Developing Web Applications with ColdFusion

ColdFusion 4.5

Allaire Corporation

Copyright Notice

Part number: AA-45WEB-RK

Contents

Chapter 11: Indexing and Searching Data 149

Chapter 12: Using the Application Framework 183

Welcome to ColdFusion

This manual describes the process of developing Web applications using ColdFusion. In the first six chapters, you can follow the instructions presented to learn how to create basic ColdFusion applications. Then, chapters seven through 17 cover various topics of interest in enhancing your applications. Finally, chapters 18 through 21 explain how to extend ColdFusion's capabilities.

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

Contents

Intended Audience

This manual is particularly useful for Web application developers who are new to ColdFusion. In particular, Chapters 1 through 6 provide instructions for creating a basic ColdFusion application. If you are somewhat familiar with ColdFusion, but want to learn more about a particular topic such as sending and receiving email, refer to Chapters 7 through 17. Finally, if you want to extend ColdFusion's capabilities with CFML scripting or creating custom tags, Chapters 18 through 21 will be helpful.

Welcome to the ColdFusion 4.5 Web Application Server

The ColdFusion 4.5 release focuses on fundamentals — the fundamentals of delivering your e-business: faster development, better reliability, enhanced scalability, expanded integration, and stronger security.

At the center of the ColdFusion 4.5 release is an application server platform that's been highly optimized with new functionality and native support for UNIX. As a result, your e-business systems will run better and do more. With this release we're launching a new edition of ColdFusion Server for Linux so you can take advantage of the reliability and performance of the hottest new Internet server operating system.

While optimizing the core server, we also enhanced fundamental features including email integration, server-side FTP and HTTP, advanced security, scheduling, and database connectivity — again giving you more reliability and new functionality.

The focus on fundamentals extends to new features. As part of a broad new commitment to Java, ColdFusion 4.5 has a range of new Java integration options from Java CFXs to Java Servlet support to Java object and EJB connectivity. In ColdFusion Studio 4.5, we added new tools to make you more productive including a flexible new project architecture that makes managing and deploying complex Web applications a snap. On the server, we focused on reliability, performance and security with features such as service-level fail-over, Cisco Local Director integration, and OS security integration.

Whether you're revolutionizing your company's HR operations, building the next generation of your firm's global intranet, or launching the next killer .COM company, you'll find the speed, scalability, connectivity, and security you need in ColdFusion 4.5.

Products and System Requirements

ColdFusion has been fully tested on the following platforms and with the following configurations.

ColdFusion Server 4.5 Enterprise Edition for Windows

- Windows NT 4.0 SP4+
- Intel Pentium or above
- 150 MB hard disk space
- 128 MB RAM (256 MB recommended for clustering)

ColdFusion Server 4.5 Enterprise Edition for Solaris

- SPARC Solaris 2.5.1, 2.6, or 7 (patch 103582-1B or higher)
- 128 MB RAM (256 MB recommended for clustering)
- 200 MB hard disk space

ColdFusion Server 4.5 Enterprise Edition for Linux

- Red Hat Linux 6.0 or 6.1
- Intel Pentium or above
- 128 MB RAM (256 MB recommended for clustering)
- 150 MB hard disk space

ColdFusion Server 4.5 Professional Edition for Windows

- Windows 95/98 or Windows NT 4.0
- Intel Pentium or above
- 50 MB hard disk space
- 32 MB RAM (128 MB recommended)

ColdFusion Server 4.5 Professional Edition for Linux

- Red Hat Linux 6.0 or 6.1
- Intel Pentium or above
- 64 MB RAM (128 MB recommended)
- 100 MB hard disk space

ColdFusion Studio 4.5

- Windows 95/98/NT4
- Intel Pentium or above
- 35 MB hard disk space
- 32 MB RAM (64 MB recommended)

New Features in ColdFusion 4.5

A wide range of new features are available in ColdFusion 4.5.

New visual tools

Universal File Browser — Access all your files from a single explorer that integrates access to the Windows file system, ColdFusion RDS servers, and FTP servers. Drag-and-drop between any of these services all in an integrated file browser.

Advanced Project Management — Manage your complex Web application development projects with a new project architecture that gives you more flexibility and control using physical, virtual, and auto-inclusive project folders as well as project resource browsing.

Scriptable Deployment — Deploy applications to complex server configurations with FTP or ColdFusion Remote Development Services (RDS). Use VBScript or Java Script to fully script deployment of projects with granular control over how files uploaded. Setup deployment scripts using a powerful wizard and save scripts for re-use.

Collapsible Code — Work with large, complex scripts and pages more easily by collapsing sections of the code in the editor so you can build sophisticated applications more quickly.

Function Insight — Find the parameters and format for functions instantly and inline as you code.

Image Map Editor — Create image maps right in ColdFusion Studio with a new easy-to-use visual tool.

Configuration Wizard — Setup your work environment so it meets all your needs using any of more than dozen common configurations.

TopStyle CSS Editor — Create and edit standards-compliant cascading style sheets to easily control the look and feel of your web applications.

WML Support — Build wireless Web applications quickly and easily with the complete set of Wireless Markup Language (WML) visual tools.

Enhancements to CFML

Object Scripting — Instantiate and script objects using CFML script in addition to the CFOBJECT tag easier integration with distributed object middleware such as COM and CORBA.

Structured Exception Handling — Exception handling now offers hierarchical exception handling that supports both greater customization and greater access to internal exceptions.

String Conversion Functions — Convert strings quickly and easily to be compatible with Java Script and XML standards.

Better reliability

Server Probes — Guarantee high availability by automatically detecting when a ColdFusion Server or Web server hangs or stops, failing-over to a new machine in a ColdFusion cluster, and restarting the server with problems. (Enterprise Edition only)

Improved Automatic Server Recovery — Monitor and automatically restart server process in case of failures or critical errors on individual servers not deployed in a cluster.

Clustering Support for Apache — Setup ColdFusion clusters on Linux and Solaris using the Apache Web Server. (Enterprise Edition only)

Automatic Shared Variable Locking — Lock user and session variable reads automatically at the server level to prevent destabilizing conflicts and control thread write contentions. Configure variable locking to meet the specific needs of your applications.

Individual Data Source Control — Enable and disable individual data sources individually without affecting server availability for runtime data source maintenance without server restarts.

Improved performance

Cisco Local Director Integration — Deliver very large scale sites with Cisco Local Director intelligently balancing load based on the load metrics provided by the ColdFusion Servers in a cluster. (Enterprise Edition only)

Client-Side Page Caching — Leverage browser page caching to avoid unnecessary downloads of unchanged pages and improve overall site performance. Programmatically control refresh of client-side cache to ensure users see most up-to-date output.

White Space Removal — Reduce white space left by processed code in application pages to make the pages smaller and faster. Control white space removal programmatically or administratively.

Scriptable Performance Metrics — Track key server metrics at run time through your own scripts for intelligent diagnosis of performance bottlenecks of stability problems in your applications.

Performance Debugging Data — Access detailed debugging information on the performance of each individual page included in an application page that is being debugged.

Enterprise connectivity features

Transaction Commit and Rollback — Control database transactions with programmable commit and rollback support for more reliable and better-managed database interactions.

Java Object and EJB Connectivity — Connect to any Java object or Enterprise JavaBean (EJB) hosted by any major EJB server to extend ColdFusion and access complex business logic or third party distributed components.

Java Servlets — Call Java Servlets hosted by a Servlet Engine such as Allaire JRun from within a ColdFusion application to access extended functionality

Java-based ColdFusion Extensions (CFX) — Extend ColdFusion with new functionality through CFXs created with Java.

Binary Object Support — Use Character Large Binary Object (CLOB) support to encoded binary objects, transmit them via XML, and store them in databases or files.

SQL Bind Parameters — Improve query performance, security and flexibility with explicitly typed query parameters.

WDDX 1.0 — Exchange complex data, including encoded images, between servers and with other programming environments even faster using the latest version of Web Distributed Data Exchange (WDDX).

OS Command Execution — Execute OS shell scripts, services, executables and batch files from within ColdFusion applications.

LDAP 3.0 — Use all the power of LDAP 3.0 for directory access including file filtering, SSL encryption, and Microsoft Active Directory integration.

Enhanced Mail Integration — Develop more sophisticated and robust email applications with new support for controlling mail headers, BCC, and multiple file attachments.

Improved Server-Side HTTP — Use URL redirection, SSL, cookies, return headers, and more robust server-side HTTP support for building distributed Web applications.

Security enhancements

General OS Security Integration — Secure entire Web applications and control access to files and objects through your existing Windows NT security architecture. Authenticated users in your applications can be limited to privileges authorized through Windows security. (Windows NT Only)

OS Server Sandbox Security — Secure shared hosting environments more easily by creating Server Sandboxes with Windows NT security. OS Server Sandboxes process all requests under the privileges of a designated Windows NT user account (Enterprise Edition for Windows only).

Enhanced Advanced Security — Secure CFML functions and enable CFML code segments to be executed using the run-time security permissions of a designated user.

New Advanced Security Interface — Manage Advanced Security configuration more quickly and easily with a completely redesigned browser-based resource view.

Scriptable Advanced Security Administration — Configure ColdFusion Advanced Security through your own CFML scripts for easier maintenance of ColdFusion Servers.

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.
Training and Consulting www.allaire.com/services	Information about training classes, online courses, and consulting services 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 Partners www.allaire.com/partners	The Allaire Alliance is a network of solution providers, application developers, resellers, and hosting services creating solutions with ColdFusion.

About ColdFusion Documentation

ColdFusion documentation 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.

In addition to the book set, the documentation is provided in two other formats:

- **HTML** — Browser-based Help references.
- **Adobe Acrobat (PDF)** — Available from the root level on the product CD-ROM and from the Developer area of Allaire's Web site at `http://www.allaire.com/developer`.

Documentation updates

Late additions and corrections to ColdFusion printed documentation are listed in the Documentation Updates page. To reach this page, open the Welcome to ColdFusion page installed with ColdFusion, where you'll find links to the update page as well as links to other pages containing useful information about ColdFusion, Allaire support options, and Allaire products and services.

For ColdFusion Studio users, you can access the documentation update page by clicking on the Help resource tab and browsing your way through the online help tree to the Allaire Support folder.

ColdFusion manuals

The core ColdFusion documentation set consists of the following titles.

Administering ColdFusion Server

Includes instructions for installing ColdFusion Server. Describes configuration options for maximizing performance, managing data sources, setting security levels, and a range of development and site management tasks. If you are administering a ColdFusion site, you'll need this book to help plan and implement ColdFusion security, load balancing, and for details about tuning the ColdFusion application server.

Developing Web Applications with ColdFusion

Presents the fundamentals of ColdFusion application development and deployment. Also includes detailed information about ColdFusion data sources, user interfaces, and Web technologies.

CFML Language Reference

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

Using ColdFusion Studio

Documents everything you need to know about using ColdFusion Studio, including features like projects, source control integration, as well as the Studio workspace and interface.

ColdFusion Quick Reference Guide

A valuable quick reference to CFML tags, functions, and variables.

ColdFusion Server online documentation

To view the HTML documentation, open the following URL: `http://127.0.0.1/cfdocs/dochome.htm`.

Note that because the Verity search libraries are not available on Linux for this release, the online documentation search facility is not functional on Linux. If you try to open the search page, a message box opens to explain why the facility is not available.

Acrobat versions of all ColdFusion documentation are available from the root level on the product CD. If you don't have a product CD, you can download ColdFusion documentation from the Allaire web site by visiting `http://www.allaire.com/developer` and clicking the Documentation link.

ColdFusion Studio online documentation

Click the Help resource tab in ColdFusion Studio to view online Help pages. The help tree contains ColdFusion documentation and a number of additional developer resources. Studio online documentation is searchable and individual pages can be bookmarked.

Printing ColdFusion documentation

If you are working with an evaluation version of ColdFusion and would like printed documentation, access the Adobe Acrobat files found from the root level on the product CD. If you do not have access to a product CD, you can download the Acrobat files from the Allaire web site: `http://www.allaire.com/developer`, click the Documentation link.

The Acrobat files offer excellent print output. You can print an entire manual, individual sections, or page ranges of your choice. To get the Acrobat reader, visit: `http://www.acrobat.com`.

Documentation conventions

When reading, please be aware of these formatting cues:

- Code samples, filenames, and URLs are set in a `monospaced` font
- Notes and tips are identified by bold type
- 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*

Getting Answers

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

Contacting Allaire

Corporate headquarters

Allaire Corporation
One Alewife Center
Cambridge, MA 02140

Tel: 617.761.2000
Fax: 617.761.2001

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)

For complete details about Allaire Product Support options, please refer to the Allaire Support pages on the Allaire web site: http://www.allaire.com/support.

Postings to the ColdFusion Support Forum (http://forums.allaire.com) can be made 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 1

Introduction to ColdFusion

This chapter explains the difference between creating static Web pages with HTML and creating dynamic applications with ColdFusion. It also describes what ColdFusion is and how it works.

Contents

A Quick Web Overview

Over the last few years, the Web has changed from being simply a collection of static HTML pages to an application development platform. Rather than offering a space where organizations can merely advertise goods and services, similar to traditional yellow pages directories, companies conduct business ranging from ecommerce to managing internal business processes. For example, a static HTML page would allow a bookstore to publish its location, list services such as the ability to place special orders, and advertise upcoming events like book signings. A dynamic site for the same bookstore would allow customers to order books online, write reviews of books they've read, and even get suggestions for purchasing books based on their reading preferences.

ColdFusion is a rapid application development environment that lets you build dynamic sites. You can use the Web to handle business transactions, and even to conduct the day to day business of your organization.

What You Should Already Know

Before you begin using ColdFusion to create your Web applications, you should be familiar with the following topics:

HTML

You'll find that ColdFusion tags (CFML) are similar in syntax to HTML tags, yet, unlike HTML, they enable you to create dynamic Web pages. You should understand how to create a basic HTML page, put information into tables, gather data in forms, and create links.

Relational Database Design and Management

If you plan on creating applications that use data from existing data sources, you should understand how the data is organized. In most cases, this means understanding how tables are organized to prevent unnecessary duplication of data. For example, if you have data about employees, rather than repeating the department number and name in each employee's record, you would most likely have a separate table that lists each department number and name just once.

SQL

Familiarity with some Structured Query Language (SQL) will be helpful as you develop your ColdFusion applications. In particular, you should be able to use the SELECT, UPDATE, INSERT, and DELETE statements, as well as WHERE clauses and boolean logic operators.

What is ColdFusion?

ColdFusion lets you create page-based Web applications using ColdFusion Markup Language (CFML), the tag-based language you use to create server-side scripts that dynamically control data integration, application logic, and user interface generation. ColdFusion Web applications can contain XML, HTML, and other client technologies such as CSS and JavaScript.

ColdFusion application pages are different from static HTML pages in the following ways:

- They are saved and referenced with a specific file extension.
- The default ColdFusion file extension is CFM.
- They contain ColdFusion Markup Language.

Editions of ColdFusion

There are three editions of ColdFusion: Enterprise, Professional, and Express. Using ColdFusion Enterprise and Professional editions and ColdFusion Studio, you can build Web applications that leverage existing technologies and business systems such as RDBMS, messaging servers, file repositories, directory servers, and distributed object middleware. ColdFusion Enterprise also offers advanced security features, load balancing, server fail-over, and visual cluster administration. Using ColdFusion Express, you can build Web applications that interact with desktop databases that support the ODBC standard.

ColdFusion Features

ColdFusion provides a comprehensive set of features that enable:

- Rapid development
- Scalable deployment
- Open integration
- Complete security

Rapid development

The ColdFusion development platform enhances the speed and ease of development through the following features:

- A tag-based server scripting language that is powerful and intuitive.
- Two-way visual programming and database tools.
- Remote interactive debugging for quickly identifying and fixing problems.
- Web application wizards to automate common development tasks.
- Source control integration to enable team development.
- Secure file and database access via HTTP for remote development.
- A tag-based component architecture for flexible code reuse.

Scalable deployment

ColdFusion delivers a high-performance platform for application deployment through the following features:

- A multi-threaded service architecture that scales across processors.
- Database connection pooling to optimize database performance.
- Just-in-time page compilation and caching to accelerate page request processing.
- Dynamic load balancing for scalable performance in a cluster environment (Enterprise Edition only).
- Automatic server recovery and fail-over for high availability (Enterprise Edition only).

Open integration

ColdFusion integrates with new and legacy technologies through the following features:

- Database connectivity using native database drivers (Enterprise Edition only), ODBC, or OLE-DB.
- Embedded support for full text indexing and searching.
- Standards-based integration with directory, mail, HTTP, FTP, and file servers.
- Connectivity to distributed object technologies including CORBA (Enterprise Edition only), COM (Windows Enterprise Edition only), Java objects and EJBs.
- Open extensibility with C/C++ and Java.

Complete security

ColdFusion provides a foundation for building secure applications through the following features:

- Integration with existing authentication systems including Windows NT domain and LDAP directory servers, and proprietary user and group databases.
- Advanced access control so that server administrators can control developers' access to files and data sources.
- Support for existing database security.
- Server sandbox security for protecting multiple applications on a single server (Enterprise Edition only).
- Support for existing Web server authentication, security, and encryption.

For detailed information on security, refer to *Administering ColdFusion Server*. Also, for the latest publications from Allair on security, visit the Security Zone at http://www.allaire.com/developer/securityzone/. For a complete feature list and more detailed information, refer to the ColdFusion product pages, http://www.allaire.com/coldfusion.

ColdFusion Components

ColdFusion applications rely on several core components:

- ColdFusion Studio
- ColdFusion application pages
- ColdFusion Server
- ColdFusion Administrator
- ODBC data sources and other data sources

ColdFusion application pages look somewhat like HTML pages, but, as you will see, are much more dynamic and powerful. You will probably want to use ColdFusion Studio to create the application pages, although you can use the editor of your choice. ColdFusion Server processes the ColdFusion application pages. For example, you may access a data source from your application pages.

In addition to the core components, as you become more familiar with ColdFusion and build more complex applications, you can use ColdFusion Extensions to extend its capabilities.

ColdFusion Studio

ColdFusion Studio is the development environment for ColdFusion Server. It offers visual development tools, including dynamic page previews using your Web browser, an interactive debugger, a query builder, an expression builder, project management and source control tools, and many other productivity enhancements. To learn more about ColdFusion Studio, see *Using ColdFusion Studio*.

ColdFusion application pages

Application pages are the functional parts of a ColdFusion application, including the user interface pages and forms that handle data input and format data output. They can contain ColdFusion tags (CFML), HTML tags, CFScript, JavaScript, and anything else you can normally embed in an ordinary HTML page. The default file extension used for ColdFusion application pages is .CFM.

CFML

CFML is a tag-based server scripting language that encapsulates complex processes such as connecting to databases and LDAP servers, and sending email. The core of the ColdFusion development platform language is more than 70 server-side tags and more than 200 functions.

ColdFusion Server

ColdFusion Server listens for requests from the Web server to process ColdFusion application pages. It runs as a service under Windows NT and as a process under UNIX. For information on installing and configuring ColdFusion Server, refer to *Administering ColdFusion Server*.

ColdFusion Administrator

You use the Administrator to configure various ColdFusion Server options, including:

- ColdFusion data sources
- Debugging output
- Server settings
- Application security
- Server clustering
- Scheduling page execution
- Directory mapping

See *Administering ColdFusion Server* for details on using the Administrator.

