OffColor = “Orange”
        ClickColor = “Red”
        >
        etc.  --->

<!--- NOTE: This tag allows links to change color in IE4,
and the most control can be achieved in IE4. However, it
should gracefully degrade (not give errors) for all
other browsers -->

<!--- Initialize all Attribute Scope variables so
they have defaults --->
<CFPARAM NAME=”Attributes.LinkName” DEFAULT=”Allaire”>
<CFPARAM NAME=”Attributes.LinkHref”
    DEFAULT=”http://www.allaire.com”>
<CFPARAM NAME=”Attributes.OnColor” DEFAULT=”violet”>
<CFPARAM NAME=”Attributes.OffColor” DEFAULT=”Navy”>
<CFPARAM NAME=”Attributes.ClickColor” DEFAULT=”Red”>
<CFPARAM NAME=”Attributes.LinkFont” DEFAULT=”Arial”>
<CFPARAM NAME=”Attributes.LinkSize” DEFAULT=”14”>
<CFPARAM NAME=”Attributes.LinkMessage”
    DEFAULT=”#Attributes.LinkHref#”>

<BODY LINK=<CFOUTPUT>”#Attributes.OffColor#”</CFOUTPUT>
vlink=”<CFOUTPUT>#Attributes.OffColor#”</CFOUTPUT> >

<STYLE>
    BODY
    {font-family:<CFOUTPUT>#Attributes.LinkFont#</CFOUTPUT>
    ;color:black;font-size:
        <CFOUTPUT>#Attributes.LinkSize#</CFOUTPUT>;}

</STYLE>

<SCRIPT LANGUAGE=”javascript”>
<!--- find out what browser is being used --->
browserType = navigator.appName
browserVer = parseInt(navigator.appVersion)

    if (browserType == “Microsoft Internet Explorer” && browserVer >= 4)
    browser = “IE4”;
    else browser = “other”

    if (browser == “IE4”) { document.body.onmouseover=makeCool;
    document.body.onmouseout=makeNormal;

<!--- this will change the color of the link when moused-over,
if you have IE4--->
    function makeCool() {
        src = event.toElement;
        if (src.tagName == ‘A’) {
            src.oldcol = ‘<CFOUTPUT>#Attributes.OffColor#</CFOUTPUT>’;
            src.style.color ‘<CFOUTPUT>#Attributes.OnColor#</CFOUTPUT>’;
        }
    }
    function makeNormal() {
        src=event.fromElement;
        if (src.tagName == ‘A’)  {
            src.style.color ‘<CFOUTPUT>#Attributes.OffColor#</CFOUTPUT>’;
        }
    }

<!--- this will change the color of the link
when it’s clicked, if you have IE4 --->
    function makeCooler() {
    if (src.tagName == ‘A’) {
        src.oldcol = ‘<CFOUTPUT>#Attributes.OffColor#</CFOUTPUT>’;
        src.style.color ‘<CFOUTPUT>#Attributes.ClickColor#</CFOUTPUT>’;
        }
    }

</SCRIPT>

<CFSET BROWSERTYPE=#CGI.HTTP_USER_AGENT#>
    <!--- If IE4 --->
<CFIF (FIND(“MSIE 4.0”, BROWSERTYPE))>
    <!--- Make the Link Cool --->
    <CFSET CLICKACTION = “onclick=’makeCooler()’”>
<CFELSE>
    <!--- Otherwise don’t make the link highlighted --->
    <CFSET CLICKACTION = “   “>
</CFIF>

<!--- Output the Link --->
<CFOUTPUT>
<A HREF=”#Attributes.linkhref#” TARGET=”main”
ONMOUSEOVER=”window.status=’#Attributes.linkmessage#’; return true”
#CLICKACTION#  >#Attributes.linkname#</A>
</CFOUTPUT>

<--- /CoolLink.cfm to change link appearance and properties --->
```

CFML 4.0 Custom Tag Enhancements

This release of ColdFusion adds significant new features to CFML custom tags. These changes are part of overall architectural enhancements designed to address requests for greater power and flexibility in custom tags.

Main features of CFML 4.0 custom tags

- 100% backward compatibility
- End tags are accessible through any invocation syntax
- A tag's generated content is accessible
- Sub-tags can communicate their attributes to the base tag
- Collaborating tags can exchange data without user intervention
- Tag implementations can be provided in a single file
- Tags have control over iteration, that is, the number of times the tag body executes

Tag nesting

In CFML 4.0, any tag with an end tag present can be an ancestor to another tag.

For developers seeking to encapsulate complex functionality or data operations, nesting custom tags can be a productive mechanism. The logic of nesting tags is based on the relationship you establish between elements within a custom tag. Generally, ancestor/descendant and parent/child terminology is used to describe nested hierarchies, as it provides an easily recognizable frame of reference. A more generic terminology uses the idea of base tags and the sub-tags they contain. These relationships can be framed according to individual preferences and the exact nature of a given hierarchy.

Associating sub-tags with the base tag

While the ability to create nested custom tags is a tremendous productivity gain, keeping track of complex nested tag hierarchies can become a chore. A simple mechanism, 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.

See "High-level data exchange" on page 108for details.

Tag instance data

During the execution of a custom tag template ColdFusion keeps some amount of data related to the tag instance. The ThisTag scope is used to preserve this data with a unique identifier. The behavior is similar to the File scope.

The following variables are generated by the ThisTag scope:

- ExecutionMode — valid values are “start” and “end”
- HasEndTag — used for code validation, it distinguishes between custom tags that have and don’t have end tags for ExecutionMode=start. The name of the Boolean value is ThisTagHasEndTag.
- GeneratedContent — can be processed as a variable.
- AssocAttribs — holds the attributes of all nested tags if CFASSOCIATE was used them.

Pattern of execution

The same CFML template may be executed for both the start and end tag of a custom tag.

Modes of execution

A custom tag template may be invoked in either of two modes:

- Start tag execution
- End tag execution

If an end tag is not explicitly provided and shorthand empty element syntax (<TagName …/>) is not used, then the custom tag template will be invoked only once in start tag mode. If a tag must have an end tag provided, use ThisTag.HasEndTag during start tag execution to validate this.

Specifying execution modes

A variable with the reserved name ThisTag.ExecutionMode will specify the mode of invocation of a custom tag template. The variable will have one of the following values:

- Start — start tag execution
- End — end tag execution

During the execution of the body of the custom tag, the value of the ExecutionMode variable is going to be *inactive*. In this framework, the template of a custom tag that wants to perform some processing in both modes may look something like the following:

```
<CFIF ThisTag.ExecutionMode is 'start'>
    <!--- Start tag processing --->
<CFELSE>
    <!--- End tag processing --->
</CFIF>
```

CFSWITCH can also be used:

```
<CFSWITCH expression=#ThisTag.ExecutionMode#>
    <CFCASE value='start'>
        <!--- Start tag processing --->
    </CFCASE>
    <CFCASE value='end'>
        <!--- End tag processing --->
    </CFCASE>
</CFSWITCH>
```

CFEXIT

CFEXIT terminates execution of a custom tag. In ColdFusion 4.0, CFEXIT has been extended with a METHOD attribute that specifies where execution continues. With the introduction of start and end tags for custom tags, CFEXIT can specify that processing continues from the first child of the tag or continues immediately after the end tag marker.

The METHOD attribute can also be used to specify that the tag body should be executed again. This enables custom tags to act as high-level iterators, emulating CFLOOP behavior.

The following table summarizes CFEXIT behavior:

CFEXIT Behavior in a Custom Tag

METHOD Attribute Value	Location of CFExit Call	Behavior
ExitTag (default)	Base template	Acts like CFABORT
	ExecutionMode=start	Continue after end tag
	ExecutionMode=end	Continue after end tag
ExitTemplate	Base template	Acts like CFABORT
	ExecutionMode=start	Continue from first child in body
	ExecutionMode=end	Continue after end tag
Loop	Base template	Error

CFEXIT Behavior in a Custom Tag		
METHOD Attribute Value	**Location of CFExit Call**	**Behavior**
	ExecutionMode=start	Error
	ExecutionMode=end	Continue from first child in body

Access to generated content

Custom tags 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 portion of the results that is generated by the body of a given tag. This includes all results generated by descendant tags, too. Any changes to the value of this variable will result in changes to the generated content.

ThisTag.GeneratedContent 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.

As an example, consider a tag that comments out the HTML generated by its descendants. Its implementation could look something like this:

```
<CFIF ThisTag.ExecutionMode is 'end'>
<CFSET ThisTag.GeneratedContent =
       '<!--#ThisTag.GeneratedContent#-->'>
</CFIF>
```

Inter-tag data exchange

A key custom tag feature for CFML 4.0 is the ability of collaborating custom tags to exchange complex data without user intervention and without violating the encapsulation of a tag's implementation outside the circle of its collaborating tags. The following issues need to be addressed:

- 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 4.0 custom tags can expose all their data to collaborating tags.

Custom tag developers should document all variables that collaborating tags can access and/or modify. Developers of custom tags that collaborate with other custom tags should make sure that they do not modify any undocumented data.

We highly recommend that developers preserve encapsulation by putting all tag data access and modification operations into custom tags. For example, rather than documenting that the variable Q in a tag's implementation holds an important query result set and expecting users of the custom tag to manipulate Q directly, the developer should create another nested custom tag that manipulates Q. This protects the users of the custom tag from changes in the tag's implementation.

Where is data accessible?

Two custom tags can be related in a variety of ways in a template. One can be a sibling, parent, ancestor, child, or descendant of the other. For most practical purposes, sibling and cousin relationships rarely matter. Ancestor and descendant relationships do matter because they relate to the order of tag nesting.

A tag's descendants are inactive while the template is executed, that is, they have no instance data. The tag's data access is therefore restricted to ancestors only. Ancestor data will be available from the current template and from the whole runtime tag context stack. The tag context stack is the path from the current tag element back 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. CFINCLUDE tags and custom tags will appear on the tag context stack.

Custom tag names

To avoid ambiguity, the following naming convention should be observed for CFML tags:

- Native tags: *CFTagName*
- CFX tags: *CFX_TagName*
- Shorthand invocation: *CF_TagName*
- Shared tags: <CFMODULE Name="path.TagName"> *CF_TagName*
- Application-specific tags: <CFMODULE template="path/*TagName*.cfm"> *CF_TagName*

Note that since this naming convention makes the name of a custom tag template the same as the name of the tag, name collisions are possible. However, locality of reference suggests that tags with a close ancestral relationship are likely to be related. It is unlikely that unrelated tags with the same template name have a close ancestral relationship.

Ancestor data access

The ancestor's data is represented by a structure object that contains all the ancestor's data, in much the same way that a COM object in CFML contains properties.

The following set of functions provide access to ancestral data:

- GetBaseTagList() — Returns a comma-delimited list of uppercased ancestor tag names. An empty string is returned if this is a top-level tag. The first element of a non-empty list is the parent tag.
- GetBaseTagData(TagName, InstanceNumber=1) — Returns an object that contains all the variables, scopes, etc. 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 was snipped from a custom tag.

```
<CFIF thisTag.executionMode is 'start'>
    <!--- Get the tag context stack
     The list will look something like
    "CFIF,MYTAGNAME..." --->
    <CFSET ancestorList = getBaseTagList()>

    <!--- Output your own name because CFIF is
    the first element of the tag context stack --->
    <CFOUTPUT>I'm custom tag #ListGetAt(ancestorlist,2)#<p></CFOUTPUT>

    <!--- Determine whether you are nested inside a loop --->
    <CFSET inLoop = ListFindNoCase(ancestorList,'CFLOOP')>
    <CFIF inLoop neq 0>
        I'm running in the context of a CFLOOP tag.<p>
    </CFIF>

    <!--- Determine whether you are nested inside
    a custom tag. Skip the first two elements of the
    ancestor list, i.e., CFIF and the name of the
    custom tag I'm in --->
    <CFSET inCustomTag = ''>
    <CFLOOP index=elem
        list=#ListRest(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'>
            template because the tag is in
            its start or end execution mode.
        <CFELSE>
            body
        </CFIF>
        <p>
    <CFELSE>
        <!--- Say you are lonely --->
        I'm not nested inside any custom tags. :^( <p>
    </CFIF>

</CFIF>
```

High-level data exchange

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 tag syntax is:

```
<CFASSOCIATE BaseTag=base_tag_name
    DataCollection=collection_name>
```

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.*collection_name*'. The default value for the DataCollection attribute is 'AssocAttribs'. This attribute should be used only in cases where the base tag can have more than one type of sub-tag. It is convenient for keeping separate collections of attributes, one per tag type.

CFASSOCIATE performs the following operations:

```
<!--- Get base tag instance data --->
<CFSET data = getBaseTagData(baseTag).thisTag>

<!--- Create a string with the attribute
    collection name --->
<CFSET collectionName = 'data.#dataCollection#"'>

<!--- Create the attribute collection, if necessary --->
<CFIF not isDefined(collectionName)>
    <CFSET "#collectionName#" = arrayNew(1)>
</CFIF>

<!--- Append the current attributes
    to the array --->
<CFSET temp=arrayAppend(evaluate(collectionName), attributes)>
```

The CFML 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>
```

Managing Custom Tags

If you deploy custom tags in a multi-developer environment or distribute your tags publicly, you may want to make use of two additional ColdFusion capabilities:

- An advanced invocation syntax to resolve possible name conflicts
- Advanced Security. See *Administering ColdFusion Server* for information about implementing Advanced Security and also Application Security in this book.
- Encryption. See “Encrypting Custom Tags” on page 110 for more information.