Data sources

ColdFusion applications may interact with any database that supports the ODBC standard. However, ColdFusion is not limited to ODBC data sources. You can also retrieve data using OLE-DB, native database drivers, or directory servers that support the Lightweight Directory Access Protocol (LDAP). Data can also be retrieved from mail servers that support the Post Office Protocol (POP), and which is indexed in Verity collections.

How ColdFusion Server Works

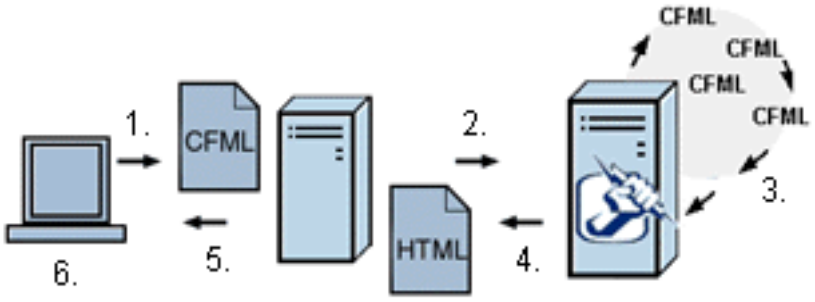
Regardless of which ColdFusion Server you have installed, ColdFusion application pages are processed on the server at runtime, each time they are requested by a browser.

A page request happens when you click on a Web site link to open a Web page in your browser. When you request a ColdFusion application page, ColdFusion server processes the request, retrieves any data if necessary, routes the data through the Web server, back to your browser.

In more detail, here's what happens when a Coldfusion page is opened:

1. The client requests a page that contains CFML tags.
2. The Web server passes files to ColdFusion Server if a page request contains a ColdFusion file extension.
3. ColdFusion Server scans the page and processes all CFML tags.
4. ColdFusion Server then returns only HTML and other client-side technologies to the Web server.
5. The Web server passes the page back to the browser

.



CHAPTER 2

Writing Your First ColdFusion Application

This chapter guides you through the ColdFusion development process as you create a ColdFusion application page, save it, and view it in a browser.

Contents

The Development Process

Whether you are creating a static HTML page or a ColdFusion application page, you follow the same iterative process:

- Write some code.
- Save the code to a document or page.
- View the page in a browser.
- Modify the page.
- Save the page again.
- View it in a browser.
- and so on...

Writing Code

Although you can code your application pages using NotePad or any HTML editor, this manual will use ColdFusion Studio because it affords many features that make ColdFusion development easier. See *Using ColdFusion Studio*for details. If you haven't already done so, you should install ColdFusion Studio.

From a coding perspective, the major difference between a static HTML page and a ColdFusion application page is that ColdFusion pages contain ColdFusion Markup Language (CFML). CFML is a markup language that's very similar in syntax to HTML so Web developers find it intuitive.

Unlike HTML which defines how things are displayed and formatted on the client, CFML identifies specific operations that are performed by ColdFusion Server.

To create a ColdFusion application page:

1. Open ColdFusion Studio.
2. Select File > New and select the Default Template for your new page.
3. Edit the file so that it appears as below:

```
<HTML>
<HEAD>
<TITLE>Call Department</TITLE>
</HEAD>
<BODY>
<STRONG>Call Department</STRONG>
<CFSET Department="Sales">
<CFOUTPUT>
I'd like to talk to someone in #Department#.
</CFOUTPUT>
</BODY>
</HTML>
```

Saving Application Pages

Instead of saving pages with an HTM or HTML file extension, you save ColdFusion application pages with a CFM or CFML extension. By default, the Web server knows to pass a page that contains a CFM extension to the ColdFusion Server when it is requested by a browser.

Save ColdFusion application pages underneath the Web root or another Web server mapping so that the Web server can publish these pages to the Internet. For example, you might want to create a directory `myapps` and save your practice pages there.

To save the page:

1. Select File > Save.
2. Save your page as `calldept.cfm` in `myapps` under the Web root directory.

   For example, the directory path on your machine may be:

   `c:/inetpub/wwwroot/myapps` on Windows NT or

   `<mywebserverdocroot>/myapps` on UNIX

Viewing Application Pages

You view the application page on the Web server to ensure that the code is working as expected. Presently, your page is very simple. But, as you add more code, you will want to ensure that the page continues to work.

To view the page in a local browser:

1. Open a Web browser on your local machine and enter the following URL:

   `http://127.0.0.1/myapps/calldept.cfm`

   Where `127.0.0.1` refers to the localhost and is only valid when you are viewing pages locally.
2. Use the Web browser facility that allows you to view a page's source code to examine the code that the browser uses for rendering.

   Note that only HTML and text is returned to the browser.

Compare the code that was returned to the browser with what you originally created. Notice that the ColdFusion comments and CFML tags are processed, but do not appear in the HTML file that's returned to the browser.

Original ColdFusion page	HTML file returned by Web server
<HTML> <HEAD> <TITLE>Call Department</TITLE> </HEAD> <BODY> <STRONG>Call Department</STRONG> <!--- Set all variables ---> <CFSET Department="Sales"> <CFOUTPUT> I'd like to talk to someone in #Department#. <!--- Display results ---> </CFOUTPUT> </BODY> </HTML>	<HTML> <HEAD> <TITLE>Call Department</TITLE> </HEAD> <BODY> <STRONG>Call Department</STRONG> I'd like to talk to someone in Sales. </BODY> </HTML>

Code Review

The application page that you just created contains both HTML and CFML. You used the CFML tag CFSET to define a variable, Department, and set its value to be "Sales." You then used the CFML tag CFOUTPUT to display text and the value of the variable.

Code	Description
<!--- Set all variables --->	CFML comment, which is not returned in the HTML page.
<CFSET Department="Sales">	Creates a variable named Department and sets the value equal to Sales.
<!--- Display results --->	CFML comment, which is not returned in the HTML page.
<CFOUTPUT> I'd like to talk to someone in #Department#. </CFOUTPUT>	Displays whatever appears between the opening and closing CFOUTPUT tags, in this case the text "I'd like to talk to someone in" followed by the value of the variable Department, which is "Sales."

Variables

A Web application page is different from a static Web page because it can publish data dynamically. This involves creating, manipulating, and outputting variables.

A variable stores data that can be used in applications. As with other programming languages, you'll set variables in ColdFusion to store data that you want to access later. And you'll reference a range of variables to perform different types of application processing.

There are a variety of variable types that you can create and reference in your ColdFusion applications. Also, ColdFusion variables are typeless, which means that you don't need to define whether or not the variable value is numeric, text, or time-date. See the *CFML Language Reference* for a complete list of variable types

The primary differences between variable types are where they exist, how long they exist, and where their values are stored. These considerations are referred to as a variable's scope.

You will learn more about scope as needed throughout this book.

For example, you would store a user's preferences in a variable in order to use that data to customize the page that's returned to the browser.

You don't use pound signs when you create the variable. However, when you want to display the value that a variable is set to, enclose the variable name in pound signs (#). The following table illustrates the use of pound signs and variable names.

CFML Code	Results
`<CFSET Department="Sales">`	The variable named Department is created and the value is set to Sales.

`<CFOUTPUT>`
`I'd like to talk to someone in Department.`
`</CFOUTPUT>`
	ColdFusion doesn't treat Department as a variable because it isn't surrounded by pound signs. The HTML page will display:
	I'd like to talk to someone in Department.

`<CFOUTPUT>`
`I'd like to talk to someone in #Department#.`
`</CFOUTPUT>`
	ColdFusion replaces the variable Department with its value. The HTML page will display:
	I'd like to talk to someone in Sales.

Adding More Variables to the Application

Applications can use many different variables. For example, the `calldept.cfm` application page can set and display values for department, city, and salary.

To modify the application:

1. Return to the file `calldept.cfm` in ColdFusion Studio,
2. Modify the code so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>Call Department</TITLE>
</HEAD>
<BODY>
<STRONG>Call Department</STRONG>
<!--- Set all variables --->
<CFSET Department="Sales">
<CFSET City="Boston">
<CFSET Salary="110000"
<!--- Display results --->
<CFOUTPUT>
I'd like to talk to someone in #Department# in #city# who earns at
least #Salary#.
</CFOUTPUT>
</BODY>
</HTML>
```

3. Save the file.
4. View the page in your Web browser by entering the following URL:

   `http://127.0.0.1/myapps/calldept.cfm`

Development Considerations

The same development rules that apply for any programming environment apply to ColdFusion. You should also follow the same programming conventions that you would with any other language:

- Comment your code as you go.

  HTML comments use this syntax: `<!-- html comment -->`

  CFML comments add an extra dash: `<!--- cfml comment --->`
- File names should be all one word, begin with a letter and can contain only letters, numbers and the underscore.
- File names should not contain special characters.

CHAPTER 3

Querying a Database

This chapter describes how to retrieve data from a database, work with query data, and enable debugging in ColdFusion applications. You will learn how to use the ColdFusion Administrator to set up a data source and enable debugging, use the CFQUERY tag to query a data source, and use the CFOUTPUT tag to output the query results to a Web page.

Contents

Publishing Dynamic Data

A Web application page is different from a static Web page because it can publish data dynamically. This can involve querying databases, connecting to LDAP or mail servers, and leveraging COM, DCOM, CORBA, or Java objects to retrieve, update, insert, and delete data at runtime — as your users interact with pages in their browsers.

For ColdFusion developers, the term "data source" can refer to a number of different types of structured content accessible locally or across a network. You can query Web sites, LDAP servers, POP mail servers, and documents in a variety of formats.

Most commonly though, a database will drive your applications, and for this discussion a data source is defined as the entry point for database operations.

During this chapter, you will build a query to retrieve data from `company.mdb`, an Access database. In subsequent chapters in this book, you will insert and update data in this database.

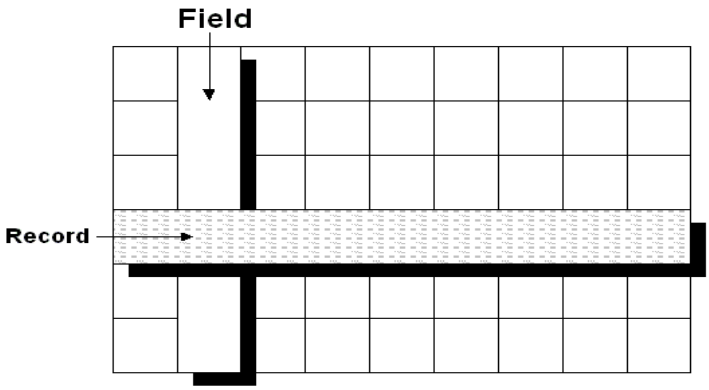
To build a query, you will need to use:

- ColdFusion data sources
- The CFQUERY tag
- SQL commands

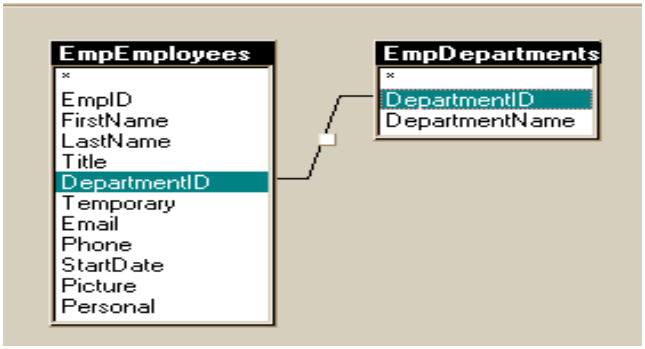
Database Basics

You don't need a thorough knowledge of databases to develop a data-driven ColdFusion application, but you will need to learn some basic concepts and techniques.

A database is a structure for storing information. Databases are organized in tables, which are collections of related items. For example, a table might contain the names, street addresses, and phone numbers of individuals. Think of a table as a grid of columns and rows. In this case, one column contains names, a second column contains street addresses, and the third column contains phone numbers. Each row constitutes one data record because the data in that row applies to a unique item, in this case, one individual. Rows are also referred to as records. Columns are also referred to as fields.



Data can be organized in multiple tables. This type of data structure is known as a relational database and is the type used for all but the simplest data sets.



From this basic description, a few database design rules emerge:

- Each record should contain a unique identifier, known as the primary key.

  This could be an employee ID, a part number, or a customer number. This is typically the column used to maintain each record's unique identity among the tables in a relational database.

- Once a column has been defined to contain a specific type of information, the data must be entered in that column in a consistent way.

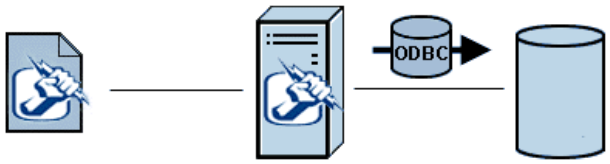
  This is accomplished by defining a data type for the column, such as allowing only numeric values to be entered in the salary column.

- Assessing user needs and incorporating those needs in the database design is essential to a successful implementation. A well-designed database accommodates the changing data needs within an organization.

The best way to familiarize yourself with the capabilities of your database product or DBMS is to review the product documentation.

Understanding Data Sources

A database is a file or server that contains a collection of data. A data source is a pointer from ColdFusion to a specific database. You add data sources to your ColdFusion Server so that you can point to the databases that you want to connect to from your ColdFusion applications.



Open Database Connectivity (ODBC)

ODBC is a standard interface for connecting to a database from an application. Applications that use ODBC must have an ODBC driver installed and configured for each data source.

On Windows, you can check your system's installed drivers by opening the ODBC Data Source Manager in the Windows Control Panel.

On Windows, the installed set of ColdFusion ODBC drivers includes:

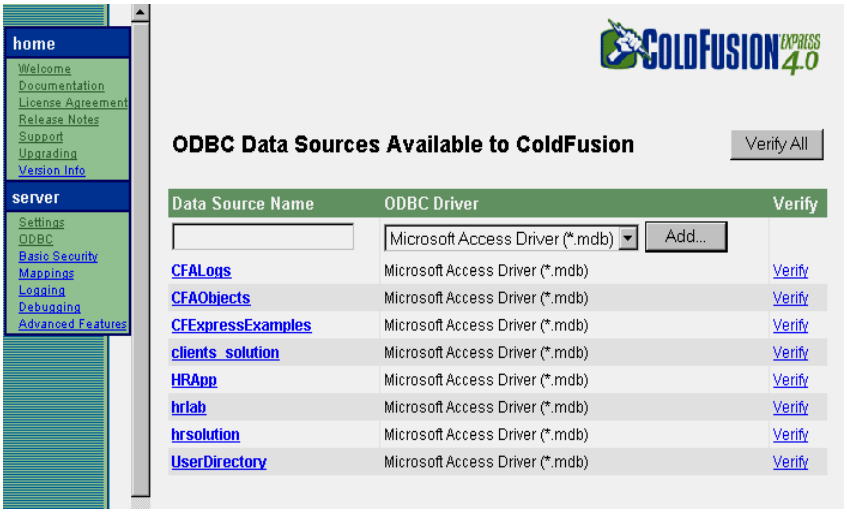
- Microsoft SQL Server
- Microsoft Access and FoxPro databases
- Borland dBase-compliant databases
- Microsoft Excel worksheet data ranges
- Delimited text files

On UNIX, look in the ODBC page of the ColdFusion Administrator for a list of avalable ODBC drivers.

A good source of information on ODBC is the *ODBC Programmer's Reference* at http://www.microsoft.com/data/odbc.

Adding Data Sources

You add data sources in the ColdFusion Administrator to define connection requirements for database access.



When you add a data source, you assign it a name so that you can reference it within tags such as CFQUERY on application pages to query databases. During a query, the data source tells ColdFusion which database to connect to and what parameters to use for the connection.

To add a data source:

1. Start the ColdFusion Administrator. On Windows, Select Start > Programs > ColdFusion Server > ColdFusion Administrator. On UNIX, enter the URL `hostname/CFIDE/administrator` in your browser.

   The Administrator prompts you for a password if you assigned one to the ColdFusion Server during install.

2. Enter a password to gain access to the Administrator.
3. Choose ODBC under the Data Sources heading on the left menu.
4. Name the data source `CompanyInfo`.
5. Select Microsoft Access Driver (*.mdb) from the dropdown box to describe the ODBC driver.
6. Choose Add.
7. In the Database File field, enter the full path of the `company.mdb` Access database and click OK.
8. Choose Create to create the `CompanyInfo` data source.

The data source is added to the data source list.

9. Locate `CompanyInfo` in the data source list.
10. Choose Verify to run the verification test on the data source.

    If the data source was created, you should see this message:

    ```
    The connection to the data source was verified successfully.
    ```

For more information about managing data sources, See *Administering ColdFusion Server*.

Data Source Notes and Considerations

When adding data sources to ColdFusion Server, keep these guidelines in mind:

- Data source names should be all one word and begin with a letter.
- Data source names can contain only letters, numbers and the underscore.
- Data source names should not contain special characters.
- Although data source names are not case-sensitive, you should use a consistent capitalization scheme.
- A data source must exist in the ColdFusion Administrator before you use it on an application page to retrieve data.

Retrieving Data

You can query databases to retrieve data at runtime. When retrieving data from a database:

- You use the CFQUERY tag on a page to tell ColdFusion how to connect to a database and how to store the retrieved data.
- You write SQL commands inside the CFQUERY block to specify the data that you want to retrieve from the database.
- The retrieved data is stored on that page as a query variable.
- You can reference the query variable data on that page in a CFOUTPUT block to use its values.

The CFQUERY Tag

The CFQUERY tag is one of the most frequently used CFML tags. You use it in conjunction with the CFOUTPUT tag so that you can retrieve and reference the data returned from a query.

When ColdFusion encounters a CFQUERY tag on a page, it does the following:

- Connects to the specified data source.

- Performs SQL commands that are enclosed within the block.
- Returns query variable values to the page.

CFQUERY tag syntax

```
<CFQUERY NAME="EmpList" DATASOURCE="CompanyInfo">
    You'll type SQL here
</CFQUERY>
```

In this example, the query code tells ColdFusion to:

- Use the `CompanyInfo` data source to connect to the `company.mdb` database.
- Store the retrieved data in the query variable EmpList.

In general, you should follow these guidelines:

- The CFQUERY tag is a block tag, that is, it has an opening <CFQUERY> and ending </CFQUERY> tag.
- Use the NAME attribute to name the query variable so that you can reference it later on the page.
- Use the DATASOURCE attribute to name an existing data source that should be used to connect to a specific database.
- Always surround attribute values with double quotes (").
- Place SQL statements inside the CFQUERY block to tell the database what to process during the query.
- When referencing text literals in SQL, use single quotes ('). For example, `Select * from mytable WHERE FirstName='Russ'` selects every record from mytable in which the first name is Russ.

Note The data source must exist in order to perform a successful query.

Writing SQL

In between the begin and end CFQUERY tags, write the SQL that you want the database to execute.

For example, to retrieve data from a database:

- Write a SELECT statement that lists the fields or columns that you want to select for the query.
- Follow the SELECT statement with a FROM clause that specifies the database tables that contain the columns.

Tip If you are using ColdFusion Studio, you can use the Query Builder to build SQL statements by graphically selecting the tables, and records within those tables you want to retrieve. See *Using ColdFusion Studio* for details.

When the database processes the SQL, it creates a data set that is returned to ColdFusion Server. ColdFusion places the data set in memory and assigns it the name that you defined for the query in the begin CFQUERY tag.

You may reference that data set by name using the CFOUTPUT tag further down on the page.

Basic SQL Syntax elements

The following sections present brief descriptions of the main SQL command elements.

Statements

These keywords identify commonly-used SQL commands:

Basic SQL Statements	
Keyword	**Description**
SELECT	Retrieves the specified records
INSERT	Adds a new row
UPDATE	Changes values in the specified rows
DELETE	Removes the specified rows

Statement clauses

These keywords are used to refine SQL statements:

Basic SQL Clauses	
Keyword	**Description**
FROM	Names the data source for the operation
WHERE	Sets one or more conditions for the operation
ORDER BY	Sorts the result set in the specified order.
GROUP BY	Groups the result set by the specified select list items.

Operators

These specify conditions and perform logical and numeric functions:

Basic SQL Operators	
Operator	**Description**
AND	Both conditions must be met, such as Paris AND Texas
OR	At least one condition must be met, such as Smith OR Smyth
NOT	Exclude the condition following, such as Paris NOT France
=	Equal to
<>	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
+	Addition
-	Subtraction
/	Division
*	Multiplication

SQL Notes and Considerations

Keep the following in mind when writing SQL in ColdFusion:

- There is a lot more to SQL than what is covered here. It's a good idea to purchase one or several SQL guides that you can refer to.
- The data source, columns, and tables that you reference must exist in order to perform a successful query.
- Some DBMS vendors use non-standard SQL syntax (known as a dialect) in their products. ColdFusion does not validate the SQL in a CFQUERY, so you are free to use any syntax that is supported by your data source. Check your DBMS documentation for non-standard SQL usage.

Building Queries

As discussed earlier in this chapter, you build queries using the CFQUERY tag and SQL.

To query the table:

1. Create a new application page.
2. Edit the page so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>Employee List</TITLE>
</HEAD>
<BODY>
<H1>Employee List</H1>
<CFQUERY NAME="EmpList" DATASOURCE="CompanyInfo">
SELECT FirstName, LastName, Salary, Contract
FROM Employees
</CFQUERY>
</BODY>
</HTML>
```

3. Save the page as `emplist.cfm` in `myapps` under the Web root directory. For example, the directory path on your machine may be:

   `C:\INETPUB\WWWROOT\myapps` on Windows NT

4. Return to your browser and enter the following URL to view `EmpList.cfm`:

   `http://127.0.0.1/myapps/emplist.cfm`

5. View source in the browser.

   The ColdFusion EmpList data set is created by ColdFusion Server, but only HTML and text is sent back to the browser. To display the data set on the page, you must code tags and variables to output the data.