Resolving file name conflicts

To avoid errors caused by duplicate custom tag file names, use the CFMODULE tag in the calling template. Note that only one of the required attributes can be used in a given instance of the tag:

CFMODULE Attributes

Attribute	Description
Template	Required. Specifies a relative path to the cfm file. Same as Template attribute in <CFINCLUDE>. Example: <CFMODULE TEMPLATE="../MyTag.cfm"> identifies a custom tag file in the parent directory.

CFMODULE Attributes	
Attribute	**Description (Continued)**
Name	Required if Template attribute is not used. Use period -separated names to uniquely identify a sub-directory under the Custom Tags root directory. Example: <CFMODULE NAME="Allaire.Alive.GetUserOptions"> identifies the file GetUserOptions.cfm in Custom Tags\Allaire.Alive directory under the ColdFusion root directory.
Attributes	Optional. You can list the custom tag's attributes.

Securing Custom Tags

ColdFusion's security framework enables you to selectively restrict access to individual tags or to tag directories. This can be an important safeguard in team development. To use this feature, you register Custom Tags as a security resource on the ColdFusion Administrator Advanced Security page and then enter the tags by name or by directory.

For more information about securing custom tags using ColdFusion Advanced security, see *Administering ColdFusion Server*.

Encrypting Custom Tags

The command-line utility `cfcrypt` can be used to encrypt any ColdFusion application. By default, the utility is installed in the `/cfusion/bin` directory. It is especially useful for securing custom tag code before distributing it.

CFCRYPT uses the following syntax:

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

The following options are supported:

cfcrypt Command Line Options	
Option	**Description**
input file	Name of the file you want to encrypt. cfcrypt will not process an encrypted file.
output file	Path and filename of the output file. **Warning**: If no output file name is specified, a warning message asks if you want to continue. If you continue the process, the encrypted file overwrites the source file.

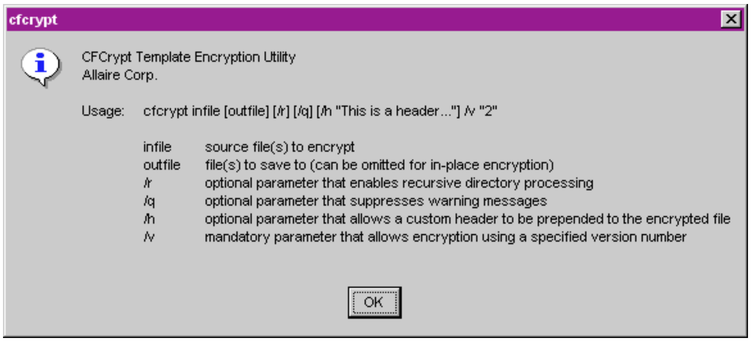
cfcrypt Command Line Options	
Option	**Description**
/r	Recursive, when used with wildcards, recurses through subdirectories to encrypt files.
/q	Suppresses warning messages.
/h	Header, allows custom header to be written to the top of the encrypted file(s).
/v	Required parameter that allows encryption using a specified version number. Use "1" for pages you want to be able to run on ColdFusion 3.x. Use "2" for pages you want to run strictly on ColdFusion 4.0 and later.

Example

```
cfcrypt c:\inetpub\wwwroot\myapp\entrypoint.cfm tags\custom01.cfm /h
"The code in this custom tag is encrypted" /v "2"
```

This command encrypts `entrypoint.cfm` for use with ColdFusion Server 4.0. and saves it to the subdirectory `encryptedpages\enrtypoint.cfm`. If you attempt to open the output file for editing, only the message specified in the `/h` option will be readable.

If you enter `cfcrypt` without arguments, a message box appears (Windows) showing the command syntax you can use.



```
CFCrypt Template Encryption Utility
Allaire Corp.

Usage:   cfcrypt infile [outfile] [/r] [/q] [/h "This is a header..."] /v "2"

         infile    source file(s) to encrypt
         outfile   file(s) to save to (can be omitted for in-place encryption)
         /r        optional parameter that enables recursive directory processing
         /q        optional parameter that suppresses warning messages
         /h        optional parameter that allows a custom header to be prepended to the encrypted file
         /v        mandatory parameter that allows encryption using a specified version number
```

Note While it is possible to encrypt binary files with CFCRYPT, it is not recommended.

Building Extensions in C++

Another technology supported for extending ColdFusion is C++. ColdFusion exposes a C++ based API you can employ for encapsulating specific features in a ColdFusion tag, generally known as a CFX tag. The ColdFusion Application Programming Interface

(CFAPI) is a C++ based API for creating C++ based tags for use in ColdFusion. These custom tags implemented as DLLs and have the following capabilities:

- The ability to handle any number of custom attributes.
- The ability to use and manipulate ColdFusion queries for custom formatting.
- The ability to generate ColdFusion queries for interfacing with non-ODBC based information sources.
- The ability to dynamically generate HTML to be returned to the client.
- The ability to set variables within the ColdFusion application page from which they are called.
- The ability to throw exceptions which result in standard ColdFusion error messages.

On Windows NT, you can get started quickly by using the ColdFusion Custom Tag Visual C++ AppWizard to generate a tag. The custom tag wizard is automatically installed during setup if Visual C++ 4.0 or higher is present on your system. By modifying the default tag implementation and experimenting, you will quickly learn how to use the API.

Before you can use your C++ compiler to build custom tags, you must enable the compiler to locate the CFAPI header file, `cfx.h`. On Windows NT, you do this by adding the CFAPI Include directory (`\cfusion\cfapi\include`) to your list of global include paths. On Solaris, you will need `-I <`*`includepath`*`>` on your compile line (see the Makefile directory list example).

Solaris only

CFX tags built on Solaris must be thread safe and should be compiled with the `-mt` switch on the Sun compiler.

Sample C++ tags

Two CFX tags are included to give you additional insight into working with the CFAPI. The two example tags are:

- CFX_DIRECTORYLIST — Queries a directory for the list of files it contains.
- CFX_NTUSERDB (Windows NT only) — Allows addition and deletion of NT users.

On Windows NT, these tags are located in the `\cfusion\cfxapi\examples` directory. On Solaris, look in *`installdirectory`*`/coldfusion/cfx/examples`.

Implementing CFX Tags

The key concept to understand in building custom tags is the use of 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 allow the custom tag to accomplish its work.

See Chapter 12, “The ColdFusion Extension API,” on page 123 for reference information about the CFX API.

Debugging CFX tags

Once a debug session is configured, you can run your custom tag from within the debugger, set breakpoints, single-step, and so on.

Windows NT

Custom tags can easily be debugged within the Visual C++ environment. To debug a tag, open the Build Settings dialog and click the Debug tab. Set the Executable for debug session setting to the full path to the ColdFusion Engine (such as, `c:\cfusion\bin\cfserver.exe`) and set the program arguments setting to `-DEBUG`.

Solaris

You can debug custom tags on Solaris using the dbx debugger. You should shutdown ColdFusion using the stop script.

Set the environment variables, including `LD_LIBRARY_PATH` and `CFHOME` as they are set in the start script. You should then be able to run the cfserver executable under the dbx debugger and set break points in your CFX code. You may need to set a break point in main (“stop in main”) so dbx loads the symbols for your CFX before you can set breakpoints in your code.

Registering CFX tags

You must register custom tags in the ColdFusion Administrator before you can use them within application pages. To register or modify the settings for a tag, use the Extensions, CFX Tags page of the ColdFusion Administrator, as described in “Managing CFX Tags” on page 114in this chapter.

Windows NT only. The Visual C++ Custom Tag Wizard automatically registers custom tags so that they can be tested and debugged.

Distribution

If you are distributing a custom tag, you may want to automatically register the custom tag during the setup process by writing the registration entries directly into the Registry. The location, key, and value names to write are as follows:

Hive — `HKEY_LOCAL_MACHINE`

Key — `SOFTWARE\Allaire\ColdFusion\CurrentVersion\CustomTags\`*`TagName`*

Values:

- LibraryPath — The full path to the DLL (Windows NT) or shared object (Solaris) that implements the custom tag.
- ProcedureName — The name of the procedure to call for processing tag requests.
- Description — A description of the tag's functionality for browsing by end users.
- CacheLibrary — Indicates whether to keep the DLL or shared object loaded in RAM (1 or 0).

You can create a file containing this information by using the Regedit utility to export the registry entry from a machine on which the custom tag is already installed.

You can also use the Regedit utility to import custom tags to the registry (on either Windows NT or Solaris). To do this:

1. Export the custom tag's registry entry by using the Regedit utility. This creates a file similar to the following:

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\Allaire\ColdFusion\CurrentVersion\
CustomTags\CFX_TEST]
"LibraryPath"="C:\\cfusion\\cfx\\CFX_TEST\\test.dll"
"ProcedureName"="ProcessTagRequest"
"Description"="Sample CFX tag."
"CacheLibrary"="1"
```

2. In the install script, import the registry entry by including the following command in the install script:

```
regedit importfilename
```

Managing CFX Tags

To use a CFX tag in your ColdFusion applications, first register it in the Extensions, CFX Tags page in the ColdFusion Administrator.

Adding a CFX tag

To add a CFX tag:

1. Click CFX Tags on the ColdFusion Administrator home page. The Registered CFX Tags page opens. All registered custom tags appear in the list.
2. Click Add. The New CFX Tag page opens.
3. Type in the new tag name after the CFX_ prefix.
4. Enter the path in the Server library (DLL) box or click Browse Server to locate the library you want to use.

5. Enter the procedure that implements the tag. The procedure name you enter must correspond with an existing procedure in the DLL you've chosen. Procedure names are case sensitive.
6. Click the Keep library loaded box to prevent having to reload the library into memory each time a referenced page is accessed.
7. Optionally type a description of the tag's function in the Description box. The text appears with the tag name in the Registered CFX Tags list.
8. Click Add to save the new tag.

Changing CFX tag settings

To change a CFX tag:

1. Click the tag you want to change in the Registered CFX Tags list.
2. Make changes as needed on the Edit CFX Tag page.
3. Click Apply to save the changes.

Deleting a CFX tag

To delete a CFX tag:

1. Click the tag you want to delete in the Registered CFX Tags list.
2. Click Delete on the Edit CFX Tag page. The tag is removed from the list but is not deleted from the system.

Using COM and CORBA Objects

ColdFusion supports COM and CORBA objects through the CFOBJECT tag. COM (Component Object Model) is an architecture defined by Microsoft to enable component portability, reusability, and versioning. OLE, for example, is an implementation of COM. DCOM (Distributed Component Object Model) is an implementation of COM for distributed services, allowing access to components residing on a local network or anywhere on the Internet.

COM objects can reside locally, or on any other machine on the network, or on other networks. Currently, COM is supported on Windows NT and Windows 95/98.

To find out more about COM/DCOM, go to http://www.microsoft.com.

About CORBA

CORBA (Common Object Request Broker Architecture) is a specification for a distributed object system. 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 2.0 on both Windows and Solaris.

The main features of CORBA 2.0 are:

- ORB Core
- OMG Interface Definition Language (OMG IDL)
- Interface Repository
- Language Mappings
- Stubs and Skeletons
- Dynamic Invocation and Dispatch
- Object Adapters
- Inter-ORB Protocols

You access CORBA objects through the CFOBJECT tag. An Object Request Broker (ORB) is required. Popular ORBs include Visibroker, Orbix, and CorbaPlus.

For more information, see “Getting Started with CORBA Objects” on page 121in this chapter.

Getting Started with COM Components

An important thing to keep in mind when working with COM objects in ColdFusion is that the components you use with ColdFusion are non-visual – that is, they don't have a graphical interface. They are server-side components that encapsulate business logic you can invoke in your ColdFusion applications. If you were to invoke an object with a graphical interface in your ColdFusion application, a window for the component might appear on the web server desktop, not the user's desktop. And each time the component was invoked, another window would open until server resources were exhausted.

COM objects used by ColdFusion are dynamically linked components. Late binding means that the component is not linked into ColdFusion until it’s actually needed. If you want to change a component on a live site, you just have to make sure no one is using it, and it’s free to be swapped out with a new version.

COM objects can be transparently relocated on a network. A component on a different machine on a network is treated the same as a component on the local system. Components can also be referenced on machines outside the local network by using DCOM.

Getting set up: COM

To make use of COM components in your ColdFusion application, you need at least the following items:

- The Microsoft COM/OLE Viewer, available (last time we checked) at http://www.microsoft.com/oledev/olecom/oleview.htm, is a handy tool you can use to view COM/OLE object interfaces.
- The COM objects you want to use in your ColdFusion application pages. These are typically DLLs or EXEs. These components should allow late binding, that is, implement the IDispatch interface. The COM/OLE Viewer from Microsoft allows you to view the component interface so that you can properly define the CLASS attribute for the CFOBJECT tag and the properties and methods for the object, in lieu of adequate documentation.

Register the object

Once you've acquired the object you want to use, you may need to register it with Windows NT in order for ColdFusion (or anything else) to find it. Some objects may be deployed with their own setup programs that register objects automatically, while others may require the use of the Windows NT regsvr32 utility. You can invoke `regsvr32.exe` either from a command prompt (a.k.a. DOS box) or using the NT Run command, in the following form:

```
regsvr32 c:\path\filename.dll
```

Find the component ProgID and methods

Your COM object should provide documentation explaining each of the component's methods and the ProgID. With this information, you're ready to work with the CFOBJECT tag. If you don't have documentation, use the COM/OLE Viewer to view the component's interface. The COM/ OLE Viewer will tell you all you need to know about the component's interface.

Using the OLE/COM Object Viewer

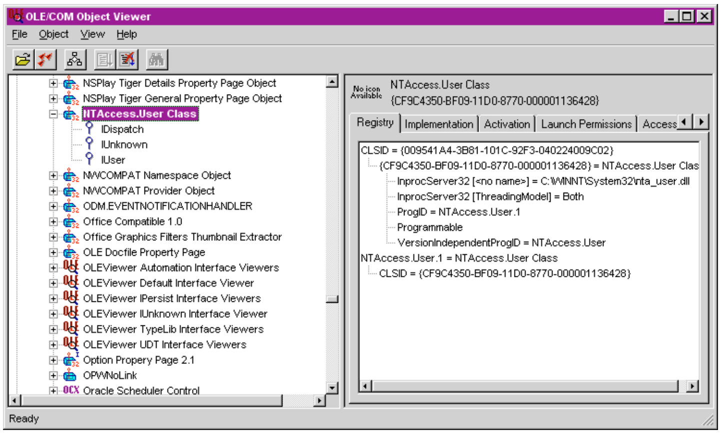
The OLE/COM Object Viewer is available for free from http://www.microsoft.com/oledev. The simple installation installs the executable by default as `\mstools\bin\oleview.exe`. You use the OLE viewer to retrieve a COM object's Program ID as well as its methods and properties.

Once you've installed the COM object, make sure you register it using the `regsvr32.exe` utility. Otherwise you won't find the object in the OLE viewer. The OLE 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 component you want to use, you can see the Program ID of the COM object you want to use. The OLE viewer also gives you access to options for the operation of the object.

To view an object's properties:

1. Open the OLE viewer and scroll to the object you want.



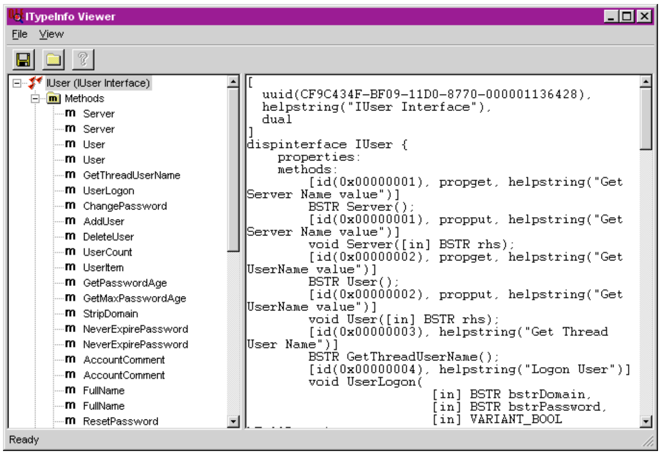
```
NSPlay Tiger Details Property Page Object
NSPlay Tiger General Property Page Object
NTAccess.User Class
    IDispatch
    IUnknown
    IUser
NWCOMPAT Namespace Object
NWCOMPAT Provider Object
ODM.EVENTNOTIFICATIONHANDLER
Office Compatible 1.0
Office Graphics Filters Thumbnail Extractor
OLE Docfile Property Page
OLEViewer Automation Interface Viewers
OLEViewer Default Interface Viewer
OLEViewer IPersist Interface Viewers
OLEViewer IUnknown Interface Viewer
OLEViewer TypeLib Interface Viewers
OLEViewer UDT Interface Viewers
Option Propery Page 2.1
OPWNoLink
Oracle Scheduler Control
```

```
No icon Available  NTAccess.User Class
                   {CF9C4350-BF09-11D0-8770-000001136428}
Registry | Implementation | Activation | Launch Permissions | Access

CLSID = {009541A4-3B81-101C-92F3-040224009C02}
  {CF9C4350-BF09-11D0-8770-000001136428} = NTAccess.User Clas
    InprocServer32 [<no name>] = C:\WINNT\System32\nta_user.dll
    InprocServer32 [ThreadingModel] = Both
    ProgID = NTAccess.User.1
    Programmable
    VersionIndependentProgID = NTAccess.User
NTAccess.User.1 = NTAccess.User Class
  CLSID = {CF9C4350-BF09-11D0-8770-000001136428}
```

2. Select and expand the object in the OLE viewer.

   By right clicking the object, an option appears for viewing it. If you view the TypeInfo, you'll see the object's methods and properties. Some objects will not have any access to the TypeInfo area. This is determined when an object is built and by the language used.



```
IUser (IUser Interface)
  Methods
    Server
    Server
    User
    User
    GetThreadUserName
    UserLogon
    ChangePassword
    AddUser
    DeleteUser
    UserCount
    UserItem
    GetPasswordAge
    GetMaxPasswordAge
    StripDomain
    NeverExpirePassword
    NeverExpirePassword
    AccountComment
    AccountComment
    FullName
    FullName
    ResetPassword
```

```
[
  uuid(CF9C434F-BF09-11D0-8770-000001136428),
  helpstring("IUser Interface"),
  dual
]
dispinterface IUser {
    properties:
    methods:
        [id(0x00000001), propget, helpstring("Get
Server Name value")]
        BSTR Server();
        [id(0x00000001), propput, helpstring("Get
Server Name value")]
        void Server([in] BSTR rhs);
        [id(0x00000002), propget, helpstring("Get
UserName value")]
        BSTR User();
        [id(0x00000002), propput, helpstring("Get
UserName value")]
        void User([in] BSTR rhs);
        [id(0x00000003), helpstring("Get Thread
User Name")]
        BSTR GetThreadUserName();
        [id(0x00000004), helpstring("Logon User")]
        void UserLogon(
                        [in] BSTR bstrDomain,
                        [in] BSTR bstrPassword,
                        [in] VARIANT_BOOL
```

Creating and Using COM Objects

In the following example, an SMTP mail handling component is created using CFOBJECT.

```
<CFOBJECT ACTION=CREATE
    NAME=MAILER
    CLASS=SMTP.Mailer>
```

The component needs to be created by ColdFusion before any methods in the component can be invoked or properties assigned in your application pages. This (hypothetical) SMTP component includes a large number of methods and properties you can use to perform a wide range of mail handling tasks. Methods perform actions and have return values you can use. Properties often return information or store information about a component. They do not execute and may not include any parameters. In the OLE/COM Viewer, methods and properties may be grouped together, making it a little confusing at first to determine one from the other.

Our hypothetical SMTP mail component includes properties such as:

- BodyText
- ConfirmRead
- ContentType
- FromName
- FromAddress

You use these properties to define elements of the mail message you want to send. The SMTP Mailer component also includes a number of methods, such as:

- SendMail
- AddRecipient
- AddCC
- AddAttachment

Two ways to create objects with CFOBJECT

There are essentially two ways to create objects using CFOBJECT: Using the Create method, which takes a COM object and instantiates it. The other connection type uses the Connect method, which links to an object that is already running on the server.

The ACTION attribute of CFOBJECT

The CFOBJECT ACTION attribute accepts two arguments, CREATE and CONNECT. You use CREATE to instantiate the object (typically a DLL) prior to invoking methods or assigning properties.

You use CONNECT to connect to an object (typically an EXE) that is already running on the specified server.

The CONTEXT attribute of CFOBJECT

The CFOBJECT CONTEXT attribute accepts three arguments:

- INPROC – This means an InProcess server object (typically a DLL) that is running in the same process space as the calling process, such as, ColdFusion.
- LOCAL – This is an OutofProcess server object (typically an EXE) that is running outside the ColdFusion process space but running locally on the same server.

- REMOTE – This is also an OutofProcess server object (also typically an EXE) that is running outside the ColdFusion process space, but running remotely, either on your corporate network, or out there somewhere on the Internet. Using REMOTE implies using the SERVER attribute to identify where the object resides.

Setting Properties and Invoking Methods

The following example, using a hypothetical SMTPMailer COM object, shows how you can assign properties to the mail message you want to send, and how you execute component methods in order to handle mail messages. In the example, form variables are used to provide method parameters and properties, such as the name of the recipient, the desired email address, and so on.

```
<!--- First, create the object --->

<CFOBJECT ACTION="Create"
    NAME="Mailer"
    CLASS="SMTPsvg.Mailer">

<!--- Then, use the form variables from the
user entry form to populate a number of properties
necessary to create and send the message. --->

<CFSET Mailer.FromName = #form.fromname#>
<CFSET Mailer.RemoteHost = #RemoteHost#>
<CFSET Mailer.FromAddress = #form.fromemail#>
<CFSET Mailer.Subject = "Testing CFOBJECT">
<CFSET Mailer.BodyText = "#form.msgbody#">
<CFSET Mailer.SMTPLog = "#logfile#">

<!--- Last, use the AddRecipient and SendMail
methods to finish and send the message along --->

<CFSET Mailer.AddRecipient("#form.fromname#","#form.fromemail#")>
<CFSET success=Mailer.SendMail()>
```

```
<!--- First, create the object --->

<CFOBJECT ACTION="Create"
    NAME="Mailer"
    CLASS="SMTPsvg.Mailer">

<!--- Then, use the form variables from the
user entry form to populate a number of properties
necessary to create and send the message --->

<CFSET Mailer.FromName = #form.fromname#>
<CFSET Mailer.RemoteHost = #RemoteHost#>
<CFSET Mailer.FromAddress = #form.fromemail#>
<CFSET Mailer.Subject = "Testing CFOBJECT">
```

```
<CFSET Mailer.BodyText = "#form.msgbody#">
<CFSET Mailer.SMTPLog = "#logfile#">

<!--- Last, use the AddRecipient and
SendMail methods to finish and send
the message along --->

<CFSET Mailer.AddRecipient("#form.fromname#","#form.fromemail#")>
<CFSET success=Mailer.SendMail()>
```

Getting Started with CORBA Objects

CORBA is the Common Object Request Brokerage Architecture, a specification for a component object system. ColdFusion Enterprise version 4.0 supports CORBA, through the Dynamic Invocation Interface (DII).

ColdFusion Enterprise version 4.0 is bundled with deployment software from VisiBroker for C++ 3.2. These runtime DLLs are used to invoke operations on object references made available using the CFOBJECT tag.

A directory for logging output from VisiBroker is created when you first start ColdFusion Server, Enterprise edition. This directory is called `vbroker\log` and its location is determined as follows:

1. If VisiBroker is already installed on the server, the log directory is the directory pointed to by the `VBROKER_ADM` environment variable.
2. If this is a new VisiBroker installation, the log directory is created on the root of the drive from which ColdFusion Server is started. For example, if ColdFusion is installed in `c:\cfusion` or `opt/coldfusion` (Solaris), then the log directory will be `c:\vbroker\log` or `/vbroker` (Solaris).
3. If the creation of the log directory on the root fails, then the directory is created in the ColdFusion installation directory.

Note User-defined types are not supported (i.e., structures).

Using CFOBJECT to create a CORBA object

In the CFOBJECT tag, several key attributes are required for calling CORBA objects:

- Set the TYPE attribute to CORBA. If no TYPE is specified, COM is assumed.
- The CONTEXT attribute shows how the object reference is obtained. Set the CONTEXT either to "IOR", for a file containing the object’s unique Interoperable Object Reference, or to “NameService”.
- If the CONTEXT attribute is set to IOR, set the CLASS attribute to the file containing the stringified version of the IOR. ColdFusion must be able to read this IOR file at all times, so it should be local to the server or on the network in an open, accessible location.

- If the CONTEXT attribute is set to a NameService, the CLASS attribute must include a period-delimited naming context for the naming service, such as Allaire.Department.Dev.
- Set the NAME attribute to the name your application uses to call the object's operations and attributes.

See the *CFML Language Reference* for information about the CFOBJECT tag as well as examples of CFOBJECT tag creating a CORBA object.

Information about CORBA

To learn more about CORBA, see the Object Management Group's site at http://www.omg.org.

CHAPTER 12

The ColdFusion Extension API

This chapter documents the ColdFusion Extension Application Programming Interface (CFXAPI), which you use to extend ColdFusion using C++.

Contents

- The ColdFusion Extension (CFX) API 124
- The CCFXException Class 125
- The CCFXQuery Class 125
- The CCFXRequest Class 130
- The CCFXStringSet Class 139

The ColdFusion Extension (CFX) API

For information about implementing ColdFusion Extensions (CFX) with C++, see Chapter 11, “Building ColdFusion Extensions,” on page 95.

Class	Members
The CCFXException Class	n/a
The CCFXQuery Class	CCFXQuery::AddRow CCFXQuery::GetColumns CCFXQuery::GetData CCFXQuery::GetName CCFXQuery::GetRowCount CCFXQuery::SetData CCFXQuery::SetQueryString CCFXQuery::SetTotalTime
The CCFXRequest Class	CCFXRequest::AddQuery CCFXRequest::AttributeExists CCFXRequest::CreateStringSet CCFXRequest::Debug CCFXRequest::GetAttribute CCFXRequest::GetAttributeList CCFXRequest::GetCustomData CCFXRequest::GetQuery CCFXRequest::GetSetting CCFXRequest::ReThrowException CCFXRequest::SetCustomData CCFXRequest::SetVariable CCFXRequest::ThrowException CCFXRequest::Write CCFXRequest::WriteDebug
The CCFXStringSet Class	CCFXStringSet::AddString CCFXStringSet::GetCount CCFXStringSet::GetIndexForString CCFXStringSet::GetString

The CCFXException Class

Abstract class that represents an exception thrown during the processing of a ColdFusion Extension (CFX) procedure.

Exceptions of this type can be thrown by the classes CCFXRequest, CCFXQuery, and CCFXStringSet. Your ColdFusion Extension code must therefore be written to handle exceptions of this type. (See the CCFXRequest::ReThrowException tag for more details on doing this correctly.)

The CCFXQuery Class

Abstract class that represents a query used or created by a ColdFusion Extension (CFX). Queries contain 1 or more columns of data that extend over a varying number of rows.

Class members

`virtual int AddRow`

Adds a new row to the query.

`virtual CCFXStringSet* GetColumns`

Retrieves a list of the query's column names.

`virtual LPCSTR GetData( int iRow, int iColumn )`

Retrieves a data element from a row and column of the query.

`virtual LPCSTR GetName`

Retrieves the name of the query.

`virtual int GetRowCount`

Retrieves the number of rows in the query.

`virtual void SetData( int iRow, int iColumn, LPCSTR lpszData )`

Sets a data element within a row and column of the query.

`virtual void SetQueryString( LPCSTR lpszQuery )`

Sets the query string that will displayed along with query debug output.

`virtual void SetTotalTime( DWORD dwMilliseconds )`

Sets the total time that was required to process the query (used for debug output).

CCFXQuery::AddRow