Code Review

The query you just created retrieves data from the CompanyInfo database.

Code	Description
`<CFQUERY NAME="EmpList" DATASOURCE="CompanyInfo">`	Query the database specified in the CompanyInfo datasource
`SELECT FirstName, LastName, Salary, Contract FROM Employees`	Get information from the FirstName, LastName, Salary, and Contract fields in the Employees table
`</CFQUERY>`	End the CFQUERY block

Query Notes and Considerations

When creating queries to retrieve data, keep these guidelines in mind:

- Enter the query NAME and DATASOURCE attributes in the begin CFQUERY tag.
- Surround attribute settings with double quotes(").
- Reference the query data by naming the query in the CFOUTPUT tag later on the page.
- Make sure that a data source exists in the ColdFusion Administrator before you reference iit n a CFQUERY tag.
- The SQL that you write is sent to the database and performs the actual data retrieval.
- Columns and tables that you refer to in your SQL statement must exist, otherwise the query will fail.

Outputting Query Data

After you have defined a query on a page, you can use the CFOUTPUT tag with the QUERY attribute to define the query variable that you want to output to a page. When you use the QUERY attribute:

- ColdFusion loops over all the code contained within the CFOUTPUT block, once for each row returned from a database.
- Reference specific column names within the CFOUTPUT block to output the data to the page.
- You can place text and HTML tags inside or surrounding the CFOUTPUT block to format the data on the page.

The CFOUTPUT tag accepts a variety of optional attributes but, ordinarily, you will use the QUERY attribute to define the name of an existing query.

To output query data on your page:

1. Return to `empList.cfm` in Studio.
2. Edit the file so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>Employee List</TITLE>
</HEAD>
<BODY>
<H1>Employee List</H1>
<CFQUERY NAME="EmpList" DATASOURCE="CompanyInfo">
    SELECT FirstName, LastName, Salary, Contract
    FROM Employees
</CFQUERY>
<CFOUTPUT QUERY="EmpList">
```

```
    #FirstName#, #LastName#, #Salary#, #Contract#<BR>
    </CFOUTPUT>
    </BODY>
    </HTML>
```

3. Save the file as `emplist.cfm`.
4. View the page in a browser.

   A list of employees appears in the browser, with each line displaying one row of data.

You have created a ColdFusion application page that retrieves and displays data from a database. At present, the output is raw. You will learn how to format the data in the next chapter.

Code Review

You now display the results of the query on the page.

Code	Description
`<CFOUTPUT QUERY="EmpList">`	Display information retrieved in the EmpList query
`#FirstName#, #LastName#, #Salary#, #Contract#`	Display the value of the `FirstName`, `LastName`, `Salary`, `Contract` fields of the first record
` `	Insert a line break (go to the next line Then, keep displaying the fields you've specified for each record, followed by a line break, until you run out of records.
`</CFOUTPUT>`	End the CFOUTPUT block

Query Output Notes and Considerations

When outputting query results, keep these guidelines in mind:

- Run a CFQUERY before referencing its results using a CFOUTPUT with a QUERY attribute.
- It's a good idea to run all queries before all output blocks.
- A query name must exist on the page in order to successfully output its data.
- Surround variable references with pound signs to output their current values to a page.
- Prefix variables with their variable type — in the case of a query variable, it's the name of the query.

- When outputting the data itself, you define the variable name using the QUERY attribute.
- When outputting query properties variables, don't use the QUERY attribute; instead, prefix the variable reference with the name of the query, for example, `EmpList.RecordCount`.
- Columns must exist and be retrieved to the application in order to output their values.
- As with other attributes, surround the QUERY value with double quotes (").
- As with any variables that you reference for output, surround column names with pound signs (#) to tell ColdFusion to output the column's current values.
- Add a
 tag to the end of the variable references so that ColdFusion will start a new line for each row that is returned from the query.

Getting Information About Query Results

Each time you query a database with the CFQUERY tag, you get not only the data itself, but also query properties, as described in the following table::

Query Properties	
Property	**Description**
RecordCount	The total number of records returned by the query.
ColumnList	Returns a comma-delimited list of the query columns.
CurrentRow	The current row of the query being processed by CFOUTPUT.

To output query data on your page:

1. Return to `emplist.cfm` in Studio.
2. Edit the file so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>Employee List</TITLE>
</HEAD>
<BODY>
<H1>Employee List</H1>
<CFQUERY NAME="EmpList" DATASOURCE="CompanyInfo">
    SELECT FirstName, LastName, Salary, Contract
    FROM Employees
</CFQUERY>
<CFOUTPUT QUERY="EmpList">
    #FirstName#, #LastName#, #Salary#, #Contract#<BR>
</CFOUTPUT>
```

```
<CFOUTPUT>
The query returned #EmpList.RecordCount# records.
</CFOUTPUT>
    </BODY>
    </HTML>
```

3. Save the file as `emplist.cfm`.
4. View the page in a browser.

The number of employees now appears below the list of employees.

Code Review

You now display the number of records retrieved in the query.

Code	Description
`<CFOUTPUT>`	Display what follows
`The query returned`	Display the text "The query returned"
`#EmpList.RecordCount#`	Display the number of records retrieved in the EmpList query
`records.`	Display the text "records"
`</CFOUTPUT>`	End the CFOUTPUT block.

Query Properties Notes and Considerations

Keep the following in mind when using query properties:

- Prefix the property with its type — in this case — prefix the property with the name of the query.
- Reference the query property within a CFOUTPUT block so that ColdFusion will output the query variable value to the page.
- Surround the query property reference with pound signs (#) so that ColdFusion knows to replace the property name with its current value.

CHAPTER 4

Retrieving and Formatting the Data You Want

This chapter explains how to select the data to display in a dynamic Web page. It also describes how to populate an HTML table with query results and how to use ColdFusion functions to format and manipulate data.

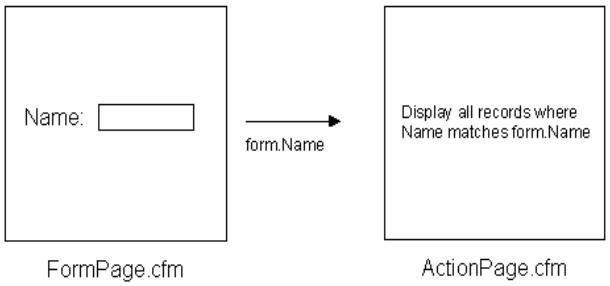
Contents

Using Forms to Specify the Data to Retrieve

Until now, you've retrieved all of the records from a table. However, there are many instances when you'll want to retrieve data based on certain criteria. For example, you may want to see records for everyone in a particular department, everyone in a particular town whose last name is Smith, or books by a certain author.

You can use forms in ColdFusion applications to allow users to specify what data they want to retrieve in a query.

When you submit a form, you pass the variables to an associated page, called an action page, where some type of processing takes place.



Note Because forms are not ColdFusion-specific, the syntax and examples that follow provide you with just enough detail to get going with ColdFusion.

FORM tag syntax

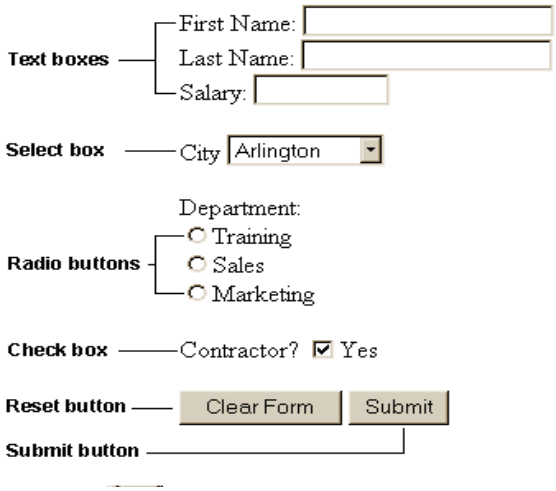
```
<FORM ACTION="actionpage.cfm" METHOD="Post">
    ...
</FORM>
```

- Use the ACTION attribute to specify an action page to which you pass form variables for processing.
- Use the METHOD attribute to specify how the variables are submitted from the browser to the action page on the server.

  All ColdFusion forms must be submitted with an attribute setting of METHOD="Post".

Form Controls

Within the form, you'll describe the form controls needed to gather and submit user input. There are a variety of form controls types available. You choose form control input types based on the type of input the user should provide.



HTML Form Controls	
Control	**Code**
Text control	`<INPUT TYPE="Text" NAME="ControlName" SIZE="Value" MAXLENGTH="Value">`

Radio buttons	`<INPUT TYPE="Radio" NAME="ControlName" VALUE="Value1">DisplayName1`
	`<INPUT TYPE="Radio" NAME="ControlName" VALUE="Value2">DisplayName2`
	`<INPUT TYPE="Radio" NAME="ControlName" VALUE="Value3">DisplayName3`

Select box	`<SELECT NAME="ControlName">`
	`  <OPTION VALUE="Value1">DisplayName1`
	`  <OPTION VALUE="Value2">DisplayName2`
	`  <OPTION VALUE="Value3">DisplayName3`
	`</SELECT>`

Check box	`<INPUT TYPE="Checkbox" NAME="ControlName" VALUE="Yes|No">Yes`
Reset button	`<INPUT TYPE="Reset" NAME="ControlName" VALUE="DisplayName">`
Submit button	`<INPUT TYPE="Submit" NAME="ControlName" VALUE="DisplayName">`

To create a form:

1. Create a new application page, using Studio.
2. Edit the page so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>Input form</TITLE>
</HEAD>
<BODY>
<!--- define the action page in the form tag. The form variables will
pass to this page when the form is submitted --->

<form action="actionpage.cfm" method="post">

<!-- text box -->
<p>
First Name: <INPUT TYPE="Text" NAME="FirstName" SIZE="20"
MAXLENGTH="35"><br>
Last Name: <INPUT TYPE="Text" NAME="LastName" SIZE="20"
MAXLENGTH="35"><br>
Salary: <INPUT TYPE="Text" NAME="Salary" SIZE="10" MAXLENGTH="10">
</P>

<!-- select box -->
City
<SELECT NAME="City">
    <OPTION VALUE="Arlington">Arlington
    <OPTION VALUE="Boston">Boston
    <OPTION VALUE="Cambridge">Cambridge
    <OPTION VALUE="Minneapolis">Minneapolis
    <OPTION VALUE="Seattle">Seattle
</SELECT>

<!-- radio buttons -->
<p>
Department:<br>
<input type="radio" name="Department" value="Training">Training<br>
<input type="radio" name="Department" value="Sales">Sales<br>
<input type="radio" name="Department" value="Marketing">Marketing<br>
</p>

<!-- check box -->
<P>
Contractor? <input type="checkbox" name="Contractor" value="Yes"
checked>Yes
</P>

<!-- reset button -->
<INPUT TYPE="Reset" NAME="ResetForm" VALUE="Clear Form">

<!-- submit button -->
<INPUT TYPE="Submit" NAME="SubmitForm" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```

3. Save the page as `formpage.cfm` within the `myapps` directory under your Web root directory.

4. View the form in a browser.

   The form appears in the browser.

   Remember that you need an action page in order to submit values; you will create one later in this chapter.

Code Review

A form appears on the page, ready for user input.

Code	Description
`<FORM ACTION="actionpage.cfm" METHOD="POST">`	Gather the information from this form using the Post method, and do something with it on the page actionpage.cfm.
`<INPUT TYPE="Text" NAME="FirstName" SIZE="20" MAXLENGTH="35">`	Create a text box called FirstName where users can enter their first name. Make it 20 characters wide, but allow input of up to 35 characters.
`<INPUT TYPE="Text" NAME="LastName" SIZE="20" MAXLENGTH="35">`	Create a text box called LastName where users can enter their first name. Make it 20 characters wide, but allow input of up to 35 characters.
`<INPUT TYPE="Text" NAME="Salary" SIZE="10" MAXLENGTH="10">`	Create a text box called Salary where users can enter a salary to look for. Make it 10 characters wide, and allow input of up to 10 characters.

`<SELECT NAME="City">`
`  <OPTION VALUE="Arlington">Arlington`
`  <OPTION VALUE="Boston">Boston`
`  <OPTION VALUE="Cambridge">Cambridge`
`  <OPTION VALUE="Minneapolis">Minneapolis`
`  <OPTION VALUE="Seattle">Seattle`
`</SELECT>`
	Create a drop down select box named City and populate it with the values "Arlington," "Boston," "Cambridge," "Minneapolis," and "Seattle."

`<input type="checkbox" name="Contractor" value="Yes|No" checked>Yes`	Create a checkbox that allows users to specify whether they want to list employees who are contractors. Have the box checked by default.
`<INPUT TYPE="Reset" NAME="ResetForm" VALUE="Clear Form">`	Create a reset button to allow users to clear the form. Put the text "Clear Form" on the button.
`<INPUT TYPE="Submit" NAME="SubmitForm" VALUE="Submit">`	Create a submit button to send the values users enter to the action page for processing. Put the text "Submit" on the button.

Form Notes and Considerations

- To make the coding process easy to follow, name form controls the same as target database fields.
- Limit radio buttons to three-to-five mutually exclusive options.

  If you need more than that many options, consider a dropdown select box.
- Use select boxes to allow the user to choose multiple items.
- All the data that you collect on a form is automatically passed as form variables to the associated action page.
- Checkboxes and radio buttons do not pass to action pages unless they are selected on a form. In fact, if you try to reference these variables on the action page, you will receive an error if they are not present.
- You can dynamically populate dropdown select boxes using query data. See "Dynamically Populating Select Boxes" on page 46 for details.

Processing Form Variables on Action Pages

A ColdFusion action page is just like any other application page except that you can use the form variables that are passed to it from an associated form. A form variable is passed for every form control that contains a value when the form is submitted.

Note If multiple controls have the same name, one form variable is passed to the action page. It contains a comma delimited list.

A form variable's name is the name that you assigned to the form control on the form page. Refer to form variable by name within tags, functions and other expressions on an action page.

Because form variables extend beyond the local page — their scope is the action page — prefix them with "form." to explicitly tell ColdFusion that you are referring to a form variable. For example this code references the LastName form variable for output on an action page:

```
<CFOUTPUT>
    #Form.LastName#
</CFOUTPUT>
```

Dynamically Generating SQL Statements

As you've already learned, you can retrieve a record for every employee in a database table by composing a query like this:

```
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
    SELECT  FirstName, LastName, Contract
    FROM   Employees
</CFQUERY>
```

But when you want to return information about employees that match user search criteria, you use the SQL WHERE clause with a SQL SELECT statement to compare a value against a character string field. When the WHERE clause is processed, it filters the query data based on the results of the comparison.

For example, to return employee data for only employees with the last name of Allaire, you would build a query that looks like this:

```
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
    SELECT FirstName, LastName, Contract
    FROM    Employees
    WHERE  LastName = 'Allaire'
</CFQUERY>
```

However, instead of putting the LastName directly in the SQL WHERE clause, you can use the text the user entered in the form for comparison:

```
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
    SELECT FirstName, LastName, Salary
    FROM Employees
    WHERE LastName='#Form.LastName#'
</CFQUERY>
```

Creating Action Pages

To create an action page for the form:

1. Create a new application page in Studio.
2. Enter the following code:

```
<HTML>
<HEAD>
<TITLE>Retrieving Employee Data Based on Criteia from Form</TITLE>
</HEAD>

<BODY>
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
    SELECT FirstName, LastName, Salary
    FROM Employees
    WHERE LastName='#Form.LastName#'
</CFQUERY>
<H4>Employee Data Based on Criteria from Form</H4>
<CFOUTPUT query="GetEmployees">
#FirstName#
#LastName#
#Salary#<BR>
</CFOUTPUT>
</BODY>
</HTML>
```

3. Save the page as `actionpage.cfm` within the `myapps` directory.
4. View `formpage.cfm` in your browser.

5. Enter data for the LastName form control and submit it.
6. Return to the form in your browser.
7. Reset the values.
8. Do not check the checkbox and submit the form again.

   An error occurs when the checkbox does not pass to the action page.

   You will receive errors if you submit the form without checking the checkbox form controls. You will learn how to apply conditional logic to your action page to compensate for this HTML limitation in "Testing for a variable's existence" on page 51.

Code Review

Code	Description
`<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">`	Query the datasource CompanyInfo and name the query GetEmployees.
`SELECT FirstName, LastName, Salary FROM Employees WHERE LastName='#Form.LastName#'`	Retrieve the FirstName, LastName, and Salary fields from the Employees table, but only if the value of the LastName field matches what the user entered in the LastName text box in the form on `formpage.cfm`.
`<CFOUTPUT query="GetEmployees">`	Display results of the GetEmployees query.
`#FirstName# #LastName# #Salary# `	Display the value of the FirstName, LastName, and Salary fields for a record, starting with the first record, then go to the next line. Keep displaying the records that match the criteria you specified in the SELECT statement, followed by a line break, until you run out of records
`</CFOUTPUT>`	Close the CFOUTPUT block

Form Variable Notes and Considerations

When using form variables, keep the following guidelines in mind:

- A form variable's scope is the action page.
- Prefix form variables with "form." when referencing them on the action page.
- Surround variable values with pound signs (#) for output.
- Checkboxes and radio buttons are only passed to the action page if an option is selected.

- Form variables for checkboxes and radio buttons generate errors on action pages if nothing is selected for the form controls.

Using HTML Tables to Layout Query Results

You have displayed each row of data from the Employees table, but the information was unformatted. You can use HTML tables to control the layout of information on the page. In addition, you can use CFML functions to format individual pieces of data such as dates and numeric values.

You can use HTML tables to specify how the results of a query appear on a page. To do so, you put the CFOUTPUT tag *inside* the table tags. You can also use the HTML TH tag to put column labels in a header row. To create a row in the table for each row in the query results, put the TR block inside the CFOUTPUT tag.

To put the query results in a table:

1. Return to the file `emplist.cfm` in Studio.
2. Modify the page so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>Retrieving Employee Data Based on Criteia from Form</TITLE>
</HEAD>

<BODY>
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
    SELECT FirstName, LastName, Salary
    FROM Employees
    WHERE LastName='#Form.LastName#'
</CFQUERY>
<H4>Employee Data Based on Criteia from Form</H4>
<TABLE>
<TR>
<TH>First Name</TH>
<TH>Last Name</TH>
<TH>Salary</TH>
</TR>
<CFOUTPUT QUERY="GetEmployees">
<TR>
<TD>#FirstName#</TD>
<TD>#LastName#</TD>
<TD>#Salary#</TD>
</TR>
</CFOUTPUT>
</TABLE>
</BODY>
</HTML>
```

3. Save the page as `actionpage.cfm` within the `myapps` directory.

4. View `formpage.cfm` in your browser.
5. Enter data for the LastName form control and submit it.
6. The records that match the criteria specified in the form appear in a table.

Code Review

Code	Description
`<TABLE>`	Put data into a table.

`<TR>`
`  <TH>First Name</TH>`
`  <TH>Last Name</TH>`
`  <TH>Salary</TH>`
`</TR>`
	In the first row of the table, include three columns, with the headings: First Name, Last Name, and Salary.

`<CFOUTPUT QUERY="GetEmployees">`	Get ready to display the results of the GetEmployees query.

`<TR>`
`  <TD>#FirstName#</TD>`
`  <TD>#LastName#</TD>`
`  <TD>#Salary#</TD>`
`</TR>`
	Create a new row in the table, with three columns. For a record, put the value of the FirstName field, the value of the LastName field, and the value of the Salary field.

`</CFOUTPUT>`	Keep getting records that matches the criteria, and display each row in a new table row until you run out of records.
`</TABLE>`	End of table.

Formatting Individual Data Items

You may want to format individual data items. For example, you can format the Salary field as a monetary value.

To format the Salary using the dollar format, you use the CFML expression `DollarFormat(number)`.

To change the format of the Salary:

1. Return to `actionpage.cfm` in Studio.
2. Change the line `<TD>#Salary#</TD>` to `<TD>#DollarFormat(Salary)#</TD>`

Performing Pattern Matching

Use the SQL LIKE operator and SQL wildcard strings in a SQL WHERE clause when you want to compare a value against a character string field so that the query returns database information based on commonalities. This is known as pattern matching and often used to query databases.

For example, to return data for employees whose last name starts with AL and ends with anything, you would build a query that looks like this:

```
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
    SELECT FirstName, LastName,
    StartDate, Salary, Contract
    FROM    Employees
    WHERE   LastName LIKE 'AL%'
</CFQUERY>
```

- The LIKE operator tells the database that the string that follows should be used for pattern matching.
- The LIKE operator tells the database that the string that follows should be used for pattern matching.
- If you placed a wildcard before and after AL, you would retrieve any record in that column that contains AL.
- Surround strings in SQL statements with single quotes (').
- When comparing a value against a numeric field, don't surround the value with single quotes (').

Note By default, SQL is not case-sensitive.

Filtering Data Based on Multiple Conditions

Combine a SQL WHERE clause with a SQL AND clause in your queries when you want to retrieve data based on the results of more than one comparison.

For example, to return data for contract employees who earn more than $50,000, you would build a query that looks like this:

```
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
    SELECT FirstName, LastName
    StartDate, Salary, Contract
    FROM    Employees
    WHERE   Contract = 'Yes'
    AND     Salary > 50000
</CFQUERY>
```

Creating Table Joins

Many times, the data that you want to retrieve is maintained in multiple tables. For example, in the database that you're working with:

- Department information is maintained in the Departments table. This includes department ID numbers.
- Employee information is maintained in the Employees table. This also includes department ID numbers.

To compare and retrieve data from more than one table during a query, use the WHERE clause to join two tables through common information.

For example, to return employee names, start dates, department names, and salaries for employees that work for the HR department, you would build a query that looks like this:

```
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
    SELECT Departments.Department.Name,
    Employees.FirstName,
    Employees.LastName,
    Employees.StartDate,
    Employees.Salary
    FROM Departments, Employees
    WHERE Departments.Department_ID = Employees.Department_ID
    AND Departments.Department_Name = 'HR'
</CFQUERY>
```

- Prefix each column in the SELECT statement to explicitly state which table the data should be retrieved from.
- The Department_ID field is the primary key of the Departments table and the Foreign Key of the Employees table.

Building Flexible Search Interfaces

Frequently, you will want users to optionally enter multiple search criteria.

Wrap conditional logic around the SQL AND clause to build a flexible search interface. To test for multiple conditions, wrap additional CFIF tags around additional AND clauses.

For example, to allow users to search for employees by last name, department, or both, you would build a query that looks like this:

```
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
    SELECT  Departments.Department.Name,
    Employees.FirstName,
    Employees.LastName,
    Employees.StartDate,
    Employees.Salary
    FROM Departments, Employees
    WHERE 1=1
```

```
    <CFIF Form.LastName IS NOT "">
    AND Employees.LastName = 'Form.LastName'
    </CFIF>
</CFQUERY>
```

Code Review

Code	Description

`SELECT  Departments.Department.Name,`
`  Employees.FirstName,`
`  Employees.LastName,`
`  Employees.StartDate,`
`  Employees.Salary`
`  FROM Departments, Employees`
`WHERE 1=1`
	Retrieve the fields listed from the Departments and Employees tables, joining the tables based on the Department_ID field in each table.

`<CFIF Form.LastName IS NOT "">`
`AND Employees.LastName = 'Form.LastName'`
`</CFIF>`
	But if the user specified a last name in the form, only retrieve the records where the last name is the same as the one the user entered in the form.

To build a flexible search interface:

1. Return to `actionpage.cfm` in Studio.
2. Modify the page so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>Retrieving Employee Data Based on Criteria from Form</TITLE>
</HEAD>

<BODY>
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
SELECT  Departments.Department.Name,
Employees.FirstName,
Employees.LastName,
Employees.StartDate,
Employees.Salary
FROM Departments, Employees
WHERE Departments.Department_ID = Employees.Department_ID
<CFIF Form.Department_Name IS NOT "">
AND Departments.Department_Name = 'Form.Department_Name'
</CFQUERY>
<H4>Employee Data Based on Criteia from Form</H4>
<TABLE>
<TR>
    <TH>First Name</TH>
    <TH>Last Name</TH>
<TH>Salary</TH>
```

```
</TR>
<CFOUTPUT QUERY="GetEmployees">
<TR>
    <TD>#FirstName#</TD>
    <TD>#LastName#</TD>
    <TD>#Salary#</TD>
</TR>
</CFOUTPUT>
</TABLE>
</BODY>
</HTML>
```

3. Save the file.
4. Test the search interface in your browser.

   The returned records will not be displayed because you have not entered that code yet, however, you will see the number of records returned if you have debugging enabled.

Returning Query Results to the User

When you build search interfaces, keep in mind that there won't always be a record returned. If there is at least one record returned from a query, you will usually format that data using an HTML table. But to make sure that a search has retrieved records, you will need to test if any records have been returned using the recordcount variable in a conditional logic expression in order to display search results appropriately to users.

For example, to inform the user that no records were found if the number of records returned for the GetEmployees query is 0, insert the following code before displaying the data:

```
<CFIF GetEmployees.RecordCount IS "0">
    No records match your search criteria. <BR>
<CFELSE>
```

- Prefix RecordCount with the queryname.
- Add a true procedure that displays a message to the user.
- Add a not true procedure after the CFELSE tag to format the returned data using an HTML table.

To return search results to users:

1. Return to `actionpage.cfm` in Studio.
2. Add the code indicated.

```
<HTML>
<HEAD>
<TITLE>Retrieving Employee Data Based on Criteia from Form</TITLE>
</HEAD>
```

```
<BODY>
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
SELECT  Departments.Department.Name,
    Employees.FirstName,
    Employees.LastName,
    Employees.StartDate,
    Employees.Salary
    FROM Departments, Employees
WHERE Departments.Department_ID = Employees.Department_ID
    <CFIF Form.Department_Name IS NOT "">
    AND Departments.Department_Name = 'Form.Department_Name'
</CFQUERY>
<H4>Employee Data Based on Criteia from Form</H4>
<CFIF GetEmployees.RecordCount IS "0">
No records match your search criteria. <br>
Please go back to the form and try again.
<CFELSE>
<TABLE>
<TR>
    <TH>First Name</TH>
    <TH>Last Name</TH>
    <TH>Salary</TH>
</TR>
<CFOUTPUT QUERY="GetEmployees">
<TR>
    <TD>#FirstName#</TD>
    <TD>#LastName#</TD>
    <TD>#Salary#</TD>
</TR>
</CFOUTPUT>
</TABLE>
</BODY>
</HTML>
```

3. Save the file.
4. Return to the form, enter search criteria and submit the form.
5. If no records match the criteria you specified, the message displays.

CHAPTER 5

Making Variables Dynamic

This chapter explains how to use CFML to dynamically populate forms and dynamically generate SQL.

Contents

Dynamically Populating Select Boxes

In the previous chapter, you hard-coded a form's select box options.

Instead of manually entering the information on a form, you can dynamically populate a select box with database fields. When you code this way, changes that you make to a database are automatically reflected on the form page.

To dynamically populate a select box:

- Use the CFQUERY tag to retrieve the column data from a database table.
- Use the CFOUTPUT tag with the QUERY attribute within the SELECT tag to dynamically populate the OPTIONS of this form control.

To dynamically populate a select box:

1. Open the file `formpage.cfm` in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>Input form</TITLE>
</HEAD>
<BODY>
<CFQUERY NAME="GetDepartments" DATASOURCE="CompanyInfo">
SELECT Location
FROM Departments
</CFQUERY>

<!--- define the action page in the form tag. The form variables will
pass to this page when the form is submitted --->

<form action="actionpage.cfm" method="post">

<!-- text box -->
<P>
First Name: <INPUT TYPE="Text" NAME="FirstName" SIZE="20"
MAXLENGTH="35"><BR>
Last Name: <INPUT TYPE="Text" NAME="LastName" SIZE="20"
MAXLENGTH="35"><BR>
Salary: <INPUT TYPE="Text" NAME="Salary" SIZE="10" MAXLENGTH="10">
</P>

<!-- select box -->
City

<SELECT NAME="City">
<CFOUTPUT QUERY="GetDepartments">
<OPTION VALUE="#GetDepartments.Location#>
#GetDepartments.Location#
</OPTION>
</CFOUTPUT>
```

```
</SELECT>

<!-- radio buttons -->
<P>
Department:<BR>
<INPUT TYPE="radio" name="Department" value="Training">Training<BR>
<INPUT TYPE="radio" name="Department" value="Sales">Sales<BR>
<INPUT TYPE="radio" name="Department" value="Marketing">Marketing<BR>
</P>

<!-- check box -->
<P>
Contractor? <input type="checkbox" name="Contractor" value="Yes"
checked>Yes
</P>

<!-- reset button -->
<INPUT TYPE="reset" NAME="ResetForm" VALUE="Clear Form">

<!-- submit button -->
<INPUT TYPE="submit" NAME="SubmitForm" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```

3. Save the page as `formpage.cfm`.
4. View `formpage.cfm` in a browser.

   The changes that you just made appear in the form.

   Remember that you need an action page to submit values.

Creating Dynamic Checkboxes and Multiple Select Boxes

When an HTML form contains either a list of checkboxes with the same name or a multiple select box, the user's entries are made available as a comma-delimited list with the selected values. These lists can be very useful for a wide range of inputs.

Note If no value is entered for a checkbox or multiple select lists, no variable is created.The SQL INSERT statement will not work correctly if there are no values. To correct this problem, make the form fields required, use Dynamic SQL, or use CFPARAM to establish a default value for the form field..

Checkboxes

When you put a series of checkboxes with the same name in an HTML form, the variable that is created contains a comma-delimited list of values. The values can be either numeric values or alphanumeric strings. These two types of values are treated slightly differently.

Searching numeric values

Suppose you want a user to select one or more departments using checkboxes. You query the database to retrieve detailed information on the selected department(s).

```
Select one or more departments to get information on:

<INPUT TYPE="checkbox"
    NAME="SelectedDept"
    VALUE="1">
    Training<BR>

<INPUT TYPE="checkbox"
    NAME="SelectedDept"
    VALUE="2">
    Marketing<BR>

<INPUT TYPE="checkbox"
    NAME="SelectedDept"
    VALUE="3">
    HR<BR>

<INPUT TYPE="checkbox"
    NAME="SelectedDept"
    VALUE="4">
    Sales<BR>

<INPUT TYPE="hidden"
    NAME="SelectedDepts_required"
    VALUE="You must select at least one organization.">
```

The text displayed to the user is the name of the department, but the VALUE attribute of each checkbox corresponds to the underlying database primary key for the department's record.

If the user checked the Marketing and Sales items, the value of the SelectedDept form field would be "2,4." If this parameter were used, the following would be the resulting SQL statement:

```
SELECT *
    FROM Departments
    WHERE Department_ID IN ( #form.SelectedDept# )
```

The statement sent to the database would be:

```
SELECT *
    FROM Departments
    WHERE Department_ID IN ( 2,4 )
```

Searching string values

To search for a database field containing string values (instead of numeric), you must modify both the checkbox and CFQUERY syntax.

The first example searched for department information based on a numeric primary key field called "Department_ID." Suppose, instead, that the primary key is a database field called "DepartmentName" that contains string values. In that case, it's necessary to make the following modifications:

- Make the value attribute of the checkboxes equal to the string value.
- Enclose the value attribute in single quotes.

```
<INPUT TYPE="checkbox"
    NAME="SelectedDepts"
    VALUE="'Training'">
    Training<BR>

<INPUT TYPE="checkbox"
    NAME="SelectedDepts"
    VALUE="'Marketing'">
    Marketing<BR>

<INPUT TYPE="checkbox"
    NAME="SelectedDepts"
    VALUE="'HR'">
    HR<BR>

<INPUT TYPE="checkbox"
    NAME="SelectedDepts"
    VALUE="'Sales'">
    Sales<BR>

<INPUT TYPE="checkbox"
    NAME="SelectedDepts_required"
    VALUE="You must select at least one organization.">
```

If the user checked Marketing and Sales, the value of the SelectedDepts form field would be 'Marketing','Sales'.

Note You must use the ColdFusion PreserveSingleQuotes function in the SQL statement to prevent ColdFusion from escaping the single quotes in the form field value:

```
SELECT *
    FROM Departments
    WHERE DepartmentName IN
    (#PreserveSingleQuotes(form.SelectedDepts)#)
```

The statement sent to the database would be:

```
SELECT *
    FROM Departments
    WHERE DepartmentName IN ('Marketing','Sales')
```

Multiple select lists

ColdFusion treats multiple select lists (HTML input type SELECT with attribute MULTIPLE) just like checkboxes. The data made available to your page from any

multiple select list is a comma-delimited list of the entries selected by the user. For example, a multiple select list contains four entries: Training, Marketing, HR, and Sales. The user selects Marketing and Sales. The value of the form field variable is then 'Marketing', 'Sales'.

Just as you can with checkboxes, you can also query with multiple select lists by searching a database field that contains either numeric values or string values.

Searching numeric values

For example, suppose you want the user to select departments from a multiple select box. The query retrieves detailed information on the selected department(s):

```
Select one or more companies to get more information on:
<SELECT Name="SelectDepts" MULTIPLE>
    <OPTION VALUE="1">Training
    <OPTION VALUE="2">Marketing
    <OPTION VALUE="3">HR
    <OPTION VALUE="4">Sales
</SELECT>

<INPUT TYPE="hidden"
    NAME="SelectDepts_required"
    VALUE="You must select at least one department.">
```

If the user selected the Marketing and Sales items, the value of the SelectDepts form field would be 2,4.

If this parameter were used in the following SQL statement:

```
SELECT *
    FROM Departments
    WHERE Department_ID IN (#form.SelectDepts#)
```

the statement sent to the database would be:

```
SELECT *
    FROM Departments
    WHERE Department_ID IN (2,4)
```

Searching string values

Suppose you want the user to select departments from a multiple select list. The database field search is a string field. The query retrieves detailed information on the selected department(s):

```
Select one or more departments to get
more information on:
<SELECT Name="SelectDepts" MULTIPLE>
    <OPTION VALUE="'Training'">Training
    <OPTION VALUE="'Marketing'">Marketing
    <OPTION VALUE="'HR'">HR
```

```
    <OPTION VALUE="'Sales'">Sales
</SELECT>

<INPUT TYPE="hidden"
    NAME="SelectDepts_required"
    VALUE="You must select at least one department.">
```

If the user selected the Marketing and Sales items, the value of the SelectDepts form field would be 'Marketing','Sales'.

Just as you did when using checkboxes to search database fields containing string values, use the ColdFusion PreserveSingleQuotes function with multiple select boxes:

```
SELECT *
    FROM Departments
    WHERE DepartmentName IN (#PreserveSingleQuotes(form.SelectDepts)#)
```

The statement sent to the database would be:

```
SELECT *
    FROM Departments
    WHERE DepartmentName IN ('Marketing', 'Sales')
```

Testing for a variable's existence

Before relying on a variable's existence in an application page, you can test to see if it exists using the IsDefined function. For example, the following code checks to see if a Form variable named Order_ID exists:

```
<CFIF Not IsDefined("FORM.Order_ID")>
    <CFLOCATION URL="previous_page.cfm">
</CFIF>
```

The argument passed to the IsDefined function must always be enclosed in double quotes. See the *CFML Language Reference* for more information on the IsDefined function.

If you attempt to evaluate a variable that has not been defined, ColdFusion will not be able to process the page. To help diagnose such problems, use the interactive debugger in ColdFusion Studio or turn debugging on in the ColdFusion Administrator. The Administrator debugging information shows which variables are being passed to your application pages.