```
int CCFXQuery::AddRow(void)
```

Add a new row to the query. You should call this function each time you want to append a row to the query.

Returns the index of the row that was appended to the query.

Example The following example demonstrates the addition of 2 rows to a query that has 3 columns ('City', 'State', and 'Zip'):

```
// First row
int iRow ;
iRow = pQuery->AddRow() ;
pQuery->SetData( iCity, iRow, "Minneapolis" ) ;
pQuery->SetData( iState, iRow, "MN" ) ;
pQuery->SetData( iZip, iRow, "55345" ) ;

// Second row
iRow = pQuery->AddRow() ;
pQuery->SetData( iCity, iRow, "St. Paul" ) ;
pQuery->SetData( iState, iRow, "MN" ) ;
pQuery->SetData( iZip, iRow, "55105" ) ;
```

CCFXQuery::GetColumns

```
CCFXStringSet* CCFXQuery::GetColumns(void)
```

Retrieves a list of the column names contained in the query.

Returns an object of class CCFXStringSet which contains a list of the columns contained in the query. You are not responsible for freeing the memory allocated for the returned string set (it will be automatically freed by ColdFusion after the request is completed).

Example The following example retrieves the list of columns and then iterates over the list, writing each column name back to the user.

```
// Get the list of columns from the quer

CCFXStringSet* pColumns = pQuery->GetColumns() ;
int nNumColumns = pColumns->GetCount() ;

// Print the list of columns to the user
pRequest->Write( "Columns in query: " ) ;
for( int i=1; i<=nNumColumns; i++ )
{
    pRequest->Write( pColumns->GetString( i ) ) ;
    pRequest->Write( " " ) ;
}
```

CCFXQuery::GetData

LPCSTR CCFXQuery::GetData(int *iRow*, int *lColumn*)

Retrieves a data element from a row and column of the query. Row and column indexes begin with 1. You can determine the number of rows in the query by calling GetRowCount. You can determine the number of columns in the query by retrieving the list of columns using GetColumns and then calling CCFXStringSet::GetCount on the returned string set.

Returns the value of the requested data element.

iRow
Row to retrieve data from (1-based).

lColumn
Column to retrieve data from (1-based).

Example The following example iterates over the elements of a query and writes the data in the query back to the user in a simple, space-delimited format:

```
int iRow, iCol ;
int nNumCols = pQuery->GetColumns()->GetCount() ;
int nNumRows = pQuery->GetRowCount() ;
for ( iRow=1; iRow<=nNumRows; iRow++ )
{
    for ( iCol=1; iCol<=nNumCols; iCol++ )
    {
    pRequest->Write( pQuery->GetData( iRow, iCol ) ) ;
    pRequest->Write( " " ) ;
    }
    pRequest->Write( "<BR>" ) ;
}
```

CCFXQuery::GetName

LPCSTR CCFXQuery::GetName(void)

Retrieves the name of the query. Returns the name of the query.

Example The following example retrieves the name of the query and writes it back to the user:

```
CCFXQuery* pQuery = pRequest->GetQuery() ;
pRequest->Write( "The query name is: " ) ;
pRequest->Write( pQuery->GetName() ) ;
```

CCFXQuery::GetRowCount

`LPCSTR CCFXQuery::GetRowCount(void)`

Retrieves the number of rows in the query. Returns the number of rows contained in the query.

Example The following example retrieves the number of rows in a query and writes it back to the user:

```
CCFXQuery* pQuery = pRequest->GetQuery() ;
char buffOutput[256] ;
wsprintf( buffOutput,
    "The number of rows in the query is %ld.",
    pQuery->GetRowCount() ) ;
pRequest->Write( buffOutput ) ;
```

CCFXQuery::SetData

`void CCFXQuery::SetData(int` ***`iRow`*****`, int`** ***`lColumn`*****`, LPCSTR`** ***`lpszData`*****`)`**

Sets a data element within a row and column of the query. Row and column indexes begin with 1. Before calling SetData for a given row, you should be sure to call AddRow and use the return value as the row index for your call to SetData.

`iRow`
Row of data element to set (1-based).

`lColumn`
Column of data element to set (1-based).

`lpszData`
New value for data element.

Example The following example demonstrates the addition of 2 rows to a query that has 3 columns ('City', 'State', and 'Zip'):

```
// First row
int iRow ;
iRow = pQuery->AddRow() ;
pQuery->SetData( iCity, iRow, "Minneapolis" ) ;
pQuery->SetData( iState, iRow, "MN" ) ;
pQuery->SetData( iZip, iRow, "55345" ) ;

// Second row
```

```
iRow = pQuery->AddRow() ;
pQuery->SetData( iCity, iRow, "St. Paul" ) ;
pQuery->SetData( iState, iRow, "MN" ) ;
pQuery->SetData( iZip, iRow, "55105" ) ;
```

CCFXQuery::SetQueryString

void CCFXQuery::SetQueryString(LPCSTR *lpszQuery***)**

Sets the query string which will displayed along with the query debug output. For queries generated by the DBQUERY tag, this is the SQL statement. For your custom tag, it may be something different, or you may not want to display a query string at all.

lpszQuery
Text of query string.

Example The following example is from a hypothetical custom tag that does directory browsing based on a command string passed to the tag:

```
LPCSTR lpszDirListCommand =
       pRequest->GetAttribute("COMMAND") ;

...Create a query (pQuery) and populate it with the
contents of the directory listing...

pQuery->SetQueryString( lpszDirListCommand ) ;
```

CCFXQuery::SetTotalTime

void CCFXQuery::SetTotalTime(DWORD *dwMilliseconds***)**

Sets the number of milliseconds that were required to process this query. This number will be displayed along with the query debug output.

dwMilliseconds
Execution time in milliseconds.

Example The following example demonstrates the methodology used to set the total time for a query:

```
DWORD dwStartTime = GetCurrentTime() ;

...execute the query and populate it with data...

pQuery->SetTotalTime( GetCurrentTime() - dwStartTime ) ;
```

The CCFXRequest Class

Overview

Abstract class that represents a request made to a ColdFusion Extension (CFX). An instance of this class is passed to the main function of your extension DLL. The class provides several interfaces which may be used by the custom extension, including functions for reading and writing variables, returning output, creating and using queries, and throwing exceptions.

Class Members

`virtual BOOL AttributeExists( LPCSTR lpszName )`

Checks to see whether the attribute was passed to the tag.

`virtual LPCSTR GetAttribute( LPCSTR lpszName )`

Retrieves the value of the passed attribute.

`virtual CCFXStringSet* GetAttributeList()`

Retrieves a list of all attribute names passed to the tag.

`virtual CCFXQuery* GetQuery()`

Retrieves the query that was passed to the tag.

`virtual LPCSTR GetSetting( LPCSTR lpszSettingName )`

Retrieves the value of a custom tag setting.

`virtual void Write( LPCSTR lpszOutput )`

Writes text output back to the user.

`virtual void SetVariable( LPCSTR lpszName, LPCSTR lpszValue )`

Sets a variable in the template that contains this tag.

`virtual CCFXQuery* AddQuery( LPCSTR lpszName, CCFXStringSet* pColumns )`

Adds a query to the template that contains this tag.

`virtual BOOL Debug()`

Checks whether the tag contains the DEBUG attribute.

`virtual void WriteDebug( LPCSTR lpszOutput )`

Writes text output into the debug stream.

`virtual CCFXStringSet* CreateStringSet()`

Allocates and returns a new CCFXStringSet instance.

`virtual void ThrowException( LPCSTR lpszError, LPCSTR lpszDiagnostics )`

Throws an exception and ends processing of this request.

`virtual void ReThrowException( CCFXException* e )`

Re-throws an exception that has been caught.

`virtual void SetCustomData( LPVOID lpvData )`

Sets custom (tag specific) data to carry along with the request.

`virtual LPVOID GetCustomData()`

Gets the custom (tag specific) data for the request.

CCFXRequest::AddQuery

CCFXQuery* CCFXRequest::AddQuery(LPCSTR *lpszName*, **CCFXStringSet*** *pColumns*)

Adds a query to the calling template. This query can then be accessed by DBML tags (e.g., DBOUTPUT or DBTABLE) within the template. Note that after calling AddQuery, the query exists but is empty (i.e., it has 0 rows). To populate the query with data, you should call the CCFXQuery member functions CCFXQuery::AddRow and CCFXQuery::SetData.

Returns a pointer to the query that was added to the template (an object of class CCFXQuery). You are not responsible for freeing the memory allocated for the returned query (it will be automatically freed by ColdFusion after the request is completed).

lpszName
Name of query to add to the template (must be unique).

pColumns
List of columns names to be used in the query.

Example The following example adds a query named 'People' to the calling template. The query has two columns ('FirstName' and 'LastName') and two rows:

```
// Create a string set and add the column names to it
CCFXStringSet* pColumns = pRequest->CreateStringSet() ;
int iFirstName = pColumns->AddString( "FirstName" ) ;
int iLastName = pColumns->AddString( "LastName" ) ;

// Create a query which contains these columns
CCFXQuery* pQuery = pRequest->AddQuery( "People", pColumns ) ;

// Add data to the query
int iRow ;
iRow = pQuery->AddRow() ;
pQuery->SetData( iRow, iFirstName, "John" ) ;
pQuery->SetData( iRow, iLastName, "Smith" ) ;
iRow = pQuery->AddRow() ;
pQuery->SetData( iRow, iFirstName, "Jane" ) ;
pQuery->SetData( iRow, iLastName, "Doe" ) ;
```

CCFXRequest::AttributeExists

`BOOL CCFXRequest::AttributeExists(LPCSTR` *`lpszName`*`)`

Checks to see whether the attribute was passed to the tag. Returns TRUE if the attribute is available; otherwise, returns FALSE.

`lpszName`
Name of the attribute to check (case insensitive).

Example The following example checks to see if the user passed an attribute named DESTINATION to the tag and throws an exception if the attribute was not passed:

```
if ( pRequest->AttributeExists("DESTINATION")==FALSE )
{
    pRequest->ThrowException(
        "Missing DESTINATION parameter",
        "You must pass a DESTINATION parameter in "
        "order for this tag to work correctly." ) ;
}
```

CCFXRequest::CreateStringSet

`CCFXStringSet* CCFXRequest::CreateStringSet(void)`

Allocates and returns a new CCFXStringSet instance. Note that string sets should always be created using this function as opposed to directly using the ’new’ operator.

Returns an object of class CCFXStringSet. You are not responsible for freeing the memory allocated for the returned string set (it will be automatically freed by ColdFusion after the request is completed).

Example The following example creates a string set and adds 3 strings to it:

```
CCFXStringSet* pColors = pRequest->CreateStringSet() ;
pColors->AddString( "Red" ) ;
pColors->AddString( "Green" ) ;
pColors->AddString( "Blue" ) ;
```

CCFXRequest::Debug

BOOL CCFXRequest::Debug(void)

Checks whether the tag contains the DEBUG attribute. You should use this function to determine whether or not you need to write debug information for this request. (See the CCFXRequest::WriteDebug tag for details on writing debug information.)

Returns TRUE if the tag contains the DEBUG attribute; otherwise, returns FALSE.

Example The following example checks to see whether the DEBUG attribute is present, and if it is, it writes a brief debug message:

```
if ( pRequest->Debug() )
{
       pRequest->WriteDebug( "Top secret debug info" ) ;
}
```

CCFXRequest::GetAttribute

`LPCSTR CCFXRequest::GetAttribute(LPCSTR` *`lpszName`***`)`**

Retrieves the value of the passed attribute. Returns an empty string if the attribute does not exist. (Use CCFX:AttributeExists to test whether an attribute was passed to the tag.)

Returns the value of the attribute passed to the tag. If no attribute of that name was passed to the tag, an empty string is returned.

`lpszName`
Name of the attribute to retrieve (case insensitive).

Example The following example retrieves an attribute named DESTINATION and writes its value back to the user:

```
LPCSTR lpszDestination = pRequest->GetAttribute("DESTINATION") ;
pRequest->Write( "The destination is: " ) ;
pRequest->Write( lpszDestination ) ;
```

CCFXRequest::GetAttributeList

`CCFXStringSet*` `CCFXRequest::GetAttributeList(void)`

Retrieves a list of all attribute names passed to the tag. To retrieve the value of an individual attribute, you should use the GetAttribute member function.

Returns an object of class CCFXStringSet that contains a list of all attributes passed to the tag.

You are not responsible for freeing the memory allocated for the returned string set (it will be automatically freed by ColdFusion after the request is completed).

Example The following example retrieves the list of attributes and then iterates over the list, writing each attribute and its value back to the user.