Creating Default Variables with CFPARAM

Another way to create a variable is to test for its existence and optionally supply a default value if the variable does not already exist. The following shows the syntax of the CFPARAM tag:

```
<CFPARAM NAME="VariableName"
    TYPE="data_type"
    DEFAULT="DefaultValue">
```

There are two ways to use the CFPARAM tag, depending on how you want the validation test to proceed.

- Use CFPARAM with only the NAME attribute to test that a required variable exists. If it does not exist, the ColdFusion server stops processing the page.
- Use CFPARAM with both the NAME and DEFAULT attributes to test for the existence of an optional variable. If the variable exists, processing continues and the value is not changed. If the variable does not exist, it is created and set to the value of the DEFAULT attribute.

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

```
<CFPARAM NAME="Form.Contract" DEFAULT="Yes">
```

Example: Testing for variables

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

For example, the following series of CFPARAM tags indicates that this page expects two form variables named StartRow and RowsToFetch:

```
<CFPARAM NAME="Form.StartRow">
<CFPARAM NAME="Form.RowsToFetch">
```

If the page with these tags is called without either one of the form variables, an error occurs and the page stops processing.

Example: Setting default values

In this example, CFPARAM is used to see if optional variables exist. If they do exist, processing continues. If they do not exist, they are created and set to the DEFAULT value.

```
<CFPARAM NAME="Cookie.SearchString" DEFAULT="temple">

<CFPARAM NAME="Client.Color" DEFAULT="Grey">

<CFPARAM NAME="ShowExtraInfo" DEFAULT="No">
```

You can also use CFPARAM to set default values for URL and Form variables, instead of using conditional logic.

Checking Query Parameters with CFQUERYPARAM

You can use the CFQUERYPARAM tag to check data types of query parameters and perform data validation.

Example: Checking data types

```
<!---------------------------------------------------------------------
This example shows the use of CFQUERYPARAM when valid input is given in
Dept_ID.
---------------------------------------------------------------------->
<HTML>
<HEAD>
<TITLE>CFQUERYPARAM Example</TITLE>
</HEAD>

<BODY>
<H3>CFQUERYPARAM Example</H3>
<CFSET Course_ID=12>
<CFQUERY NAME="getFirst" DataSource="CompanyInfo">
    SELECT *
    FROM departments
    WHERE Dept_ID=<CFQUERYPARAM VALUE="#Dept_ID#"
    CFSQLTYPE="CF_SQL_INTEGER">
</CFQUERY>
<CFOUTPUT QUERY="getFirst">
<p>Department Number: #number#<br>
 Description: #descript#
</P>
</CFOUTPUT>
</BODY>
</HTML>
```

Dynamic SQL

Embedding SQL queries that use dynamic parameters is a powerful mechanism for linking variable inputs to database queries. However, in more sophisticated applications, you will often want user inputs to determine not only the content of queries but also the structure of queries.

Dynamic SQL allows you to dynamically determine (based on runtime parameters) which parts of a SQL statement are sent to the database. So if a user leaves a search field empty, for example, you could simply omit the part of the WHERE clause that refers to that field. Or, if a user does not specify a sort order, the entire ORDER BY clause could be omitted.

Dynamic SQL is implemented in ColdFusion by using CFIF, CFELSE, CFELSEIF tags to control how the SQL statement is constructed, for example:

```
<CFQUERY NAME="queryname"
    DATASOURCE="datasourcename">
...Base SQL statement

<CFIF value operator value >
...additional SQL
</CFIF>

</CFQUERY>
```

First, you need to create an input form, which asks for information about several fields in the Employees table. Instead of entering information in each field, a user may want to search on certain fields, or even on only one field. To search for data based on only the fields the user enters in the form, you use CFIF statements in the SQL statement.

To create the input form:

1. Create a new application page in Studio.
2. Enter the following code:

```
<HTML>
<HEAD>
<TITLE>Input form</TITLE>
</HEAD>
<BODY>
<!--- Query the Employees table to be able to populate the form --->
<CFQUERY NAME="AskEmployees" DATASOURCE="CompanyInfo">
SELECT
    FirstName,
    LastName,
    Salary,
    Contract
FROM        Employees
</CFQUERY>

<!--- define the action page in the form tag. The form variables will
pass to this page when the form is submitted --->

<FORM ACTION="getemp.cfm" METHOD="post">

<!-- text box -->
<P>
First Name: <INPUT TYPE="Text" NAME="FirstName" SIZE="20"
MAXLENGTH="35"><BR>
Last Name: <INPUT TYPE="Text" NAME="LastName" SIZE="20"
MAXLENGTH="35"><BR>
Salary: <INPUT TYPE="Text" NAME="Salary" SIZE="10" MAXLENGTH="10">
</P>

<!-- check box -->
<P>
Contractor? <input type="checkbox" name="Contract" value="Yes" >Yes
if checked
</P>
```

```
<!-- reset button -->
<INPUT TYPE="reset" NAME="ResetForm" VALUE="Clear Form">

<!-- submit button -->
<INPUT TYPE="submit" NAME="SubmitForm" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```

3. Save the page as `askemp.cfm`.

Once you have created the input form, you can then create the action page to process the user's request. This action page will determine where the user has entered search criteria and search based only on those criteria.

To create the action page:

1. Create a new application page in Studio.
2. Enter the following code:

```
<HTML>
<HEAD>
    <TITLE>Get Employee Data</TITLE>
</HEAD>

<BODY>
<CFQUERY NAME="GetEmployees" DATASOURCE="CompanyInfo">
SELECT *
FROM Employees
WHERE 0=0

<CFIF #Form.FirstName# is not "">
AND Employees.FirstName LIKE '#form.FirstName#%'
</CFIF>

<CFIF #Form.LastName# is not "">
AND Employees.LastName LIKE '#form.LastName#%'
</CFIF>

<CFIF #Form.Salary# is not "">
AND Employees.Salary >= #form.Salary#
</CFIF>

<CFIF isDefined("Form.Contract") IS "YES">
AND Employees.Contract = 'Yes'
<CFELSE>
AND Employees.Contract = 'No'
</CFIF>

</CFQUERY>

<H3>Employee Data Based on Criteria from Form</H3>
```

```
<TABLE>
<TR>
    <TH>First Name</TH>
    <TH>Last Name</TH>
    <TH>Salary</TH>
    <TH>Contractor</TH>
</TR>
<CFOUTPUT QUERY="GetEmployees">
<TR>
    <TD>#FirstName#</TD>
    <TD>#LastName#</TD>
    <TD>#DollarFormat(Salary)#</TD>
    <TD>#Contract#</TD>
</TR>
</CFOUTPUT>
</TABLE>


</BODY>
</HTML>
```

3. Save the page as `getemp.cfm`.
4. Open the file `askemp.cfm` in your browser and enter criteria into any fields, then submit the form.
5. The results should meet the criteria you specify.

Code Review

The action page `getemp.cfm` build a SQL statement dynamically based on what the user enters in the form page AskEmp.cfm.

CFML Code	Description

`SELECT *`
`FROM Employees`
`WHERE 0=0`
	Get all the records from the Employees table as long as 0=0.
	The WHERE 0=0 clause has no impact on the query submitted to the database. But if none of the conditions is true, it ensures that the WHERE clause does not result in a SQL syntax error.

`<CFIF #Form.FirstName# is not "">`
`AND Employees.FirstName LIKE '#form.FirstName#%'`
`</CFIF>`
	If the user entered anything in the FirstName text box in the form, add "AND Employees.FirstName LIKE '[*what the user entered in the FirstName text box*]%'" to the SQL statement.

CFML Code	Description
<CFIF #Form.LastName# is not ""> AND Employees.LastName LIKE '#form.LastName#%' </CFIF>	If the user entered anything in the LastName text box in the form, add "AND Employees.LastName LIKE '[*what the user entered in the LastName text box*]%'" to the SQL statement.
<CFIF #Form.Salary# is not ""> AND Employees.Salary >= #form.Salary# </CFIF>	If the user entered anything in the Salary text box in the form, add "AND Employees.Salary >= *[what the user entered in the Salary text box]"* to the SQL statement.
<CFIF isDefined("Form.Contract") IS "YES"> AND Employees.Contract = 'Yes' <CFELSE> AND Employees.Contract = 'No' </CFIF>	If the user checked the Contractor check box, get data for the employees who are contractors, otherwise, get data for employees who are not contractors.

CHAPTER 6

Updating Your Data

This chapter describes how to insert, update, and delete data in a database with ColdFusion.

Contents

Inserting Data

Inserting data into a database is usually done with two application pages:

- An insert form
- An insert action page

You can create an insert form with CFFORM tags (see "Creating Forms with the CFFORM Tag" on page 124) or with standard HTML form tags. When the form is submitted, form variables are passed to a ColdFusion action page that performs an insert operation (and whatever else is called for) on the specified data source. The insert action page can contain either a CFINSERT tag or a CFQUERY tag with a SQL INSERT statement. The insert action page should also contain a message for the end user.

Creating an HTML Insert Form

To create an insert form:

1. Create a new application page in Studio.
2. Edit the page so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Insert Data Form</TITLE>
</HEAD>

<BODY>
<H2>Insert Data Form</H2>
<FORM ACTION="insertdata.cfm" METHOD="Post">
    Employee ID:
    <INPUT TYPE="text" NAME="Employee_ID" SIZE="4" MAXLENGTH="4"><BR>
    First Name:
    <INPUT TYPE="text" NAME="FirstName" SIZE="35" MAXLENGTH="50"><BR>
    Last Name:
    <INPUT TYPE="text" NAME="LastName" SIZE="10" MAXLENGTH="10"><BR>
    Department Number:
    <INPUT TYPE="text" NAME="Department_ID" SIZE="4"
MAXLENGTH="4"><BR>
    Start Date:
    <INPUT TYPE="text" NAME="StartDate" SIZE="16" MAXLENGTH="16"><BR>
    Salary:
    <INPUT TYPE="text" NAME="Salary" SIZE="10" MAXLENGTH="10"><BR>
    Contractor:
    <INPUT TYPE="checkbox" name="Contract" value="Yes"
checked>Yes<BR><BR>
    <INPUT TYPE="reset" NAME="ResetForm" VALUE="Clear Form">
    <INPUT TYPE="submit" NAME="SubmitForm" VALUE="Insert Data">
</FORM>
    </BODY>
```

```
</HTML>
```

3. Save the file as `insertform.cfm` in the `myapps` directory.
4. View `insertform.cfm` in a browser.

Data Entry Form Notes and Considerations

Creating data entry fields for an HTML form is very simple:

- You need only create the HTML form fields for each database field into which you want to insert data.
- The names of your form fields must be identical to the names of your database fields.
- You can use the full range of HTML input controls, including list boxes, radio buttons, checkboxes, and multi-line text boxes in your forms.
- ColdFusion uses the NAME attribute to map HTML form fields to the corresponding database fields and inserts the data entered by the user into the appropriate database fields.

Creating an Action Page to Insert Data

There are two ways to create an action page to insert data into a database.

The CFINSERT tag is the easiest way to handle simple inserts from either a CFFORM or an HTML form.

For more complex inserts from a form submittal you can use a SQL INSERT statement in a CFQUERY tag instead of a CFINSERT tag. The SQL INSERT statement is more flexible because you can insert information selectively or use functions within the statement.

To create an insert action page with CFINSERT:

1. Create a new application page in Studio.
2. Enter the following code:

```
<CFINSERT DATASOURCE="CompanyInfo" TABLENAME="Employees">
<HTML>
<HEAD>
    <TITLE>Input Form</TITLE>
</HEAD>
<BODY>
<H1>Employee Added</H1>
<CFOUTPUT>
You have added #Form.FirstName# #Form.LastName# to the Employees
database.
</CFOUTPUT>
</BODY>
</HTML>
```

3. Save the page. as `insertpage.cfm`.
4. View `insertform.cfm` in a browser, enter values, and click the Submit button.
5. The data is inserted into the Employees table and the message appears.

To create an insert page with CFQUERY:

1. Create a new application page in Studio.
2. Enter the following code:

```
<CFQUERY NAME="AddEmployee"
DATASOURCE="CompanyInfoB">
INSERT INTO Employees (Fi', '#Form.LastName#',
'#Form.Phone#')
</CFQUERY>

<HTML>
<HEADER>
    <TITLE>Insert Action Page</TITLE>
</HEADER>

<BODY>
<H1>Employee Added</H1>
<CFOUTPUT>
You have added #Form.FirstName# #Form.LastName# to the Employees
database.
</CFOUTPUT>
</BODY>
</HTML>
```

3. Save the page. as `insertpage.cfm`.
4. View `isertform.cfm` in a browser, enter values, and click the Submit button.
5. The data is inserted into the Employees table and the message appears.

Updating Data

Updating data in a database is usually done with two pages:

- An update form
- An update action page

You can create an update form with CFFORM tags or HTML form tags. The update form calls an update action page, which can contain either a CFUPDATE tag or a CFQUERY tag with a SQL UPDATE statement. The update action page should also contain a message for the end user that reports on the update completion.

Creating an Update Form

An update form is similar to an insert form, but there are two key differences:

- An update form contains a reference to the primary key of the record that is being updated.

  A primary key is a field or combination of fields in a database table that uniquely identifies each record in the table. For example, in a table of employee names and addresses, only the Employee_ID would be unique to each record.

- Because the purpose of an update form is to update existing data, the update form is usually populated with existing record data.

The easiest way to designate the primary key in an update form is to include a hidden input field with the value of the primary key for the record you want to update. The hidden field indicates to ColdFusion which record to update.

To create an update form:

1. Create a new page in Studio.
2. Edit the page so that it appears as follows:

```
<CFQUERY NAME="GetRecordtoUpdate"
    DATASOURCE="CompanyInfo">
    SELECT *
        FROM Employees
        WHERE Employee_ID = #URL.Employee_ID#
</CFQUERY>

<HTML>
<HEAD>
    <TITLE>Update Form</TITLE>
</HEAD>
<BODY>

<CFOUTPUT QUERY="GetRecordtoUpdate">
<FORM ACTION="UpdatePage.cfm" METHOD="Post">
<INPUT TYPE="Hidden" NAME="Employee_ID"
    VALUE="#Employee_ID#"><BR>
    First Name:
    <INPUT TYPE="text" NAME="FirstName" VALUE="#FirstName#"><BR>
    Last Name:
    <INPUT TYPE="text" NAME="LastName" VALUE="#LastName#"><BR>
    Department Number:
    <INPUT TYPE="text" NAME="Department_ID"
VALUE="#Department_ID#"><BR>
    Start Date:
    <INPUT TYPE="text" NAME="StartDate" VALUE="#StartDate#"><BR>
    Salary:
    <INPUT TYPE="text" NAME="Salary" VALUE="#Salary#"><BR>
    Contractor:
```

```
    <INPUT TYPE="Submit" VALUE="Update Information">
</FORM>
</CFOUTPUT>

</BODY>
</HTML>
```

3. Save the page. as `updatedorm.cfm`.
4. View `updateform.cfm` in a browser.

Code Review

Code	Description
`<CFQUERY NAME="GetRecordtoUpdate" DATASOURCE="CompanyInfo"> SELECT * FROM Employees WHERE Employee_ID = #URL.Employee_ID# </CFQUERY>`	Query the CompanyInfo datasource and return the records in which the employee ID matches what was entered in the URL.
`<CFOUTPUT QUERY="GetRecordtoUpdate">`	Display the results of the GetRecordtoUpdate query.
`<FORM ACTION="EmployeeUpdate.cfm" METHOD="Post">`	Create a form whose variables will be process on the EmployeeUpdate.cfm action page.
`<INPUT TYPE="Hidden" NAME="Employee_ID" VALUE="#Employee_ID#"> `	Use a hidden input field to pass the employee ID to the action page.
`First Name: <INPUT TYPE="text" NAME="FirstName" VALUE="#FirstName#"> Last Name: <INPUT TYPE="text" NAME="LastName" VALUE="#LastName#"> Department Number: <INPUT TYPE="text" NAME="Department_ID" VALUE="#Department_ID#"> Start Date: <INPUT TYPE="text" NAME="StartDate" VALUE="#StartDate#"> Salary: <INPUT TYPE="text" NAME="Salary" VALUE="#Salary#"> Contractor: <INPUT TYPE="checkbox" name="Contract" value="Yes" checked>Yes <INPUT TYPE="Submit" VALUE="Update Information"> </FORM> </CFOUTPUT>`	Populate the fields of the update form.

Creating an Action Page to Update Data

You can create an action page to update data with either the CFUPDATE tag or CFQUERY with the UPDATE statement.

The CFUPDATE tag is the easiest way to handle simple updates from a front end form. The CFUPDATE tag has an almost identical syntax to the CFINSERT tag.

To use CFUPDATE, you must include all of the fields that make up the primary key in your form submittal. The CFUPDATE tag automatically detects the primary key fields in the table you are updating and looks for them in the submitted form fields. ColdFusion uses the primary key field(s) to select the record to update. It then updates the appropriate fields in the record using the remaining form fields submitted.

For more complicated updates, you can use a SQL UPDATE statement in a CFQUERY tag instead of a CFUPDATE tag. The SQL update statement is more flexible for complicated updates.

To create an update page with CFUPDATE:

1. Create a new application page in Studio.
2. Enter the following code:

```
<CFUPDATE DATASOURCE="CompanyInfo"
    TABLENAME="Employees">

<HTML>
<HEAD>
    <TITLE>Update Employee</TITLE>
</HEAD>
<BODY>

<H1>Employee Added</H1>
<CFOUTPUT>
You have updated the information for #Form.FirstName# #Form.LastName#
in the Employees database.
</CFOUTPUT>

</BODY>
</HTML>
```

3. Save the page. as `updatepage.cfm`.
4. View `updateform.cfm` in a browser, enter values, and click the Submit button.
5. The data is updated in the Employees table and the message appears.

To create an update page with CFQUERY:

1. Create a new application page in Studio.
2. Enter the following code:

```
<CFQUERY NAME="UpdateEmployee"
DATASOURCE="CompanyInfo">
```

```
    UPDATE Employees
    SET Firstname='#Form.Firstname#',
    LastName='#Form.LastName#',
    Department_ID='#Form.Department_ID#'
    StartDate='#Form.StartDate#'>
    Salary=#Form.Salary#>
        WHERE Employee_ID=#Employee_ID#
    </CFQUERY>

    <H1>Employee Added</H1>
    <CFOUTPUT>
    You have updated the information for #Form.FirstName# #Form.LastName#
    in the Employees database.
    </CFOUTPUT>
```

3. Save the page. as `updatepage.cfm`.
4. View `updateform.cfm` in a browser, enter values, and click the Submit button.
5. The data is updated into the Employees table and the message appears.

Code Review

Code	Description

`<CFQUERY NAME="UpdateEmployee"`
`DATASOURCE="CompanyInfo">`
`UPDATE Employees`
`SET Firstname='#Form.Firstname#',`
`LastName='#Form.LastName#',`
 	
`Department_ID='#Form.Department_ID#'`
`StartDate='#Form.StartDate#'>`
`Salary=#Form.Salary#>`
`WHERE Employee_ID=#Employee_ID#`
`</CFQUERY>`
	After the SET clause, you must name a table column. Then, you indicate a constant or expression as the value for the column.
	Be sure to remember the WHERE statement. If you do not use it, If you do not use it, the SQL UPDATE statement will apply the new information to every row in the database.

Deleting Data

Deleting data in a database can be done with a single delete page. The delete page contains a CFQUERY tag with a SQL delete statement.

To delete one record from a database:

1. Open the file `updateform.cfm` in Studio.
2. Modify the file by changing the FORM tag so that it appears as follows:
   `<FORM ACTION="deletepage.cfm" METHOD="Post">`
3. Save the modified file as `deleteform.cfm`.
4. Create a new application page in Studio.

5. Enter the following code:

```
<CFQUERY NAME="DeleteEmployee"
    DATASOURCE="CompanyInfo">
    DELETE FROM Employees
    WHERE Employee_ID = #URL.EmployeeID#
</CFQUERY>

<HTML>
<HEAD>
    <TITLE>Delete Employee Record</TITLE>
</HEAD>
<BODY>
<H3>The employee record has been deleted.</H3>

</BODY>
</HTML>
```

6. Save the page. as `deletepage.cfm`.
7. View `deleteform.cfm` in a browser, enter values, and click the Submit button.

The employee is deleted from the Employees table and the message appears.

To delete several records, you would specify a condition. The following example demonstrates deleting the records for everyone in the Sales department from the Employee table. The example assumes that there are several Employees in the sales department.

```
DELETE FROM Employees
WHERE Department = 'Sales'
```

To delete all the records from the Employees table, you would use the following:

```
DELETE FROM Employees
```

Note Deleting records from a database is *not* reversible. Use delete statements carefully.

Requiring Users to Enter Values in Form Fields

One of the limitations of HTML forms is the inability to define input fields as required. Because this is a particularly important requirement for database applications, ColdFusion provides a server-side mechanism for requiring users to enter data in fields.

To define an input field as required, use a hidden field that has a NAME attribute composed of the field name and the suffix "_required." For example, to require that the user enter a value in the FirstName field, use the syntax:

```
<INPUT TYPE="hidden" NAME="FirstName_required">
```

If the user leaves the FirstName field empty, ColdFusion rejects the form submittal and returns a message informing the user that the field is required. You can customize the contents of this error message using the VALUE attribute of the hidden field. For

example, if you want the error message to read "You must enter your first name," use the syntax:

```
<INPUT TYPE="hidden"
    NAME="FirstName_required"
    VALUE="You must enter your first name.">
```

Validating the Data That Users Enter in Form Fields

Another limitation of HTML forms is that you cannot validate that users input the type or range of data you expect. ColdFusion enables you to do several types of data validation by adding hidden fields to forms. The hidden field suffixes you can use to do validation are as follows:

Form Field Validation Using Hidden Fields		
Field Suffix	**Value Attribute**	**Description**
`_integer`	Custom error message	Verifies that the user enters a number. If the user enters a floating point value, it is rounded to an integer.
`_float`	Custom error message	Verifies that the user enters a number. Does not do any rounding of floating point values.
`_range`	MIN=MinValue MAX=MaxValue	Verifies that the numeric value entered is within the specified boundaries. You can specify one or both of the boundaries separated by a space.
`_date`	Custom error message	Verifies that a date has been entered and converts the date into the proper ODBC date format. Will accept most common date forms, for example, 9/1/98; Sept. 9, 1998).
`_time`	Custom error message	Verifies that a time has been correctly entered and converts the time to the proper ODBC time format.
`_eurodate`	Custom error message	Verifies that a date has been entered in a standard European date format and converts into the proper ODBC date format.

Note Adding a validation rule to a field does not make it a required field. You need to add a separate `_required` hidden field if you want to ensure user entry.

To validate the data users enter in the Insert Form

1. Open the file `insertform.cfm` in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Insert Data Form</TITLE>
</HEAD>

<BODY>
<H2>Insert Data Form</H2>
<FORM ACTION="insertdata.cfm" METHOD="Post">
    <INPUT TYPE="hidden"
        NAME="DeptID_integer"
        VALUE="The department ID must be a number.">
    <INPUT TYPE="hidden"
        NAME="StartDate_date"
        VALUE="Enter a valid date as the start date.">
    <INPUT TYPE="hidden"
        NAME="Salary_float"
        VALUE="The salary must be a number.">
    Employee ID:
    <INPUT TYPE="text"
        NAME="Employee_ID"
        SIZE="4"
        MAXLENGTH="4"><BR>
    First Name:
    <INPUT TYPE="text"
        NAME="FirstName"
        SIZE="35"
        MAXLENGTH="50"><BR>
    Last Name:
    <INPUT TYPE="text"
        NAME="LastName"
        SIZE="10"
        MAXLENGTH="10"><BR>
    Department Number:
    <INPUT TYPE="text"
        NAME="Department_ID" SIZE="4"
        MAXLENGTH="4"><BR>
    Start Date:
    <INPUT TYPE="text"
        NAME="StartDate" SIZE="16"
        MAXLENGTH="16"><BR>
    Salary:
    <INPUT TYPE="text"
        NAME="Salary"
        SIZE="10"
        MAXLENGTH="10"><BR>
    Contractor:
    <INPUT TYPE="checkbox"
        NAME="Contract"
        VALUE="Yes" CHECKED>Yes<BR><BR>
```

```
    <INPUT TYPE="reset"
        NAME="ResetForm"
        VALUE="Clear Form">
    <INPUT TYPE="submit"
        NAME="SubmitForm"
        VALUE="Insert Data">
</FORM>
    </HTML>
```

3. Save the file.

The VALUE attribute is optional. A default message displays if no value is supplied.

When the form is submitted, ColdFusion scans the form fields to find any validation rules you specified. The rules are then used to analyze the user's input. If any of the input rules are violated, ColdFusion sends an error message to the user that explains the problem. The user then must go back to the form, correct the problem and resubmit the form. ColdFusion will not accept form submission until the entire form is entered correctly.

Because numeric values often contain commas and dollar signs, these characters are automatically stripped out of fields with `_integer`, `_float`, or `_range` rules before they are validated and saved to the database.

Note If you use CFINSERT or CFUPDATE and you specified columns in your database that are numeric, date, or time, form fields that insert data into these fields are automatically validated. You can use the hidden field validation functions for these fields to display a custom error message.

CHAPTER 7

Reusing Code

This chapter describes how to reuse common code with CFINCLUDE, and create custom CFML tags that encapsulate common code.

Contents

Ways to Reuse Code

ColdFusion provides several different ways to reuse code. If you are using ColdFusion Studio, you can write code snippets, which you can copy into templates. For more information on writing code snippets, see *Using ColdFusion Studio*. You can include a template within another template with the CFINCLUDE tag. In addition, you can create custom tags in CFML. Unlike included templates, these custom tags act as other tags do, allowing you to pass parameters to them. Included templates, on the other hand, behave just as though you typed the included code directly into the calling page.

Reusing Common Code with CFINCLUDE

Often times, you'll use some of the same elements in multiple pages; for example, navigation, headers, and footer code.

Instead of copying and maintaining the same code from page to page, ColdFusion allows you to store the code in one page and then refer to it in many pages. This way, you can modify one file; the changes appear throughout an entire application.

Use the CFINCLUDE tag to automatically include an existing file in the current page. The file to include is the *template*. The page that calls the template is also known as the calling page. Each time the calling page is requested, the template's file contents are included in that page for processing.

Refer to the *CFML Language Reference* for CFINCLUDE syntax.

To reference code in a calling page:

1. Open the file `askemp.cfm` in Studio.
2. Include `logo.cfm` in this page:

   ```
   <CFINCLUDE TEMPLATE="logo.cfm">
   ```
3. Save the page.
4. Open `getemp.cfm` in Studio.
5. Include `logo.cfm` file in this page:

   ```
   <CFINCLUDE TEMPLATE="Logo.cfm">
   ```
6. View `askemp.cfm` in a browser, then submit the form so that you display `getemp.cfm`.

   The logo should appear on both pages.

Note The file `logo.cfm` must be in the same directory where you saved `askemp.cfm` and `getemp.cfm`. If it isn't, make sure it is in a directory that has a mapping defined in ColdFusion Administrator, or move it to the appropriate directory.

About Custom Tags in CFML

Custom tags wrap functionality in a page that can be called from a ColdFusion application page. ColdFusion custom tags built in CFML allow for rapid application development and code re-use while offering off-the-shelf solutions to many programming chores.

An online RealVideo title called "Creating 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.

Using Existing Custom Tags

Before creating a custom tag in CFML, you will probably want to visit the Custom Tag section of the Allaire Developer Exchange at . 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 tags, see Chapter 18, "Building Custom CFAPI Tags," on page 275.

If you don't find a tag that meets your specific needs, you want to create your own custom tags in CFML.

Writing Custom CFML Tags

Writing a custom tag in CFML is no different from writing any CFML template. You can use all CFML constructs, as well as HTML.

Custom tags are stored either in the current directory or under the `customtags` directory. You call them using 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_getweather invokes the file `getweather.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.

Note While tag names in templates are case-insensitive, custom tag file names must be lower case on UNIX.

Defining attributes

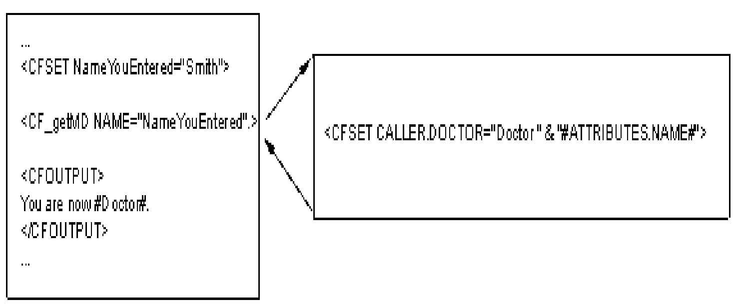
CFML custom tags support both required and optional attributes. Attributes are defined as name-value pairs. Custom tag attributes conform to CFML coding standards:

- ColdFusion passes any attributes in the ATTRIBUTES scope.
- Use the CFPARAM tag at the top of a custom tag to test for and assign defaults for each attribute that may be passed from a calling template.
- Use the ATTRIBUTES.*attribute_name* syntax when initializing passed attributes to distinguish them from local ones.
- 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.

Passing Attribute Values between Custom Tags

Because custom tags are individual templates, variables and other data aren't automatically shared between a custom tag and the calling template. To pass data, you define attributes for the custom tag, just as in standard CFML coding.

To pass data from the calling template to the custom tag, use the ATTRIBUTES scope. Conversely, to pass values back to the calling template, use the CALLER scope. You can also access variables already set on the calling page in the custom tag by simply prefixing the variable with the 'CALLER.' prefix.



To create a custom tag:

1. Create a new application page (the calling page) in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Enter Name</TITLE>
</HEAD>

<BODY>
<!--- Enter a name, which could also be done in a form --->
<!--- This example simply uses a cfset --->
<CFSET NameYouEntered="Smith">

<!--- display the current name --->
<CFOUTPUT>
Before you leave this page, you're #NameYouEntered#.<BR>
</CFOUTPUT>

<!--- go to the custom tag --->
<CF_GETMD NAME="#NameYouEntered#">
<!--- come back from custom tag --->


<!--- display the results of the custom tag --->
<CFOUTPUT>
You are now #DOCTOR#.
</CFOUTPUT>

</BODY>
</HTML>
```

3. Save the page as `callingpage.cfm`.
4. Create another new page (the custom tag) in Studio.
5. Enter the following code:

```
<HTML>
<HEAD>
    <TITLE>GetMD Custom Tag</TITLE>
</HEAD>

<BODY>
<!--- get the value of the varible NAME from the calling page --->
<!--- put the text "Doctor " in front of the name --->
<!--- create a variable called DOCTOR, make its value "Doctor NAME" -
-->
<!--- and make its scope CALLER so that you can pass it back to the
calling page --->

<CFPARAM VALUE="Attributes.Name" DEFAULT="Who"

<CFSET CALLER.DOCTOR="Doctor " & "#ATTRIBUTES.NAME#">

</BODY>
</HTML>
```

6. Save the page as `getmd.cfm`.

7. Open the file `callingpage.cfm` in your browser.

The calling page uses the `getmd` custom tag and displays the results.

Code Review

Code	Description
`<CFSET NameYouEntered="Smith">`	In the calling page, create a variable NameYouEntered and assign it the value "Smith."
`<CF_GETMD NAME="#NameYouEntered#">`	In the calling page, call the getMD custom tag and pass it the NAME parameter whose value is the value of the variable NameYou Entered.
`<CFPARAM VALUE="Attributes.Name" DEFAULT="Who"`	Assign the value "Who" to Name if it has no value.
`<CFSET CALLER.DOCTOR="Doctor " & "#ATTRIBUTES.NAME#">`	See below. (It is helpful to look at this code from right to left.)
`#ATTRIBUTES.NAME#`	Get the value of the variable NAME from the calling page
`<CFSET DOCTOR="Doctor " & "#ATTRIBUTES.NAME#">`	Create a variable called DOCTOR, make its value "Doctor NAME"
`<CFSET CALLER.DOCTOR="Doctor " & "#ATTRIBUTES.NAME#">`	Make the variable's scope CALLER so that you can pass it back to the calling page

Tip Be careful not to overwrite variables that might already exist on the calling page. You should adopt a naming convention to minimize the chance of overwriting variables. For example, prefix the returned variable with *customtagname_*, with customtagname being the name of the custom tag.

Note Data pertaining to the HTTP request or to the current application is visible. This includes the variables in FORM, URL, CGI, COOKIES, SERVER, APPLICATION, SESSION, and CLIENT scopes.

The Request scope is a reserved variable/scope that allows you to store data pertaining to the complete hierarchy of custom tags used in a single page request. It is a structure named "request." The variable is available to all templates: base, includes, and custom tags. Collaborating custom tags that are not nested in a single tag can exchange data via the request structure. You should assign a unique name for each variable. You should store data in structures nested inside the request scope.

Nesting Custom Tags

ColdFusion lets you turn a custom tag into a special kind of container that can enclose additional custom tags, which allows you to nest tags. Base tags are also know as ancestors or parent tags, while the tags that base tags call are known as sub tags, descendants, or child tags.

Terms to Describe the Relationship Between Nested Tags		
Calling tag	**Tag that is nested within the calling tag**	**Notes**
base tag	sub tag	A base tag is an ancestor that has been explicitly associated with a descendant with CFASSOCIATE.
ancestor	descendant	An ancestor is any tag that contains other tags between its start and end tags.
parent	child	"Parent" and "child" are synonyms for "ancestor" and "descendant."

You can create multiple levels of nesting by closing a sub tag. In this case, the sub tag becomes the base tag for its own sub tags. Any tag with an end tag present can be an ancestor to another tag.

Nested custom tags operate through three modes of processing, which are exposed to the base tags through the variable ThisTag.ExecutionMode:

- The start mode, in which the base tag is processed for the first time.
- The inactive mode, in which sub tags and other code contained within the base tag are processed.

  No processing occurs in the base tag during this phase.
- The end mode, in which the base tag is processed a second time.

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.

Passing Data Between Nested Custom Tags

A key custom tag feature 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 custom tags can expose all their data to collaborating tags.

When you develop custom tags, you should document all variables that collaborating tags can access and/or modify. When your custom tags collaborate with other custom tags, you should make sure that they do not modify any undocumented data.

To preserve encapsulation, put all tag data access and modification operations into custom tags. For example, rather than simply documenting that the variable MyQueryResults in a tag's implementation holds an important query result set and expecting users of the custom tag to manipulate MyQueryResults directly, create another nested custom tag that manipulates MyQueryResult. 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 page. Ancestor and descendant relationships are important because they relate to the order of tag nesting.

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

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.

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>
```

Ancestor data access

The ancestor's data is represented by a structure object that contains all the ancestor's data.

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're 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>
```

Passing Custom Tag Arguments via CFML Structures

Attributes can be passed to custom tags via the reserved attribute ATTRIBUTECOLLECTION. ATTRIBUTECOLLECTION must reference a structure.

Syntax

CFMODULE

```
<CFMODULE TEMPLATE=template
    OTHERATTR1=value
    ATTRIBUTECOLLECTION=structure
    OTHERATTR2=value>
```

Shorthand

```
<CF_MYCUSTOMTAG OTHERATTR1=value
    ATTRIBUTECOLLECTION=structure
    OTHERATTR2=value>
```

The key/value pairs contained within the structure specified by ATTRIBUTECOLLECTION will be copied into the ATTRIBUTES scope. This has essentially the same effect as specifying these attributes in the custom tag's attribute list.

ATTRIBUTECOLLECTION may be freely mixed with other attributes within the custom tag's attribute list.

The reserved attribute name ATTRIBUTECOLLECTION

Custom tag processing reserves ATTRIBUTECOLLECTION to refer to the structure holding a collection of custom tag attributes. If ATTRIBUTECOLLECTION does not refer to such a collection, the custom tag processor will raise a TEMPLATE exception.

A custom tag invoked by the two examples above may refer to #attributes.x# and #attributes.y# to access the attributes passed via structure.

If the called custom tag uses a CFASSOCIATE tag to save its attributes in the base tag, the attributes passed via structure will be saved as independent attribute values, with no indication that they were aggregated into a structure by the custom tag's caller.

Examples

Via CFMODULE

```
<CFSET zort=StructNew()>
<CFSET zort.X = "-X-">
<CFSET zort.Y = "-Y-">
<CFMODULE TEMPLATE="testtwo.cfm"
    a="blab"
    attributecollection=#zort#
    foo="16">
```

Via shorthand

```
<CFSET zort=StructNew()>
<CFSET zort.X = "-X-">
<CFSET zort.Y = "-Y-">
<CF_TESTTWO a="blab" attributecollection=#zort# foo="16">
```

Accessing attributes within the custom tag

If `testtwo.cfm` contains this CFML:

```
---custom tag ---<br>
<CFOUTPUT>#attributes.a# #attributes.x# #attributes.y#
    #attributes.foo#</cfoutput>
    <BR>--- end custom tag ---
```

Its output will be:

```
---custom tag ---
blab -X- 12 16
--- end custom tag ---
```

Executing Custom Tags

Tag instance data

During the execution of a custom tag template, ColdFusion keeps some 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 ThisTag.HasEndTag.
- 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 is executed for both the start and end tag of a custom tag.

Modes of execution

ColdFusion invokes a custom tag template 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, 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>
```

Terminating tag execution

CFEXIT terminates execution of a custom tag. CFEXIT's METHOD attribute specifies where execution continues. 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
	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>
```

Installing Custom Tags

Custom tags are just like other .cfm files *except* that they must be installed in a specific location to be accessible from the calling template. Because ColdFusion loads the first instance it finds of the custom tag called by a template, you should avoid placing copies of a custom tag in different locations.

Local tags

The ColdFusion engine first searches for a custom tag in 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, for example `C:\CFUSION\CustomTags`. 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.

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 additional ColdFusion capabilities:

- An advanced invocation syntax to resolve possible name conflicts
- Advanced security
- Template encoding

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 if the NAME attribute is not used. Specifies a relative path to the cfm file. Same as TEMPLATE attribute in CFINCLUDE. Note that the directory must have a mapping defined in ColdFusion Administrator Example: <CFMODULE TEMPLATE="../MyTag.cfm"> identifies a custom tag file in the parent directory.
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 avoid name conflicts, you can register custom tags as a security resource on the ColdFusion Administrator Advanced Security page. See *Administering ColdFusion Server*for details.