```
LPCSTR lpszName, lpszValue ;
CCFXStringSet* pAttribs = pRequest->GetAttributeList() ;
int nNumAttribs = pAttribs->GetCount() ;

for( int i=1; i<=nNumAttribs; i++ )
```

```
{
        lpszName = pAttribs->GetString( i ) ;
        lpszValue = pRequest->GetAttribute( lpszName ) ;
        pRequest->Write( lpszName ) ;
        pRequest->Write( " = " ) ;
        pRequest->Write( lpszValue ) ;
        pRequest->Write( "<BR>" ) ;
}
```

CCFXRequest::GetCustomData

`LPVOID CCFXRequest::GetCustomData(void)`

Gets the custom (tag specific) data for the request. This member is typically used from within subroutines of your tag implementation to extract tag specific data from within the request.

Returns a pointer to the custom data or returns NULL if no custom data has been set during this request using SetCustomData.

Example The following example retrieves a pointer to a request specific data structure of hypothetical type MYTAGDATA:

```
void DoSomeGruntWork( CCFXRequest* pRequest )
{
        MYTAGDATA* pTagData =
            (MYTAGDATA*)pRequest->GetCustomData() ;

        ... remainder of procedure ...
}
```

CCFXRequest::GetQuery

`CCFXQuery* CCFXRequest::GetQuery(void)`

Retrieves the query that was passed to the tag. To pass a query to a custom tag, you use the QUERY attribute. This attribute should be set to the name of an existing query (created using the DBQUERY tag or another custom tag). The QUERY attribute is optional and should only be used by tags that need to process an existing dataset.

Returns an object of class CCFXQuery that represents the query that was passed to the tag. If no query was passed to the tag, NULL is returned. You are not responsible for freeing the memory allocated for the returned query (it will be automatically freed by ColdFusion after the request is completed).

Example The following example retrieves the query which was passed to the tag. If no query was passed , an exception is thrown:

```
CCFXQuery* pQuery = pRequest->GetQuery() ;
if ( pQuery == NULL )
{
        pRequest->ThrowException(
            "Missing QUERY parameter",
            "You must pass a QUERY parameter in "
            "order for this tag to work correctly." ) ;
}
```

CCFXRequest::GetSetting

LPCSTR CCFXRequest::GetSetting(LPCSTR ***lpszSettingName*****)**

Retrieves the value of a global custom tag setting. Custom tag settings are stored within the CustomTags section of the ColdFusion Registry key.

Returns the value of the custom tag setting. If no setting of that name exists, an empty string is returned.

lpszSettingName
Name of the setting to retrieve (case insensitive).

Example The following example retrieves the value of a setting named 'VerifyAddress' and uses the returned value to determine what actions to take next:

```
LPCSTR lpszVerify = pRequest->GetSetting("VerifyAddress") ;
BOOL bVerify = atoi(lpszVerify) ;
if ( bVerify == TRUE )
{
        // Do address verification...
}
```

CCFXRequest::ReThrowException

void CCFXRequest::ReThrowException(CCFXException* ***e*****)**

Re-throws an exception that has been caught within an extension procedure. This function is used to avoid having C++ exceptions thrown by DLL extension code propagate back into ColdFusion. You should catch ALL C++ exceptions that occur in your extension code and then either re-throw them (if they are of the CCFXException class) or create and throw a new exception using ThrowException.

e
An existing CCFXException that has been caught.

Example The following code demonstrates the correct way to handle exceptions in ColdFusion Extension DLL procedures:

```
try
{

        ...Code which could throw an exception...

}
catch( CCFXException* e )
{
        ...Do appropriate resource cleanup here...

        // Re-throw the exception
        pRequest->ReThrowException( e ) ;
}
catch( ... )
{
        // Something nasty happened, don't even try
        // to do resource cleanup

        pRequest->ThrowException(
            "Unexpected error occurred in CFX tag", "" ) ;
}
```

CCFXRequest::SetCustomData

`void CCFXRequest::SetCustomData(LPVOID` *`lpvData`***`)`**

Sets custom (tag specific) data to carry along with the request. You should use this function to store request specific data that you want to pass along to procedures within your custom tag implementation.

`lpvData`
Pointer to custom data.

Example The following example creates a request-specific data structure of hypothetical type MYTAGDATA and stores a pointer to the structure in the request for future use:

```
void ProcessTagRequest( CCFXRequest* pRequest )
{
        try
        {
            MYTAGDATA tagData ;
            pRequest->SetCustomData( (LPVOID)&tagData ) ;

        ... remainder of procedure ...
}
```

CCFXRequest::SetVariable

`void CCFXRequest::SetVariable(LPCSTR` *`lpszName`***`, LPCSTR`** *`lpszValue`***`)`**

Sets a variable in the calling template. If the variable name specified already exists in the template, its value is replaced. If it does not already exist, a new variable is created. The values of variables created using SetVariable can be accessed in the same manner as other template variables (e.g., #MessageSent#).

`lpszName`
Name of variable.

`lpszValue`
Value of variable.

Example The following example sets the value of a variable named ’MessageSent’ based on the success of an operation performed by the custom tag:

```
BOOL bMessageSent ;

...attempt to send the message...

if ( bMessageSent == TRUE )
{
       pRequest->SetVariable( "MessageSent", "Yes" ) ;
}
else
{
       pRequest->SetVariable( "MessageSent", "No" ) ;
}
```

CCFXRequest::ThrowException

`void CCFXRequest::ThrowException(LPCSTR` *`lpszError`***`,`**
`LPCSTR` *`lpszDiagnostics`***`)`**

Throws an exception and ends processing of this request. You should call this function when you encounter an error that does not allow you to continue processing the request. Note that this function is almost always combined with the ReThrowException member function to provide protection against resource leaks in extension code.

`lpszError`
Short identifier for error.

`lpszDiagnostics`
Error diagnostic information.

Example The following example throws an exception indicating that an unexpected error occurred while processing the request:

```
char buffError[512] ;
wsprintf( buffError,
        "Unexpected Windows NT error number %ld "
        "occurred while processing request.", GetLastError() ) ;

pRequest->ThrowException( "Error occurred", buffError ) ;
```

CCFXRequest::Write

void CCFXRequest::Write(LPCSTR *lpszOutput***)**

Writes text output back to the user.

lpszOutput
Text to output.

Example The following example creates a buffer to hold an output string, fills the buffer with data, and then writes the output back to the user:

```
CHAR buffOutput[1024] ;
wsprintf( buffOutput, "The destination is: %s",
              pRequest->GetAttribute("DESTINATION") ) ;
pRequest->Write( buffOutput ) ;
```

CCFXRequest::WriteDebug

void CCFXRequest::WriteDebug(LPCSTR *lpszOutput***)**

Writes text output into the debug stream. This text is only displayed to the end-user if the tag contains the DEBUG attribute. (For more information, see the Debug member function.)

lpszOutput
Text to output.

Example The following example checks to see whether the DEBUG attribute is present, and if it is, it writes a brief debug message:

```
if ( pRequest->Debug() )
{
        pRequest->WriteDebug( "Top secret debug info" ) ;
}
```

The CCFXStringSet Class

Overview

Abstract class that represents a set of ordered strings. Strings can be added to a set and can be retrieved by a numeric index (the index values for strings are 1-based). To create a string set, you should use the CCFXRequest member function CCFXRequest::CreateStringSet.

Class members

`virtual int AddString( LPCSTR lpszString )`

Adds a string to the end of the list.

`virtual int GetCount()`

Gets the number of strings contained in the list.

`virtual LPCSTR GetString( int iIndex )`

Gets the string located at the passed index.

`virtual int GetIndexForString( LPCSTR lpszString )`

Gets the index for the passed string.

CCFXStringSet::AddString

`int CCFXStringSet::AddString(LPCSTR` *`lpszString`***`)`**

Adds a string to the end of the list. Returns the index of the string that was added.

`lpszString`
String to add to the list.

Example The following example demonstrates adding three strings to a string set and saving the indexes of the items that are added:

```
CCFXStringSet* pSet = pRequest->CreateStringSet() ;
int iRed = pSet->AddString( "Red" ) ;
int iGreen = pSet->AddString( "Green" ) ;
int iBlue = pSet->AddString( "Blue" ) ;
```

CCFXStringSet::GetCount

`int CCFXStringSet::GetCount(void)`

Gets the number of strings contained in the string set. This value can be used along with the GetString function to iterate over the strings in the set (when iterating, remember that the index values for strings in the list begin at 1).

Returns the number of strings contained in the string set.

Example The following example demonstrates using GetCount along with GetString to iterate over a string set and write the contents of the list back to the user:

```
int nNumItems = pStringSet->GetCount() ;
for ( int i=1; i<=nNumItems; i++ )
{
        pRequest->Write( pStringSet->GetString( i ) ) ;
        pRequest->Write( "<BR>" ) ;
}
```

CCFXStringSet::GetIndexForString

`int CCFXStringSet::GetIndexForString(LPCSTR` ***`lpszString`*****`)`**

Does a case insensitive search for the passed string.

If the string is found, its index within the string set is returned. If it is not found, the constant CFX_STRING_NOT_FOUND is returned.

`lpszString`
String to search for.

Example The following example illustrates searching for a string and throwing an exception if it is not found:

```
CCFXStringSet* pAttribs = pRequest->GetAttributeList() ;

int iDestination =
        pAttribs->GetIndexForString("DESTINATION") ;
if ( iDestination == CFX_STRING_NOT_FOUND )
{
        pRequest->ThrowException(
            "DESTINATION attribute not found."
            "The DESTINATION attribute is required "
            "by this tag." ) ;
}
```

CCFXStringSet::GetString

`LPCSTR CCFXStringSet::GetString(int` ***`iIndex`*****`)`**

Retrieves the string located at the passed index (note that index values are 1-based).

Returns the string located at the passed index.

`iIndex`
Index of string to retrieve.

Example The following example demonstrates using GetString along with GetCount to iterate over a string set and write the contents of the list back to the user:

```
int nNumItems = pStringSet->GetCount() ;
for ( int i=1; i<=nNumItems; i++ )
{
        pRequest->Write( pStringSet->GetString( i ) ) ;
        pRequest->Write( "<BR>" ) ;
}
```

CHAPTER 13

Connecting to LDAP Directories

Support for the Lightweight Directory Access Protocol (LDAP) API in CFML is part of Allaire's commitment to open networking standards.

Contents

- ColdFusion Support for LDAP 144
- Directory Structures 144
- Key Terms 146
- Operations 147
- Search Filters 148
- Examples 148

ColdFusion Support for LDAP

The CFLDAP tag extends ColdFusion's query capabilities to TCP network directory services. CFLDAP offers developers significant opportunities in several areas:

- Create Internet White Pages for users to easily locate people and resources and to receive information about them. Selected ODBC data (names, contact information, etc.) can be copied to an LDAP server.
- Provide a front end to manage and update directory entries.
- Build applications that incorporate data from directory queries in their processes.

As its name implies, LDAP is the lightweight version of the X.500 Directory Access Protocol. It fills a need for access to directory structures that is less complex and demanding of system resources than DAP. The development of an LDAP server in the Internet environment enables direct access to directories without the requirement to query X.500 servers, though this remains an option. LDAP retains the majority of DAP functionality and is a major step forward in the evolution of the global directory service envisioned by X.500.

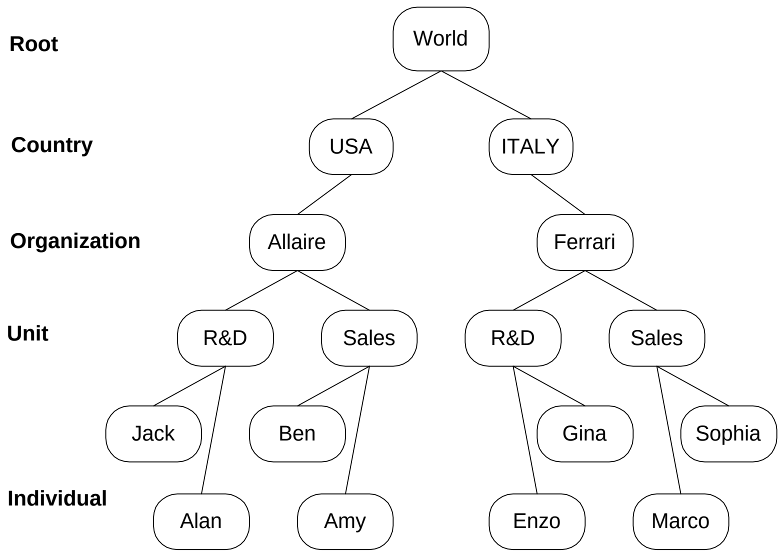
References

Extensions to the LDAP protocol are ongoing and its wide support in the Internet community is growing. Additional material on LDAP is available from these sources:

- The LDAP specification was originally developed at the University of Michigan. Their site http://www.umich.edu/~dirsvcs/ldap/index.html contains a wealth of information and resources.
- The stated purpose of the Internet Engineering Task Force LDAP Extensions Working Group is to "...define and standardize extensions to the LDAP version 3 protocol and extensions to the use of LDAP on the Internet." Their site is at http://www.ietf.org/html.charters/ldapext-charter.html.
- The Directory Enabled Networks (DEN) specification, based on LDAP, is under development by a number of vendors, including Microsoft and Cisco Systems. You can follow the progress of this proposed standard at the DEN Ad Hoc Working Group site at http://murchiso.com/den/.

Directory Structures

An LDAP directory is usually a hierarchical structure, though this is not a requirement. LDAP supports a flat, or one-level, structure as readily as multiple levels. The illustration below shows a simplified tree of entries from the root level to the individual level.



The complexity and flexibility allowed in this structure is a key to LDAP's success. A directory's structure abstracts the structure of the organization it represents. Properly devising and maintaining this structure is the LDAP server administrator's responsibility. The type, quantity, and accessibility of the information for individual entries will obviously vary widely across organizations and their LDAP servers.

A ColdFusion application developed for an organization's intranet could easily include LDAP query and output capability from its internal LDAP server and from allied servers. Changes in the directory structure would, presumably, be updated in the application code. Venturing into the wider world of the Internet needs special attention, though. Communication with data source administrators is as important in LDAP implementations as it is in other data-driven applications.

Viewing directory schema

Currently, you cannot use CFLDAP to determine the attributes of an LDAP data source. The syntax requires the distinguished name of an entry to initiate a query. In other words, the user must supply the starting point for a search. Full support of the LDAP 3.0 standard will be enabled in a future ColdFusion release.

As a ColdFusion developer, you must do the work of providing that starting point for your users or for an LDAP query you run internally. The more focus you can provide the user, the more effective the search.

LDAP attributes

Following is a list of the common attributes:

Common LDAP Attributes	
Attribute	**Name**
c	country
st	state or province
l	locality
o	organization
ou	organizational unit
cn	common name
sn	surname

Key Terms

Following is a brief description of the LDAP information structure.

Entry

The basic information object of LDAP is the entry. An entry is composed of attributes, each of which has a type defining what information can be contained in the attribute's values and what behaviors the attribute exhibits during processing. Entries are subject to content rules that specify its required and optional attributes. Content rules can be defined in the syntax or on the LDAP server.

Distinguished name

A naming convention for LDAP entries ensures compliance with the protocol regardless of the complexity of directory trees. LDAP name syntax begins at the entry level and specifies each level up to the root. In other words, it proceeds from the individual to the global. The Distinguished Name of an entry locates it in the directory tree. Each Distinguished Name (DN) is made up of Relative Distinguished Names (RDN) that contain one or more of the entry's attributes. As with file systems pathnames and URLs, entering the correct LDAP name format is essential to successful search operations.

Scope

Sets the limits of a search from the starting point of a query. The default is one level below the distinguished name specified in the Start attribute. If, for example, the Start attribute is "ou=support, o=allaire" the level below "support" is searched. You can optionally restrict a query to the level of the Start entry or extend it to the entire subtree.

Referral

While not supported directly in the LDAP2 standard, the ability of an LDAP server to refer a client query to another server is an attractive feature and has been implemented in the Netscape and University of Michigan servers. ColdFusion developers need to be aware of the possibilities for referrals when designing their query forms.

Operations

An LDAP directory is a database with a limited and specific role in an organization. It offers performance advantages over conventional databases, and its operations are familiar to database users.

Security

You can restrict access to CFLDAP operations by setting the user name and password attributes. You could, for instance, allow queries by all users but limit update operations to qualified users.

Query

CFLDAP implements the extensive search parameters of the LDAP API. You can develop meaningful forms-based pages that focus the user's search by controlling the tag's attributes. The syntax permits a high level of control of search criteria via the filter attribute.

Output

Query results can be sorted and returned to the browser or they can be further processed by CFOUTPUT, CFREPORT, and related tags.

Update

Entries can be added, modified, and deleted. Remote administration of an LDAP server is one possible use of these options.

Search Filters

A search string of the form *attribute operator value* defines the filter syntax. The default filter, objectclass=*, returns all entries for the attribute.

The following table lists the filter operators. Note the prefix notation for the Boolean operators.

CFLDAP Filter Operators

Operator	Example
=	o=allaire - organization name equals allaire
~=	o~=alliare - organization name approximates allaire
>=	st>=ma - names appearing after "ma" in an alphabetical state attribute list
<=	st<=ma - names appearing before "ma" in an alphabetical state attribute list
*	o=alla* - organization names starting with "alla" o=*aire - organization names ending with "aire" o=all*aire - organization names starting with "all and " ending with "aire"
&	(&(o=allaire)(co=usa)) - organization name = "allaire" AND country = "usa"
|	(|(o=allaire)(sn=allaire)) - organization name = "allaire" OR surname = "allaire"
!	(!(STREET=*)) - all entries that do NOT contain a StreetAddress attribute

Although sophisticated search criteria can be constructed from these filter operators, performance may degrade if the LDAP server is slow to process the synchronous search routines supported by CFLDAP. The TIMEOUT and MAXROWS attributes can be used to control query performance.

Examples

The Query sample code can be copied to a ColdFusion application page and tested. The code examples for the Delete and ModifyDN actions use placeholders for the `SERVER`, `USERNAME`, and `PASSWORD` values because these must be specified by the user.

Example: Action="Query"

This example uses CFLDAP to retrieve the name and telephone numbers for US organizations with a common name that starts with ’A’ through ’E’. The search starts in the country: US. The filter is a regular expression that limits the search to expressions of any length that begin with "A," "B," "C," "D," or "E."

```
<CFLDAP NAME="OrgList"
    SERVER="ldap.itd.umich.edu"
    ACTION="QUERY"
    ATTRIBUTES="o,st,telephoneNumber"
    SCOPE="ONELEVEL"
    FILTER="(|(o=A*)(o=B*)(o=C*)(o=D*)(o=E*))"
    MAXROWS=200
    SORT="o"
    START="c=US">

<HTML>
<HEAD>
    <TITLE>LDAP Directory Example</TITLE>
</HEAD>

<BODY>

<H3>US Organizations begining with
    the letter ’A’ thru ’E’:</H3>

<CFFORM NAME="GridForm" ACTION="org_query.cfm">

    <CFGRID NAME="grid_one"
        QUERY="OrgList"
        HEIGHT=250
        WIDTH=620
        HSPACE=20
        VSPACE="6">

        <CFGRIDCOLUMN NAME="o"
            HEADER="Organization" WIDTH=380>
        <CFGRIDCOLUMN NAME="st"
            HEADER="State" WIDTH=100>
        <CFGRIDCOLUMN NAME="telephoneNumber"
            HEADER="Phone ##" WIDTH=150>
    </CFGRID>

</CFFORM>

</BODY>
</HTML>
```

Example: Action= "Delete"

This example executes a Delete based on the user selection, and then performs a query of the LDAP data source.

```
<!--- If the delete parameter is sent
then run this update --->
<CFIF IsDefined(dn)>
    <CFLDAP Name="LDAPDelete"
        SERVER="ldap.com"
        USERNAME="cn=Directory Manager,
            o=Ace Industry, c=US"
        PASSWORD="testldap"
        ACTION="Delete"
        DN=#dn#>
</CFIF>

<!--- Use CFLDAP to retrieve the common name
and distinguished name for all employees that
have a surname that contains ens and a common
name that is > K. Search starts in the country
US and organization Ace Industry. --->

<CFLDAP Name="EntryList"
    SERVER="ldap.com"
    ACTION="Query"
    ATTRIBUTES="dn,cn, sn"
    SCOPE="SUBTREE"
    SORT="cn ASC"
    FILTER="(cn>=A)"
    START="o=Ace Industry, c=US"
    TIMEOUT=30>
```

Example: Action="ModifyDN"

This code determines whether an insert or an update to an entry in an LDAP data source was requested and executes an LDAP operation accordingly. Output is directed to pages that populate forms with data returned in the LDAP operation.

```
<!--- If the update parameter is sent
    then run this update --->
<!--- If the insert parameter is sent
    then run this insert --->

<CFIF IsDefined(rename_dn)>

    <CFLDAP Name="CustomerRename"
        SERVER="ldap.com"
        USERNAME="cn=Directory Manager,
            o=Ace Industry, c=US"
        PASSWORD="testldap"
        ACTION="MODIFYDN"
        ATTRIBUTES=#new_dn#
        DN=#rename_dn#>

<CFELSE>

    <CFIF IsDefined(dn)>
    <CFSET #UPDATE_ATTRS#=#mailtag# & #email# & ";" &
```

```
        #phonetag# & #Phone#>

        <CFLDAP Name="CustomerModify"
            SERVER="ldap.com"
            USERNAME="cn=Directory Manager,
                o=Ace Industry, c=US"
            PASSWORD="testldap"
            ACTION="MODIFY"
            ATTRIBUTES=#UPDATE_ATTRS#
            DN=#dn#>

<CFELSE>

<!--- If the insert parameter is sent
    then run this insert --->

    <CFIF IsDefined(Distinguished_Name)>
    <CFSET #ADD_ATTRS# = "objectclass=top,
        person,organizationalPerson,inetOrgPerson;" &
        #fullnametag# &
        #Fullname# &
        ";" &
        #surnametag# &
        #Surname# &
        ";" &
        #mailtag# &
        #Email# &
        ";" &
        #phonetag# &
        #Phone#>

        <CFLDAP Name="CustomerAdd"
            SERVER="ldap.com"
            USERNAME="cn=Directory Manager,
                o=Ace Industry, c=US"
            PASSWORD="testldap"
            ACTION="Add"
            ATTRIBUTES=#ADD_ATTRS#
            DN=#Distinguished_Name#>

    </CFIF>
    </CFIF>
</CFIF>

<!--- Use CFLDAP to retrieve the common
name and distinguished name for all employees
that have a surname that contains ens and a common
name that is > K. Search starts in the country US
and organization Ace Industry.--->

<CFLDAP Name="EntryList"
    SERVER="ldap.com"
    ACTION="Query"
    ATTRIBUTES="dn,cn, sn"
    SCOPE="SUBTREE"
    SORT="sn ASC"
    FILTER="(&(sn=*ens*)(cn>=K))"
    START="o=Ace Industry, c=US"
    MAXROWS=50
    TIMEOUT=30>

<HTML>
<HEAD>
    <TITLE>LDAP Directory Example</TITLE>
</HEAD>

<P>To modify the attributes of an entry,
select the entry and click the <B>Update</B>
button. To create a new entry, click the
<B>Add</B> button.