Encoding Custom Tags

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

`cfencode` uses the following syntax:

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

The following options are supported:

cfencode Command Line Options	
Option	**Description**
input file	Name of the file you want to encode. cfencode will not process an encoded file.
output file	Path and filename of the output file. **Warning**: If you don't specify an output file name, a warning message asks if you want to continue, in which case the encoded file will overwrite the source file.
/r	Recursive, when used with wildcards, recurses through subdirectories to encode files.
/q	Suppresses warning messages.
/h	Header, allows custom header to be written to the top of the encoded file(s).
/v	Required parameter that allows encoding 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.

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

CHAPTER 8

Debugging and Error Handling

ColdFusion includes sophisticated debugging and code validation tools. This chapter gives an overview of the debugging options available in the ColdFusion Administrator and how to enable CFML attribute validation.

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

ColdFusion Studio provides interfaces for debugging application pages and for dynamically validating multiple levels of HTML and CFML code. For information on using these features, see *Using ColdFusion Studio*.

Contents

Debug Settings in the ColdFusion Administrator

ColdFusion can provide important debugging information for every application page requested by a browser. When enabled, debugging output is shown in a block following normal page output.

For detailed information on the debugging and logging settings in the ColdFusion Administrator, see *Administering ColdFusion Server* .

Note By default, when you enable any of these options, debug output becomes visible to all users. You can, however, restrict debug output by using the Restrict debug output to selected IP address form at the bottom of the Debug Settings page.

Generating debug information for an individual page

You can view the parameters and CGI environment variables for an individual application page without turning on the global debug settings in the ColdFusion Administrator. Simply append the parameter "mode=debug" to the end of the URL.

```
www.myserver.com/cfdocs/test.cfm?mode=debug
```

Generating debug information for an individual query

You can view debug information for an individual query by putting the DEBUG attribute into the opening CFQUERY tag:

```
<CFQUERY NAME="TestQuery" DATASOURCE="CompanyInfo" DEBUG>
    SELECT * FROM TestTable
</CFQUERY>
```

When this query runs, it places the debug information into the output page where the query is placed.

Error messages

If ColdFusion is unable to fulfill a request because of an error, it returns a diagnostic message to the user. The message includes a link that allows the user to email a report of the error to the site administrator. You enable this feature in the Mail Logging page of the ColdFusion Administrator. Errors are written to a log file for later review.

ColdFusion returns:

- Database errors, including the ODBC error code, the extended error message returned fromt the ODBC driver, the name of the data source, and the SQL statement submitted to the database.
- Syntax error, including the line of the application page file on which the error occurred.
- System-related errors, such as out of memory conditions, or file or disk access errors.

Tip If you get a message that does not explicitly identify the cause of the error, check on key system parameters like available memory and disk space.

For information on using the Logging settings and Mail Logging settings, see *Administering ColdFusion Server*.

CFML Code Validation

The ColdFusion Application Server features two modes of attribute checking for processing application pages: strict and relaxed. Allaire recommends that you always use the strictest possible level of CFML validation. To enable strict validation, open the ColdFusion Administrator Server Settings page and check the "Enable Strict Attribute Validation" box.

The code validator inspects all code before execution begins. In addition, attribute validation is generally performed at p-code time and not at execution. The exceptions to this rule are tags with a "switch" attribute, such as ACTION= or METHOD=, for which the value is provided at runtime. These instances are validated at runtime. There are two implications:

- There will be a slight performance penalty due to runtime attribute validation.
- The CFML syntax checker will not be able to detect an invalid attribute combination — in this case because it does not execute the CFML page it checks.

Although dynamically providing an action can save a few lines of code, you should avoid this practice in the interest of a more complete validation and faster application performance.

Tip If a commercially purchased custom tags fails to run, try turning off the "Enforce Strict Attribute Validation" setting in the ColdFusion Administrator. If the tag continues to generate errors, you should contact the tag's vendor.

The CFML Syntax checker application page is:

webroot`/cfdocs/cfmlsyntaxcheck.cfm.`

Troubleshooting Common Problems

The following section describes a few common problems that you may encounter and ways to resolve them.

ODBC data source configuration

Problem: ODBC driver manager cannot make a connection to the database.

Connection errors may include problems with the location of files, network connections, and database client library configuration.

First, verify that you can connect to the database by clicking the Verify button on the ODBC Data Sources page of the ColdFusion Administrator. If you are unable to make a simple connection from that page, you need to work with your database and/or driver vendor to solve the problem.

Problem: Data source does not exist or name is incorrectly specified.

Create data sources before you refer to them in your application source files. Also, check the spelling of the data source name.

HTTP/URL

Problem: ColdFusion cannot correctly decode the contents of your form submission.

The METHOD in forms sent to the ColdFusion server must be Post, for example:

```
<FORM ACTION="test.cfm" METHOD="Post">
```

Problem: The browser complains when you include spaces in URLs.

URLs cannot have embedded spaces. Use a plus sign (+) wherever you want to include a space. ColdFusion correctly translates the + sign into a space.

A common scenario in which this error occurs is when you dynamically generate your URL from database text fields that may have embedded spaces. To avoid this problem, include only numeric values in the dynamically generated portion of URLs.

Or, you can use the URLEncodedFormat function, which automatically replaces spaces with + signs.

CFML syntax errors

Problem: You get an error message you don't understand.

Make sure all your CFML tags have matching end tags where appropriate. It is a common error to omit the end tag for the CFQUERY, CFOUTPUT, CFTABLE, or CFIF tag.

When developing pages in ColdFusion Studio, use the Tag Completion feature, which adds an editing tag each time you create an opening tag.

Problem: Invalid attribute or value.

If you use an invalid attribute or attribute values, ColdFusion returns an error message. To prevent such syntax errors, use the ColdFusion syntax validation tools in ColdFusion Studio.

Problem: Mismatched quotes and escape characters.

Check strings in attributes and expressions for proper placement of single and double quotes. Color coding in ColdFusion Studio can help you spot improper quote placement.

Generating Custom Error Messages (CFERROR)

ColdFusion displays error pages that can help you to debug your application. There are four types of errors in ColdFusion:

- **REQUEST** — Request errors occur when a application page is requested and there is an error in the page's code.
- **VALIDATION** — Validation errors occur when a user violates the form field validation rules during a form submittal.
- **EXCEPTION** — Exception errors handle exceptions.
- **MONITOR** — Sets up an exception monitor.

By default, ColdFusion returns a standard page for these errors. But you may want to customize the error pages that are returned, to make them consistent with the look and feel of your application. Custom error pages also allow you to control the error information that users see, as well as offering work-arounds or ways for users to report the errors.

You set the custom error application pages with the CFERROR tag. You can set the custom error application pages page-by-page, but because custom error pages generally apply to an entire application, it is more efficient to include the CFERROR tag in the `Application.cfm` file. After you create a custom error page, you must include the CFERROR tag in your application's `Application.cfm` page. See "Understanding the Web Application Framework" on page 184 for more information.

For information on the syntax of the CFERROR tag, see the *CFML Language Reference*.

Creating an error application page

The error application page is a file that includes HTML and the parameters associated with the error. The error application page cannot use any CFML tags.

The parameters associated with an error depend on the type of error. All the error parameters use the Error prefix (for example, `Error.Diagnostics`).

See the *CFML Language Reference* for more information on the error variables and on using the CFERROR tag.

The following examples show the two types of custom error pages.

Example of a request error

The following example shows a custom error page for a request error:

```
<HTML>
<HEAD>
    <TITLE>Products - Error</TITLE>
</HEAD>
<BODY>

<CFOUTPUT>
<H2>Sorry</H2>
```

```
<P>An error occurred when you requested this page.
Please email the Webmaster to report this error.
We will work to correct the problem and apologize
for the inconvenience.</P>

<TABLE BORDER=1>
<TR><TD><B>Error Information</B> <BR>
    #Error.DateTime# <BR>
    #Error.Template# <BR>
    #Error.RemoteAddress# <BR>
    #Error.HTTPReferer#
</TD></TR></TABLE>

</CFOUTPUT>
</BODY>
</HTML>
```

Example of a validation error

The following example shows a custom error page for a validation error.

```
<HTML>
<HEAD>
    <TITLE>Products - Error</TITLE>
</HEAD>
<BODY>

<H2>Oops</H2>

<P>You failed to complete all the fields
in the form. The following problems occurred:</P>

#Error.InvalidFields#

</BODY>
</HTML>
```

Overview of Exception Handling in ColdFusion

Ordinarily, when ColdFusion encounters an error, it stops processing. However, you can use ColdFusion's exception handling tags 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.

In order for your code to handle an exception, the tags in question must appear within a CFTRY block. It's a good idea to enclose an entire application page in a CFTRY block, and use a CFCATCH blocks to trap potential errors. When an exception occurs within the CFTRY block, processing is 'thrown' to the CFCATCH block.

Note For cases when the error handler is not able to successfully handle the thrown error, use the CFRETHROW tag within a <CFCATCH> block.

```
<CFTRY>
... Add code here ...
    <CFCATCH TYPE="exception type1">
    ... Add exception processing code here ...
    </CFCATCH>
    <CFCATCH TYPE="exception type2">
    ... Add exception processing code here ...
    </CFCATCH>
    <CFCATCH TYPE="Any">
    ... Add exception processing code here ...
    </CFCATCH>
</CFTRY>
```

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.

Note Do not attempt to enclose an entire application in a CFTRY block by putting the CFTRY tag in Application.cfm because you can't be sure that there will be a matching CFTRY end tag.

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

Types of recoverable exceptions supported

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

Types of recoverable exceptions		
Type	**Tag(s)**	**Notes**
Application-defined exception events	CFTHROW CFCATCH TYPE="Application" CFCATCH TYPE="Any" a CFCATCH block that has no TYPE attribute	Raise exceptions using the CFTHROW tag (with an optional diagnostic message), then catch using CFCATCH. If you specify the type to be "APPLICATION," the CFCATCH tag catches only those custom exceptions that have been specified as having the APPLICATION type in the CFTHROW tag that defines them.
Database failures	CFCATCH TYPE="Database" CFCATCH TYPE="Any"	Catch failed database operations, such as failed SQL statements, ODBC problems, and so on.
Template errors	CFCATCH TYPE="Template" CFCATCH TYPE="Any"	Catch general application page errors. The tags that throw an exception of TYPE="TEMPLATE" are CFINCLUDE, CFMODULE, and CFERROR.
Missing included file errors	CFCATCH TYPE="MissingInclude" CFCATCH TYPE="Any"	Catch errors for missing included files.
Object exceptions	CFCATCH TYPE="Object"	Catch exceptions in ColdFusion code that works with objects.
Security exceptions	CFCATCH TYPE="Security"	Raise catchable exceptions in ColdFusion code that works with security.
Expression exceptions	CFCATCH TYPE="Expression"	Catch exceptions when an expression fails evaluation.
Locking exceptions	CFCATCH tag with TYPE="Lock"	Catch failed locking operations, such as when a CFLOCK critical section times out or fails at runtime.
Custom exceptions	CFCATCH TYPE="*your_custom_exception_type*"	Specify a custom type as well as one of the predefined types.
Unexpected internal exceptions	CFCATCH TYPE="Any"	Catch unexpected exceptions

Specifying the type as ANY causes the ColdFusion Application Server to catch internal exceptions, memory allocation errors, and access violations, which you may not be prepared to handle.

Applications can optionally use the CFTHROW tag to raise custom exceptions. Such exceptions are caught with any of the following type specifications:

- `TYPE="custom_exception_type"`
- `TYPE="APPLICATION"`
- `TYPE="ANY"`

The *custom_exception_type* type designates the name of a user-defined type specified with the CFTHROW tag.

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

Exception Information in CFCATCH

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

Exception Property Variables	
Property variable	**Description**
`CFCATCH.TYPE`	The exception's type, returned as a string:
`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.
`CFCATCH.EXTENDEDINFO`	A custom error message. This is returned only for CFCATCH tags where `TYPE="APPLICATION"` or a custom type.

Exception Property Variables (Continued)	
Property variable	**Description**
CFCATCH.ERRORCODE	Any exception that is a part of the CFML exception hierarchy supplies a value for this variable. For TYPE="Application" CFTHROW tags may supply a value for this code via the ERRORCODE attribute. For Type="Database"CFCATCH.ERRORCODE has the same value as CFCATCH.SQLSTATE. Otherwise, the value of CFCATCH.ERRORCODE is the empty string.
CFCATCH.TAGCONTEXT	Provides the name and position of each tag in the tag stack and the full path names of the files that contain the tags in the tag stack.

Tag context information

The ColdFusion Administrator's debugging settings page allows you to"Enable CFML stack trace." When this setting is enabled, CFCATCH blocks make available an array of structures called CFCATCH.TagContext. Each structure represents one level of the ColdFusion runtime's active tag context at the time when the ColdFusion interpreter detected the exception.

The structure at position 1 of the array represents the outermost tag in the stack of tags that were executing when the interpreter detected the exception. The structure at position ArrayLen(CFCATCH.TAGCONTEXT) represents the currently executing tag at the time the interpreter detected the exception.

The TagContext structures have the following attributes:

TEMPLATE — The pathname of the application page that contains the tag.

LINE and COLUMN — The tag's line number and column number within the application page.

Note Turn off "Enable CFML stack trace" to avoid having production servers expend resources creating a traceback stack by default. When this setting is disabled, CFCATCH.TAGCONTEXT is a zero-length array.

Database exceptions

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

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

Expression exceptions

Property variable	Description
CFCATCH.ERRNUMBER	An internal expression error number, valid only when TYPE="Expression"

Locking exceptions

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

Property variable	Description
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:

Property variable	Description
CFCATCH.MISSINGFILENAME	The name of the file missing in an exception of type MissingInclude.

Exception handling strategies

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

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.

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 an exception occurs during the CFQUERY statement's execution, the application page flow switches to the CFCATCH TYPE="Database" exception handler. It then resumes with the next statement after the CFTRY block, once the CFCATCH TYPE="Database" handler completes.

Similarly, the CFCATCH TYPE="MissingInclude" block handles exceptions raised by the CFINCLUDE tag. 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. --->
<!--- Block executes only if a MissingInclude exception is thrown. --->

    <CFCATCH TYPE="MissingInclude">
        <H1>Missing Include File</H1>
        <CFOUTPUT>
        <UL>
        <LI><B>Message:</B> #CFCATCH.Message#
        <LI><B>Detail:</B> #CFCATCH.Detail#
        <LI><B>File name:</B> #CFCATCH.MissingFilename#
        </UL>
        </CFOUTPUT>
    </CFCATCH>

<!--- Use CFCATCH to test for database errors.--->
<!--- Print error messages. --->
<!--- Block executes only if a Database exception is thrown. --->


    <CFCATCH TYPE="Database">
    <H1>Database Error</H1>
    <CFOUTPUT>
    <UL>
    <LI><B>Message:</B> #CFCATCH.Message#
    <LI><B>Native error code:</B> #CFCATCH.NativeErrorCode#
    <LI><B>SQLState:</B> #CFCATCH.SQLState#
    <LI><B>Detail:</B> #CFCATCH.Detail#
    </UL>
    </CFOUTPUT>
    </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>
```

Custom Exception Types

The TYPE attribute allows a CFTHROW tag to throw an exception of a specific type, which can be caught by a CFCATCH tag that has a matching TYPE attribute.

A CFTHROW tag without a TYPE attribute will throw a TYPE="Application" exception.

Naming conventions

A naming convention for custom exception types follows a convention that is similar to Java class naming conventions: domain name in reverse order, followed by project identifiers, as in this example:

```
<CFTHROW
    TYPE="COM.Allaire.ProjectName"
    ERRORCODE="Dodge14B">
```

The predefined exception types, except for TYPE="APPLICATION" are reserved; for example, <CFTHROW TYPE="Database"> will be rejected.

A CFCATCH tag can specify a custom type as well as one of the predefined types. For example, to catch the exception thrown above, you would use this syntax:

```
<CFCATCH TYPE="COM.Allaire.ProjectName">
```

ColdFusion uses the catch type as a pattern to find a catch handler. For example,

```
<CFTHROW TYPE="MyApp.BusinessRuleException.InvalidAccount">
```

would try to find:

```
<CFCATCH TYPE="MyApp.BusinessRuleException.InvalidAccount">
<CFCATCH TYPE="MyApp.BusinessRuleException">
<CFCATCH TYPE="MyApp">
```

The type comparison is case-insensitive. To match types exactly, rather than performing pattern matching, use the CFSETTING attribute CATCHEXCEPTIONSBYPATTERN=No.

CHAPTER 9

Handling Complex Data with Structures

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

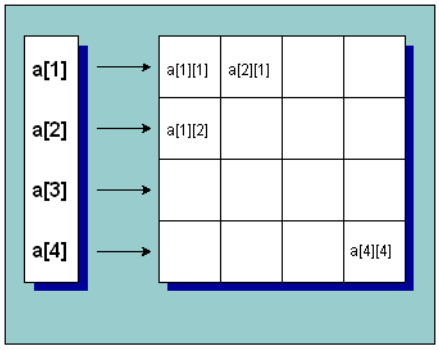
ColdFusion also supports structures for managing lists of key-value pairs. This chapter explains the basics of creating and working with structures.

Contents

About Arrays

Traditionally, an array is a tabular structure used to hold data, much like a spreadsheet table with clearly defined limits and dimensions. A 2-dimensional (2D) array is like a simple table. In ColdFusion, you typically use arrays to temporarily store data. For example, if your site allows users to order goods online, their shopping cart contents can be stored in an array. This allows you to make changes easily without committing the information, which the user may change before completing the transaction, to a database.

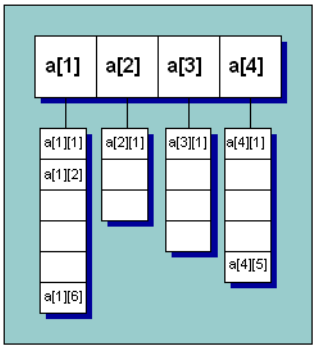
Conventional fixed-size 2D array



A 3-dimensional array is like a cube made up of individual cells.

ColdFusion arrays differ somewhat from traditional arrays because they are dynamic. For example, in a conventional array, array size is constant and symmetrical, whereas in a ColdFusion 2D array you can have 'columns' of differing lengths based on the data that has been added or removed.

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.

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

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.

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]#
```

Multidimensional Arrays

ColdFusion supports dynamic multidimensional arrays. When you declare an array with the ArrayNew function, you can specify up to three dimensions. However, 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, ArrayPrepend to create a new array index at the beginning of the array, and 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*.

Note Because ColdFusion arrays are dynamic, if you add or delete an element from the middle of an array, subsequent index positions all change.

Referencing Elements in Dynamic Arrays

In ColdFusion, array indexes are counted starting with position 1, which means that position 1 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>
```

Populating Arrays with Data

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

Array Functions (Continued)	
Function	**Description**
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 *CFML Language Reference*.

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 can also be used 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

There are three types of notation for structures:

Types of Structure Notation	
Notation	**Description**
Objects.property	Use 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" on page 119 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. The sample structure is called *employee,* and 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 uppercase.

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)>
```

Getting a list of keys in a structure

To get a list of the keys in a CFML structure, you use the StructKeyList function:

```
<CFSET temp=StructKeyList(structure_name, [delimiter] )>
```

The delimiter can be any delimiter, but the default is a comma (,).