<CFFORM NAME="MyForm"
    ACTION="ldap_update.cfm"
    TARGET="Lower">

    <CFSELECT NAME="dn"
        SIZE="5"
        REQUIRED="Yes"
        QUERY="EntryList"
        Value="dn"
        Display="cn">
    </CFSELECT>

    <INPUT TYPE="Submit" VALUE="Update...">

</CFFORM>

<FORM ACTION="ldap_add.cfm"
    METHOD="Post"
    TARGET="Lower">

    <INPUT TYPE="Submit" VALUE="Add...">
</FORM>

</BODY>
</HTML>
```

CHAPTER 14

Application Security

ColdFusion 4.0 for Windows NT supports several levels of Advanced Security. This chapter describes how to deploy user security, which is controlled by the ColdFusion developer and offers runtime user security. It also describes the Remote Development Services security feature, where developers accessing server resources through ColdFusion Studio are authenticated before receiving access to protected resources.

For information on setting up security elements or using Administrator-controlled security features, see *Administering ColdFusion Server*.

Contents

- ColdFusion Security Features 154
- Remote Development Services (RDS) Security........ 154
- Overview of User Security 155
- Using Advanced Security in Application Pages 156
- Example of User Authentication and Authorization 160

ColdFusion Security Features

Security options in ColdFusion have been greatly enhanced in this release. ColdFusion Server now supports several levels of Advanced Security:

- **Remote Development Services Security (RDS)** — Developers accessing server resources through ColdFusion Studio can be authenticated before receiving access to protected resources.
- **User security** — Implemented in ColdFusion application pages by the ColdFusion developer, User Security offers runtime user authentication and authorization.
- **Server sandbox security** — Controlled by the ColdFusion administrator of a hosted site, offers runtime security based on directory access at hosted sites (ColdFusion Enterprise only).
- **Administrator security** — Individual administrative operations can be secured against unauthorized access.

This chapter describes User Security and Remote Development Services (RDS) security. For more information on the Sandbox and Administrator security features, see *Administering ColdFusion Server*.

Note Advanced security is not currently supported in ColdFusion Server for Solaris.

Remote Development Services (RDS) Security

ColdFusion RDS security provides security services to developers working in ColdFusion Studio. RDS security is at the core of the security framework in a team-oriented ColdFusion development environment where groups of developers, working in ColdFusion Studio, require different levels of access to ColdFusion files and data sources.

Working in ColdFusion Studio, developers access these ColdFusion resources remotely, opening *.cfm files or accessing data sources. RDS security authenticates users and grants them access only to the resources appropriate to their login. Authentication is carried out against the NT domain server or an LDAP directory specified in the Administrator as part of a security context.

RDS and Basic security

In addition to Advanced security and debugging, RDS security also provides basic security for ColdFusion. Access to RDS for Basic security is enabled by specifying a ColdFusion Studio password on the Administrator Basic Security page. RDS secures ColdFusion Server data sources and files, and enables file browsing and debugging as well.

To access these resources, developers in ColdFusion Studio must supply a password which, when authenticated, permits access to RDS Services: file browsing, editing, database operations, debugging, and so on.

For more information see the Configuring Basic Security chapter in the *Administering ColdFusion Server* book.

Configuring RDS security

A ColdFusion Administrator implements RDS security, so that when developers attempt to access protected resources, they must provide a username and password.

When developers working in ColdFusion Studio connect to the ColdFusion Server and attempt to access remote servers, files or data sources, access is granted according to the rules and policies associated with their logins.

For more information see the *Administering ColdFusion Server* book.

Overview of User Security

The advanced User Security feature allows ColdFusion developers to authenticate users and match protected resources with authorized users in ColdFusion application pages.

The User Security feature is composed of the following elements:

Advanced Security Concepts

Term	Description
Security contexts	At the top level of the security hierarchy, the security context is a kind of container in which rules, policies, and users are referenced.
Security rules	You use rules to define the access restrictions you want for a particular ColdFusion resource, such as defining which SQL statements are allowed to be executed against a specific data source or which CFML tag ACTIONS are restricted.
Users/groups	Individual users and groups are authenticated within a particular domain. A security directory can be a specified Windows NT domain or an LDAP directory.
User directories	Defines the mechanism to use when authenticating users. Available mechanisms are: a Windows NT domain, which authenticates users with accounts on the server you specify; an LDAP directory that stores user and group account information.

Advanced Security Concepts (Continued)

Term	Description
Security policies	A policy associates specific users or groups with privileges to a set of restricted resources that these users have access to. These restrictions are in the form of rules, such as allowing a particular user or group to execute a SQL UPDATE on a particular data source.
ColdFusion resources	ColdFusion resources include data sources, Verity collections, ColdFusion tags, custom tags, specific files, and so on.
Security server	A hostname or IP address you specify where the security authentication and authorization services run and is used to authenticate individual users or groups.
Security sandboxes	A security framework established by applying a particular security context, with all that it contains, to a directory structure. Intended mainly to help ISPs hosting ColdFusion applications to partition application pages in individually secure areas.

Implementation summary

To implement runtime user security for applications, you use the ColdFusion Administrator to

- Set up the security server.
- Create a security context for your application.
- Set up rules and policies that match secured resources with authorized users.

After the security framework is in place, you use the CFAUTHENTICATE tag in individual application pages (or the Application.cfm file) to authenticate users. The IsAuthenticated and IsAuthorized functions enable developers to offer or deny access based on the established security policies.

See the Example of User Authentication and Authorization in this chapter to see code examples that show how this works.

Using Advanced Security in Application Pages

After you set up the security context, rules, and policies for your application, you can use security in application pages. This section describes how developers use security tags and functions to authenticate users and provide or withhold resources according to the security context's rules.

- You can use CFAUTHENTICATE on any application page, or on the Application.cfm file for your application, to authenticate users (in other words, to make sure they are who they say they are, and are allowed to use this security context). Pass this information to subsequent pages, where you can test for authentication.

ColdFusion sets a cookie, CFAUTH, to contain authentication information. If you choose not to use this cookie, you must check authentication for each request.

- The IsAuthenticated function checks to see if the current user is authenticated.
- The IsAuthorized function checks to see if the user is authorized for certain resources.

Encrypting application pages

You can encrypt strings using the Encrypt and Decrypt functions. See the *CFML Language Reference* for descriptions of these functions.

CFAUTHENTICATE syntax

The CFAUTHENTICATE tag has several required attributes:

- SECURITYCONTEXT— Describes which security context to use for authentication and authorization. This name matches the security context as defined in the Advanced Security page of the Administrator.
- USERNAME — The username required to access the protected resources.
- PASSWORD — The password required to access the protected resources.

The USERNAME and PASSWORD are usually variables passed in a cookie from form fields on a secure login page for the current session.

In addition, CFAUTHENTICATE has two optional attributes:

- SETCOOKIE — Indicates whether ColdFusion sets a cookie to contain authentication information. This cookie is encrypted and includes the user name, security context, browser remote address, and the http user agent. Default is Yes.
- THROWONFAILURE — Indicates whether ColdFusion throws an exception of type Security if authentication fails. Default is Yes.

Example

```
<CFAUTHENTICATE SECURITYCONTEXT=”SecurityContextName”
    USERNAME=#userID#
    PASSWORD=#pwd#>
```

If the user has not already been defined in the system, a ColdFusion Security exception is thrown. You can either reject access to the resource or re-route the user to a login page. For example, you can display a login form and then pass the user along to the originally-requested page.

For information on exception handling strategies in ColdFusion, see Chapter 9, “Structured Exception Handling,” on page 83.

See the *CFML Language Reference* for a full description of the CFAUTHENTICATE tag.

Authentication and Authorization functions

After using CFAUTHENTICATE to check if the user is defined for the security context, you can use two security functions:

- IsAuthenticated checks to see if the current session has been authenticated by the CFAUTHENTICATE tag.
- IsAuthorized checks whether the authenticated user has access to the named resource, based on rules defined in the security context.

IsAuthenticated Syntax

The IsAuthenticated function returns TRUE if the user has been authenticated for the current request; otherwise, it returns FALSE.

The IsAuthenticated function does not take any parameters. Instead it checks whether a CFAUTHENTICATE tag has been successfully executed for the current request. If not, if looks for the CFAUTH cookie to determine if the user is authenticated or not.

If you choose not to set a cookie in CFAUTHENTICATE (by specifying SETCOOKIE="No" in CFAUTHENTICATE), you must call CFAUTHENTICATE for every request in the application.

IsAuthorized Syntax

Once a user is authenticated, you can use the IsAuthorized function to check which resources the user is allowed to access.

IsAuthorized returns TRUE if the user is authorized to perform the specified action on the specified ColdFusion resource. IsAuthorized takes three parameters:

```
IsAuthorized(ResourceType, ResourceName, [ResourceAction])
```

For example, to check whether the authenticated user is authorized to update a datasource resource called orders, use this syntax:

```
IsAuthorized("Datasource", "orders", "update")
```

The IsAuthorized function returns TRUE if the user is authorized for the named Datasource, or if the Datasource is not protected in the security context.

Note The ColdFusion server only checks to see if a user is authorized when a developer specifically requests it with the IsAuthorized function. It is up to the developer to decide what action to take based on the results of the IsAuthorized call.

See the *CFML Language Reference* for full descriptions of the IsAuthorized and IsAuthenticated functions.

Catching security exceptions

You can use the structured exception handling tags, CFTRY and CFCATCH, to catch security exceptions. Setting the TYPE attribute in CFCATCH to “Security” enables you to catch failures in the CFAUTHENTICATE tag. You can also catch catastrophic failures from the IsAuthorized or IsAuthenticated functions.

Set the THROWONFAILURE attribute to Yes and enclose the CFAUTHENTICATE tag in a CFTRY/CFCATCH block if you want to handle possible exceptions programmatically.

For information on exception handling strategies in ColdFusion, see Chapter 9, “Structured Exception Handling,” on page 83.

Example

```
<!--- This exaple shows the use of excpetion handling
    with CFAUTHENTICATE in an Application.cfm file --->
<HTML>
<HEAD>
    <TITLE>CFAUTHENTICATE Example</TITLE>
</HEAD>

<BODY>
<H3>CFAUTHENTICATE Example></H3>

<P>The CFAUTHENTICATE tag authenticates a user and
sets the security context for an application.

<P>Code this tag in the Application.cfm file to set a
security context for your application.

<P>If the user has not already been defined in the
system, you can either reject the page, request that
the user respecify the username and password, or define
a new user.

<!--- This code is from an Application.cfm file --->

<CFTRY>

    <CFAUTHENTICATE SECURITYCONTEXT=”Allaire”
        USERNAME=#user#
        PASSWORD=#pwd#>
    <CFCATCH TYPE=”Security”>
        <!--- The message to display --->
        <H3>Authentication error</H3>
        <CFOUTPUT>
<--- Display the message. Alternatively,
    you might place code here to define the
    user to the security context. --->
        <P>#CFCATCH.Message#
        </CFOUTPUT>
    </CFCATCH>
</CFTRY>

<CFAPPLICATION NAME=”Personnel”>

</BODY>
</HTML>
```

Example of User Authentication and Authorization

The following sample pages illustrate how a developer might implement user security by authenticating users and then allowing users to see/use only the resources they are authorized to use.

In this example, a user requests a page in an application named Orders, which is part of a security context, also named Orders, that governs pages and resources for an order tracking application.

User security is generally handled in two steps:

- First, the Application.cfm page checks to see if the current user is *authenticated*. If not, we present a login form and the user must submit a username and password for authentication.

  If a user passes the authentication test, ColdFusion passes a cookie to carry the user's authentication state to subsequent application pages governed by this Application.cfm page.

- Next, only authenticated users are able to access the requested application page, for selecting and updating customer orders in a database. This page checks to see which resources the authenticated user is *authorized* to see and use.

Authenticating users in Application.cfm

This example code for an Application.cfm page checks first to see whether the current user is authenticated by checking to see if a login form was submitted. If the username and password can be authenticated for the current security context, the user passes through and the requested page is served.

If the Application.cfm page does not receive the user's login information from the previous page, it prompts the user to provide a username and password. The user's response is checked against the list of valid users defined for the current security context.

If the user passes the authentication step too, the requested page appears. We use the CGI variables *script_name* and *query_string* keep track of the page originally requested. This way, once users are authenticated, we can serve the page they originally requested.

All pages governed by this Application.cfm page — those in the same directory as Application.cfm and in its sub-tree — will invoke this authentication test.

Note To use this code in your own Application.cfm page, change the application name and security context name to match your application and security names.

Example: Application.cfm

```
<CFAPPLICATION NAME="Orders">


<CFIF not IsAuthenticated()>
    <!--- The user is not authenticated --->

    <CFSET showLogin = "No">
<CFIF IsDefined("form.username") and
    IsDefined("form.password")>

<!--- The login form was submitted --->
<CFTRY>
    <CFAUTHENTICATE SecurityContext="Orders"
        username="#form.username#"
        password="#form.password#"
        setCookie="YES">

<CFCATCH TYPE="security">
<!--- Security error in login occurred,
    show login again --->
    <H3>Invalid Login</H3>
    <CFSET showLogin = "Yes">
</CFCATCH>
</CFTRY>

<CFELSE>
<!--- The login was not detected --->
    <CFSET showLogin = "Yes">
</CFIF>

<CFIF showLogin>
<!--- Recreate the url used to call this template --->
    <CFSET url = "#cgi.script_name#">
<CFIF cgi.query_string is not "">
    <CFSET url = url & "?#cgi.query_string#">
</CFIF>

<!--- Populate the login with the recreated url --->

<CFOUTPUT>
    <FORM ACTION="#url#" METHOD="Post">
    <TABLE>
    <TR>
    <TD>username:</TD>
    <TD><INPUT TYPE="text" NAME="username"></TD>
    </TR>
    <TR>
    <TD>password:</TD>
    <TD><INPUT TYPE="password" NAME="password"></TD>
    </TR>
    </TABLE>
    <INPUT TYPE="submit" VALUE="Login">

    </FORM>
</CFOUTPUT>
<CFABORT>
</CFIF>

</CFIF>
```

Checking for Authentication and Authorization

Inside application pages, developers can use the IsAuthorized function to check whether an authenticated user is authorized to access the protected resources, and then display only the authorized resources.

The following sample page appears to users who pass the authentication test in the Application.cfm page above. It uses the IsAuthorized function to test whether authenticated users are allowed to update or select data from a datasource.

Example: orders.cfm

```
<!---  This example calls the IsAuthorized function. --->

...

<!--- First, check whether a form button was submitted --->

<CFIF IsDefined("form.btnUpdate")>

<!--- Is user is authorized to update or select
    information from the Orders data source? --->

    <CFIF ISAUTHORIZED("DataSource", "Orders", "update")>
    <CFQUERY NAME="AddItem" DATASOURCE="Orders">
        INSERT INTO Orders
        (Customer, OrderID)
        VALUES
            <CFOUTPUT>(#Customer#, #OrderID#)</CFOUTPUT>
        </CFQUERY>
        <CFOUTPUT QUERY="AddItem">
        Authorization Succeeded. Order information added:
        #Customer# - #OrderID#<BR>
        </CFOUTPUT>

    <CFELSE>
        <CFABORT SHOWERROR="You are not allowed
        to update order information.">

    </CFIF>

</CFIF>

<CFIF ISAUTHORIZED("DataSource", "Orders", "select")>
    <CFQUERY NAME="GetList" DATASOURCE="Orders">
        SELECT *    FROM Orders
    </CFQUERY>
    Authorization Succeeded. Order information follows:
    <CFOUTPUT QUERY="GetList">
        #Customer# - #BalanceDue#<BR>
    </CFOUTPUT>

<CFELSE>
    <CFABORT SHOWERROR="You cannot view
        order information.">

</CFIF>
```

For more information

For more information on setting up security in ColdFusion, see the Configuring Advanced Security chapter of the *Administering ColdFusion Server* book.

Index

Special

21st century dates 11

A

Accessing ancestor data 106
Accessing generated content 105
ACTION attribute, in CFOBJECT 119
Adding
  CFX tags 114
  data to structures 59
  queries 131
  rows to a query 126
  string 139
Advanced dynamic expressions 38
Advanced security
  concepts 155
  implementing 156
Allaire
  contacting 5
  Developer resources 2
  Documentation 3
  Sales 5
  Technical support 5
  Web site 2
Allaire Alive 96
Allaire Tag Gallery 96
Ancestor data access 106
Anchoring a regular expression to a string 44
Application.cfm file
  application security 160
Application-defined exception events 84
Arithmetic operators 16
Arrays
  adding elements 51
  basic techniques 50
  creating 48
  dynamic 49
  functions 56
  index 48
  multidimensional 50
  populating with CFLOOP 53
  populating with data 52
  terminology 48
Arrays, associative 58, 63
ArraySet function 53
Associating sub-tags with the base tag, CFML custom tags 102
Associative arrays 58, 63
Attributes in CFXs
  getting 133
  passing 132
Authenticating users in Application.cfm 160
Authentication example 160
Authorization
  example 160

B

Backreferences in regex replace strings 43
Boolean operators 19
Boolean values 10
Building Extensions in C++ 111
Building Extensions in CFML 96

C

C++ extensions 111
Calculating an array index 52
Catching security exceptions 159
CCFXException class 125
CCFXQuery
  AddRow 125
  GetColumns 126
  GetData 127
  GetName 127
  GetRowCount 128
  SetData 128
  SetQueryString 129
  SetTotalTime 129
CCFXRequest
  AddQuery 131
  AttributeExists 132
  CreateStringSet 132
  Debug 132
  GetAttribute 133
  GetAttributeList 133
  GetCustomData 134
  GetQuery 134
  GetSetting 135
  ReThrowException 135
  SetCustomData 136
  SetVariable 136
  ThrowException 137
  Write 138
  WriteDebug 138
CCFXStringSet
  AddString 139
  GetCount 139
  GetIndexForString 140
  GetString 140
CF_COOLLINK custom tag 99
CF_COUNTWORD custom tag 99
CF_MERGEQUERY custom tag 98
CFAPI tag 112
  CFAPI_tag_table 124
CFAUTHENTICATE tag
  example 157
  syntax 157
  using SETCOOKIE 157
CFCATCH tag
  exception handling 83
  exception information 88
  variables 88
cfcrypt encryption utility 110
CFEXIT 104
CFLDAP tag 144

Action= "Delete" 149
Action="ModifyDN" 150
Action="Query" 149
CFML custom tag enhancements 102
CFML Scripting 75
CFMODULE tag
resolving custom tag name conflicts 109
CFOBJECT tag 117
ACTION attribute 119
CORBA objects 121
CFOUTPUT tag, LDAP 147
CFREGISTRY tag 92
CFREPORT tag, LDAP 147
CFScript 75
assignment statements 77
CFML expressions 77
comments 80
differences from JavaScript 80
expressions 80
interaction with CFML 81
reserved words 81
CFTHROW tag
exception handling 83
syntax 87
CFTRY tag 88
exception handling 83
order of evaluation 88
CFWDDX 67
CFX tags
adding 114
C++
CFX tags 111
changing settings 115
implementing 112
Solaris 112
Character classes in regular expressions 41
CLASS attribute 117
ColdFusion expressions, about 8
ColdFusion extensions, about 96
ColdFusion Support for LDAP 144
COM objects 14
creating 118
invoking methods 120
late binding 116
setting properties 120
transparent location 116
COM/OLE Viewer 117
Comments in CFScript 80
Common Object Request Broker Architecture. See CORBA. 115
Complex expressions 25
Configuring RDS security 155
Contacting Allaire 5
Converting CFML Data to a JavaScript Object 70
Copying structures 60
CORBA objects
CFOBJECT 121
getting started 121
OMG IDL 116
using 115
Corporate headquarters, Allaire 5
Custom C++ tag 124
adding and deleting 114
CCFXException class 125
CCFXQuery class 125
CCFXRequest class 130
CCFXStringSet class 132
debugging and registering 113
global setting 135
request object 112
settings 115
Custom CFML tags
deleting 115
editors 97
encrypting 110
enhancements 102
examples 98
modes of execution 103
names 106
resolving name conflicts 109
securing 110
sharing 97
Custom tags
installing 97
managing 109
naming 98
writing 97

D
Data exchange across application servers 69
Database
exceptions 89
failures 84
LDAP 147
Date-and-time values 11
DBQUERY tag 134
DCOM 115
DEBUG attribute 132, 138
Debugging
CFX tags 113
custom pages and tags 113
expressions 30
setting watches 30
Decision, or Comparison, operators 18
Defining Attributes 98
Deleting
CFX tags 115
custom CFML tags 115
Registry values 94
structures 61
Developer resources 2
Developing applications in ColdFusion Studio 3
Directory, LDAP. TCP 144
Distinguished name 146
Distributing custom CFML tags 113
DLLs, deploying COM objects as 117
DN (Distinguished Name) 146
Documentation
Allaire 3
distribution 4
Dynamic arrays 49
shifting indexes 51
Dynamic expression evaluation 33
about 34
string expressions 34
Dynamic variables 37

E
Encrypting application pages 157
Encrypting custom CFML tags 110
Encryption
using cfcrypt 110
Evaluate function 37
guidelines 36
string expressions 36
using with SetVariable function 37
Evaluating dynamic expressions 33
Exception handling 83
database exceptions 89
exception types 84
locking exceptions 89
strategies 85
types of recoverable exceptions 84
using CFCATCH 88
using CFTHROW 87
using CFTRY 88
Exceptions
application-defined 84
database 84
EXEs, COM objects deployed as 117
Expression Builder 9
Expressions
21st century dates 11
about 8
arithmetic operators 16

arrays 13
Boolean values 10
COM objects 14
creating 9
creating in ColdFusion Studio 9
date-and-time values 11
debugging 30
evaluating 30
examples 8
exceptions 85
Expression Builder 9
how ColdFusion processes 27
in CFScript 80
integer numbers 9
lists 12
numbers in 9
operators 16
queries 14
real numbers 9
scientific notation 10
setting watches 30
strings 10
structure of 9
structures 13
testing 31
time formats 11
troubleshooting 30
typelessness 27
variables 15
Extensions, about 96
Extensions, C++ 111

F

Find and Replace commands 40
Function tags 99
Functions 21
Boolean values 23
ColdFusion 21
types 21
usage 22

G

Generated content, accessing 105
Getting Registry Values 92

H

Handling exceptions
based on diagnostic information 86
based on point of origin 85
High-level data exchange 108
How WDDX Works 69

I

Implementing CFX tags 112
Inserting complex expressions in strings 26
Installing custom tags 97
Interaction of CFScript with CFML 81
Inter-tag data exchange 105
IsAuthenticated function 158
IsAuthorized function 158

K

Key-value pairs in structures 57

L

Late binding, COM objects 116
LDAP
attributes 146
key terms 146
viewing directory schema 145
Lists, in expressions 12
Local CFML custom tags, installing 97
Locking exceptions 85
Looping through structures 63

M

Managing CFX Tags 114
Managing custom tags 109
Microsoft COM/OLE Viewer 117
Missing included file errors 85
MissingInclude exceptions 89
Modes of execution, custom CFML tags 103
Multi-character regular expressions 42
Multidimensional arrays 50

N

Naming custom tags 98
resolving conflicts 109
Nested pound signs in expressions 26
Nesting CFLOOPs
2D array 54
3D array 54

O

Object exceptions 85
OLE viewer 117
OMG Interface Definition Language 116
Online documentation 4
Operation-driven evaluation 27
Operators 16
arithmetic 16
Boolean 18
comparison 18
decision 18
decision, or comparison 18
precedence 20
string operators 17
Optional arguments in functions 23
ORBs 116
Order of evaluation 88
Output, LDAP 147

P

Populating arrays with data 52
POSIX character classes 41
Pound signs
in CFOUTPUT tags 24
in general expressions 27
inside string expressions 36
inside strings 25
inside tag attributes 26
nested 26
using 24
Properties and methods
COM 120

Q

Queries 14
QUERY attribute 134
Query, data
adding 126
CFX 124
execution time 129
LDAP 147

R

RDN (Relative Distinguished Names) 146
RDS security 154
configuring 155
Referencing elements in dynamic arrays 51
Referral, LDAP 147
REFind function 40
REFindNoCase function 40
Registering
CFX tags 113
COM objects 117
Registry
getting a specific value 93
getting all keys and values 92
Registry Values 92
Regular expressions
anchoring to search string 44
backreferences 42
CFML examples 45

character classes 41
examples 44
finding repeated words 43
multi-character 42
returning matched sub-expressions 43
single-character 40
Remote Development Services (RDS)
Security 154
REReplace function 40
REReplaceNoCase function 40
Reserved words in CFScript 81
Resolving file name conflicts in custom CFMLtags 109

S

Sales, Allaire 5
Sample C++ tags 112
Scientific notation in expressions 10
Scope, LDAP 147
Scripting in CFML 75
Search and replace 40
Search Filters 148
Securing custom tags 110
Security
application security 153
authentication example 160
authorization example 160
contexts 155
encrypting strings 157
exceptions 85
LDAP 147
policies 156
rules 155
sandboxes 156
server 156
user directories 155
users/groups 155
Server, LDAP 144
SETCOOKIE in CFAUTHENTICATE 157
Setting
global custom tag 135
Properties and invoking methods in CFOBJECT 120
variables dynamically 37
Setting Registry values 93
SetVariable function
using with Evaluate function 37
Shared custom CFML tags 97
Shifting indexes in dynamic arrays 51
Shorthand notation for Boolean operators 18
Single-character regular expressions 40
Solaris
building CFX tags 112
String expressions 34
converting CFML expressions 34
evaluating 35
pound signs (#) 36
quote marks 34
Structured exception handling. See also Exception handling 83
Structures 13, 57
about 58
adding data to 59
as associative arrays 63
copying 60
creating 58
deleting 61
examples 61
finding information in 59
functions 64
getting information about 60
looping through 63
notation 58
Supported statements in CFScript 76

T

Tag Gallery 96
TCP network directory 144
Technical support, Allaire 5
Template errors, handling 84
Testing expressions using CFSET and CFOUTPUT 31
Thread safe, CFX tags 112
Time formats 11
Transferring Data From Browser to Server 72
Troubleshooting
expressions 30
Typeless expression evaluation
examples 29
Typelessness
conversion between types 28
in expressions 27
Types of recoverable exceptions supported 84

U

Unary arithmetic operators 17
Unexpected internal exceptions 85
Updating data, LDAP 147
User security 153
authenticating users 158
authorizing users 158
catching exceptions 159
concepts 155
example 161
example of IsAuthorized 162
implementing 156
overview 155
Using backreferences in regex replace strings 43

V

Variables in CFScript 80
Viewing directory schema, LDAP 145
Visual Tool Markup Language 97
VTML 97

W

WDDX 69
Components 68
example 70
Web Distributed Data Exchange (WDDX) 67
Windows NT 113
COM objects 117
Writing custom tags 97

X

XML and ColdFusion 67