The StructKeyArray function returns an array of keys in a structure:

```
<CFSET temp=StructKeyArray(structure_name)>
```

Note The StructKeyList and StructKeyArray functions do not return keys in any particular order. Use the ListSort or ArraySort function to sort the results.

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 [, indicatenotexisting ])
```

This deletes the named key and its associated value. Note that the *indicatenotexisting* parameter indicates whether the function returns FALSE if the named *key* does not exist. The default is FALSE, which means that the function returns Yes regardless of whether *key* exists. If you specify TRUE for this parameter, the function returns Yes if *key* exists and No if it does not.

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

```
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, `structinsert.cfm`, and to 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 structinsert.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 parameters for first time through --->

<CFPARAM NAME="FORM.firstname" DEFAULT="">
<CFPARAM NAME="FORM.lastname" DEFAULT="">
<CFPARAM NAME="FORM.email" DEFAULT="">
<CFPARAM NAME="FORM.phone" DEFAULT="">
<CFPARAM NAME="FORM.department" DEFAULT="">

<!--- If 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: 
<INPUT NAME="firstname" TYPE="text" HSPACE="30" MAXLENGTH="30">
<P>Last Name: 
<INPUT NAME="lastname" TYPE="text" HSPACE="30" MAXLENGTH="30">
<P>EMail: 
<INPUT NAME="email" TYPE="text" HSPACE="30" MAXLENGTH="30">
<P>Phone: 
<INPUT NAME="phone" TYPE="text" HSPACE="20" MAXLENGTH="20">
<P>Department: 
<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. When used as associative arrays, structures index repetitive 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>Department</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 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.
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.
StructKeyArray	Returns an array of keys in the specified structure.
StructKeyExists	Returns TRUE if the specified key is in the specified structure.
StructKeyList	Returns a list of keys in the specified structure.

Structure Functions (Continued)	
Function	**Description**
`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 10

Building Dynamic Forms

This chapter shows you how to use the CFFORM tag to enrich your forms with sophisticated graphical controls, including several Java applet-based controls. These controls can be enabled without the need to code Java directly.

Contents

Creating Forms with the CFFORM Tag

You've already learned how to use HTML forms to gather user input. (See "Using Forms to Specify the Data to Retrieve" on page 30.) This chapter shows you how to use the CFFORM tag to create dynamic forms in CFML. In addition to HTML control types, you can use CFFORM to create forms that contain controls such as:

- Text boxes in which you can specify the appearance such as fonts and colors
- Java applet based controls, inclusing trees, sliders, and grids
- Other Java applets that act as form elements

With CFFORM, you gain the advantage of access to these Java applet-based controls without having to know the Java language, and, you don't have to juggle CFOUTPUT tags and HTML FORM tags to reference ColdFusion variables in your forms.

In addition, most CFFORM controls offer input validation attributes you can use to validate a user's entry, selection, or interaction. This means you don't have to write separate CFML code specifically for input validation as you do in HTML forms.

Using HTML in a CFFORM

You can use the HTML FORM tag in combination with the CFFORM tag. ColdFusion generates HTML forms dynamically from CFFORM tags and passes through to the browser any HTML code it finds in the form. You can use the PASSTHROUGH attribute of the CFFORM, CFINPUT, and CFSELECT tags to enter any HTML attributes that are not explicitly allowed in these tags. The attribute values will be passed through to the HTML generated by these form tags. You can also replace your existing HTML FORM tags with CFFORM and your forms will work fine.

CFFORM controls

Forms created using CFFORM use one or more of the following controls:

CFFORM Controls	
Control	**Description**
CFGRID	A Java applet-based control used to create a data grid you can populate from a query or by defining the contents of individual cells. Grids can also be used to insert, update, and delete records from a data source.
CFSLIDER	A Java applet-based control used to define a slider.
CFINPUT	Used to place radio buttons, check boxes, text input boxes, and password entry boxes.

CFFORM Controls (Continued)	
Control	**Description**
CFTREE and CFTREEITEM	Java applet-based controls used to define a tree control and individual tree control items.
CFTEXTINPUT	A Java applet-based control used to define a text input box.
CFSELECT	Used to define a drop-down list box.
CFAPPLET	Used to embed your own Java applets.

Improving performance with ENABLECAB

The CFFORM ENABLECAB attribute allows you to improve the performance of Java-applet based CFFORM controls. When you use ENABLECAB, ColdFusion prompts the end user to accept a download of the Java classes needed for the CFFORM controls that use them. CAB files are digitally signed using VeriSign digital IDs to ensure file security.

Note The ENABLECAB attribute is supported only for MS Internet Explorer clients that have Authenticode 2.0 installed. Authenticode 2.0 can be downloaded from http://www.microsoft.com/Windows/ie/security/authent2.asp.

Browsers that disable Java

Since each of the Java applet-based controls, CFGRID, CFSLIDER, CFTEXTINPUT, and CFTREE require a Java applet to run, browsers that do not support Java or that have disabled Java execution will not execute the forms that contain these controls. Using the NOTSUPPORTED attribute, ColdFusion allows you to present an error message rather than the blank applet space that appears in the browser. This attribute is available in each of the Java applet-based controls as well as the CFAPPLET tag. You use NOTSUPPORTED to specify the message you want to appear, formatted as HTML, when an application page is loaded by a browser that does not support Java.

Input Validation with CFFORM Controls

The CFINPUT and CFTEXTINPUT tags include the VALIDATE attributes which allows you to specify a valid data type entry for the control. You can validate user entries on the following data types.

Input Validation Controls	
VALIDATE Entry	**Description**
Date	Verifies US date entry in the form mm/dd/yyy.
Eurodate	Verifies valid European date entry in the form dd/mm/yyyy.
Time	Verifies a time entry in the form hh:mm:ss.
Float	Verifies a floating point entry.
Integer	Verifies an integer entry.
Telephone	Verifies a telephone entry. Telephone data must be entered as ###-###-####. The hyphen separator (-) can be replaced with a blank. The area code and exchange must begin with a digit between 1 and 9.
Zipcode	(U.S. formats only) Number can be a 5-digit or 9-digit zip in the form #####-####. The hyphen separator (-) can be replaced with a blank.
Creditcard	Blanks and dashes are stripped and the number is verified using the mod10 algorithm.
Social_security_number	Number must be entered as ###-##-####. The hyphen separator (-) can be replaced with a blank.

When you specify an input type in the VALIDATE attribute, ColdFusion tests for the specified input type when the form is submitted and submits form data only on a successful match. A true value is returned on successful form submission, false if validation fails.

Input Validation with JavaScript

In addition to native ColdFusion input validation using the VALIDATE attribute of the CFINPUT and CFTEXTINPUT tags, the following tags support the ONVALIDATE attribute , which allows you to specify a JavaScript function to handle your CFFORM input validation:

- CFINPUT
- CFSLIDER
- CFTEXTINPUT
- CFTREE

JavaScript objects passed to the validation routine

The following JavaScript objects are passed by ColdFusion to the JavaScript function you specify in the ONVALIDATE attribute:

- form_object
- input_object
- object_value

Handling failed validation

The ONERROR attribute allows you to specify a JavaScript function you want to execute in the event of a failed validation. For example, if you specify a JavaScript function to handle input validation in the ONVALIDATE attribute you can also specify a JavaScript function in the ONERROR attribute to handle a failed validation, which returns a false value. ONERROR is available in the following CFFORM tags:

- CFINPUT
- CFSELECT
- CFSLIDER
- CFTEXTINPUT
- CFTREE

When you specify a JavaScript routine in the ONERROR attribute, ColdFusion passes the following JavaScript objects to the specified routine:

- form_object
- input_object
- object_value
- error message text

To use JavaScript to validate form data:

1. Create a new file in Studio.
2. Edit the page so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>JavaScript Validation</TITLE>

    <SCRIPT>
    <!--

    function testbox(form) {
    Ctrl = form.inputbox1;
    if (Ctrl.value == "" || Ctrl.value.indexOf ('@', 0) == -1) {
    return (false);
    } else
    return (true);
    }

    //-->
    </SCRIPT>

    </HEAD>

    <BODY>
    <H2>JavaScript validation test</H2>

    <P>Please enter your email address:</P>
    <CFFORM NAME="UpdateForm"
        ACTION="update.cfm" >

        <CFINPUT TYPE="text"
            NAME="inputbox1"
            REQUIRED="YES"
    ONVALIDATE="testbox"
        MESSAGE="Sorry, invalid entry."
        SIZE="10"
        MAXLENGTH="10">

    <INPUT TYPE="Submit" VALUE=" Update... ">
    </CFFORM>

    </BODY>
    </HTML>
```

3. Save the page as `validjs.cfm`.
4. View `validjs.cfm` in your browser.

   When you enter an invalid email address, an error appears. Even if you enter a valide email address, and Error 404 appears because you haven't created the action page `update.cfm`.

Code Review

Code	Description

<SCRIPT>
<!--
 	
function testbox(form) {
Ctrl = form.inputbox1;
if (Ctrl.value == "" || Ctrl.value.indexOf
('@', 0) == -1) {
return (false);
} else
return (true);
}
 	
//-->
</SCRIPT>
	JavaScript code to test for valid entry in text box.

ONVALIDATE="testbox"	Text box control parameter that calls the JavaScript test.

See the following Web site for information on JavaScript validation scripts:

- http://www.dannyg.com/javascript

Building Tree Controls with CFTREE

The CFTREE form lets you display hierarchical information in a space-saving collapsible tree populated from data source queries. To build a tree control with CFTREE, you use individual CFTREEITEM tags to populate the control. You can specify one of six built-in icons to represent individual items in the tree control.

To create and populate a tree control from a query:

1. Open a new file named `tree1.cfm` in Studio.
2. Modify the page so that it appears as follows:

```
<CFQUERY NAME="engquery" DATASOURCE="CompanyInfo">
    SELECT FirstName + ' ' + LastName AS FullName
    FROM EMPLOYEES
</CFQUERY>
<CFFORM NAME="form1" ACTION="submit.cfm"
    METHOD="Post">
    <CFTREE NAME="tree1"
    REQUIRED="yes"
    HSCROLL="no"
    VSCROLL="yes">
    <CFTREEITEM VALUE=FullName
```

```
    QUERY="engquery"
    QUERYASROOT="yes"
    IMG="folder,document">
    </CFTREE>
    </CFFORM>
```

3. Save the page and view it in your browser.

Code Review

Code	Description
`<CFTREE NAME="tree1"`	Create a tree and name it tree1.
`REQUIRED="yes"`	Specify that a user must select an item in the tree.
`HSCROLL="no"`	Don't allow horizontal scrolling.
`VSCROLL="yes">`	Allow vertical scrolling.
`<CFTREEITEM VALUE=FullName QUERY="engquery"`	Create an item in the tree and put the results of the query named engquery in it.
`QUERYASROOT="yes"`	Specify the query name as the root level of the tree control.
`IMG="folder,document"`	Use the images "folder" and "document" that ship with ColdFusion in the tree structure.

Grouping output from a query

In a similar query, you may want to organize your employees by the department. In this case, you separate column names with commas in the CFTREEITEM VALUE attribute

To organize the tree based on ordered results of a query:

1. Open a new file named `tree2.cfm` in Studio.
2. Modify the page so that it appears as follows:

```
<!--- CFQUERY with an ORDER BY clause --->
<CFQUERY NAME="deptquery" DATASOURCE="CompanyInfo">
    SELECT Department_ID, FirstName + ' ' + LastName
    AS FullName
    FROM EMPLOYEES
    ORDER BY Department_ID
</CFQUERY>
```

```
<!--- Build the tree control --->
<CFFORM NAME="form1" ACTION="submit.cfm"
    METHOD="Post">

<CFTREE NAME="tree1"
    HSCROLL="no"
    VSCROLL="no"
    BORDER="yes"
    HEIGHT="350"
    REQUIRED="yes">

<CFTREEITEM VALUE="Department_ID, FullName"
    QUERY="deptquery"
    QUERYASROOT="Department_ID"
    IMG="cd,folder">

</CFTREE>

<BR><INPUT TYPE="Submit" VALUE="Submit">
</CFFORM>
```

3. Save the page and view it in your browser.

Code Review

Code	Description
`ORDER BY Department_ID`	Order the query results by department.
`<CFTREEITEM VALUE="Department_ID, FullName"`	Popluate the tree with the Department ID, and under each department, the Full Name for each employee in the department
`QUERYASROOT="Department_ID"`	Make the Department ID the root of the tree
`IMG="cd,folder">`	Use the ColdFusion-supplied images CD and Folder.

Note that the comma-separated items in the IMG and the VALUE attributes correspond. If you leave out the IMG attribute, ColdFusion uses the folder image for all levels in the tree.

CFTREE form variables

The CFTREE tag allows you to force a user to select an item from the tree control by setting the REQUIRED attribute to YES. With or without the REQUIRED attribute, ColdFusion passes two form variables to the application page specified in the CFTREE ACTION attribute:

- form.*treename*.node — Returns the node of the user selection.

- form.*treename*.path — Returns the complete path of the user selection, in the form: root\node1\node2\node_n*value*

The root part of the path is only returned if you set the COMPLETEPATH attribute tof CFTREE to YES; otherwise, the path value starts with the first node.

In the previous example, if the user selects the name "John Allen" in this tree, the following form variables are returned by ColdFusion:

```
form.tree1.node = John Allen
form.tree1.path = Department_ID\3\John Allen
```

You can specify the backslash character used to delimit each element of the path form variable in the CFTREE DELIMITER attribute.

Input validation

Although, the CFTREE does not include a VALIDATE attribute, you can use the REQUIRED attribute to force a user to select an item from the tree control. In addition, you can use the ONVALIDATE attribute to specify the JavaScript code to perform validation.

Structuring Tree Controls

Tree controls built with CFTREE can be very complex. Knowing how to specify the relationship between multiple CFTREEITEM entries will help you handle even the most labyrinthine of CFTREE constructs.

Note The following tree examples all use the result set from folllowing query. To run any of the tree examples, insert the query into your test template:

```
<!--- CFQUERY with an ORDER BY clause --->
<CFQUERY NAME="deptquery" DATASOURCE="CompanyInfo">
    SELECT Department_ID, FirstName + ' ' + LastName
    AS FullName
    FROM EMPLOYEES
    ORDER BY Department_ID
</CFQUERY>
```

Example: One-level tree control

This example consists of a single root and a number of individual items:

```
<CFFORM NAME="form1" ACTION="submit.cfm">
    <CFTREE NAME="tree1">
        <CFTREEITEM VALUE="FullName"
        QUERY="deptquery"
        QUERYASROOT="Department">
    </CFTREE>

<BR><INPUT TYPE="submit" VALUE="Submit">
</CFFORM>
```

Example: Multi-level tree control

When populating a CFTREE, you manipulate the structure of the tree by specifying a TREEITEM parent. In this example, every TREEITEM, except the top level, specifies a parent. The PARENT attribute allows your CFTREE to show the relationships between elements in the tree control.

```
<CFFORM NAME="form1" ACTION="cfform_submit.cfm"
    METHOD="Post">
<CFTREE NAME="tree1" HSCROLL="no" VSCROLL="no"
    BORDER="no">
    <CFTREEITEM VALUE="Divisions">
    <CFTREEITEM VALUE="Development"
        PARENT="Divisions" IMG="folder">
    <CFTREEITEM VALUE="Product One"
        PARENT="Development">
    <CFTREEITEM VALUE="Product Two"
        PARENT="Development">
    <CFTREEITEM VALUE="GUI"
        PARENT="Product Two" IMG="document">
    <CFTREEITEM VALUE="Kernel"
        PARENT="Product Two" IMG="document">
    <CFTREEITEM VALUE="Product Three"
        PARENT="Development">
    <CFTREEITEM VALUE="QA"
        PARENT="Divisions" IMG="folder">
    <CFTREEITEM VALUE="Product One"
        PARENT="QA">
    <CFTREEITEM VALUE="Product Two" PARENT="QA">
    <CFTREEITEM VALUE="Product Three"
        PARENT="QA">
    <CFTREEITEM VALUE="Support"
        PARENT="Divisions" IMG="fixed">
    <CFTREEITEM VALUE="Product Two"
        PARENT="Support">
    <CFTREEITEM VALUE="Sales"
        PARENT="Divisions" IMG="cd">
    <CFTREEITEM VALUE="Marketing"
        PARENT="Divisions" IMG="document">
    <CFTREEITEM VALUE="Finance"
        PARENT="Divisions" IMG="element">
</CFTREE>

</CFFORM>
```

Image names in a CFTREE

When you use the TYPE="Image" attribute, ColdFusion attempts to display an image corresponding to the value in the column, which can be a built-in ColdFusion image name, or an image of your choice in the `cfide\classes` directory or subdirectory, referenced with a relative URL.

The built-in image names are:

- cd
- computer
- document
- element
- folder
- floppy
- fixed
- remote

Embedding URLs in a CFTREE

The HREF attribute in the CFTREEITEM tag allows you to designate tree items as links. To use this feature in a CFTREE, you simply define the destination of the link in the HREF attribute of CFTREEITEM.

To embed links in a CFTREE:

1. Open a new file named `tree3.cfm` in Studio.
2. Modify the page so that it appears as follows:

```
<CFFORM ACTION="submit.cfm">

    <CFTREE NAME="oak"
        HIGHLIGHTHREF="yes"
        HEIGHT="100"
        WIDTH="200"
        HSPACE="100"
        VSPACE="6"
        HSCROLL="no"
        VSCROLL="no"
        BORDER="no"
        DELIMITER="?">

        <CFTREEITEM VALUE="Important Links">
        <CFTREEITEM VALUE="Allaire Home"
            PARENT="Important Links"
            IMG="document"
▸       HREF="http://www.allaire.com">
        <CFTREEITEM VALUE="Allaire Forums"
            PARENT="Important Links"
            IMG="document"
▸       HREF="http://forums.allaire.com">
    </CFTREE>
</CFFORM>
```

3. Save the page and view it in your browser.

Code Review

Code	Description
`HREF="http://www.allaire.com">`	Make the node of the tree a link.
`HREF="http://forums.allaire.com"`	Make the node of the tree a link. Note HREF can refer to the name of a column in a query if the tree item is populated from that query.

Specifying which tree items to send to the action page

When a user selects a tree item and submits the form, the CFTREEITEMKEY variable is appended to the URL passed to the application page specified in the CFFORM ACTION attribute, in the form:

```
http://myserver.com?CFTREEITEMKEY=selected_value
```

You can disable this key by setting the APPENDKEY attribute in the CFTREE tag to No.

Creating Data Grids with CFGRID

The CFGRID tag allows you to build CFFORM grid controls. A grid control resembles a spreadsheet table and can contain data populated from a CFQUERY or from other sources of data. As with other CFFORM tags, CFGRID offers a wide range of data formatting options as well as the option of validating user selections with a JavaScript validation script.

You can also do the following with CFGRID:

- Sort data in the grid alphanumerically
- Update , insert and delete data
- Embed images in the grid

When users select grid data and submit the form, ColdFusion passes the selection information as form variables to the application page specified in the CFFORM ACTION attribute.

Just as the CFTREE tag uses CFTREEITEM, CFGRID uses the CFGRIDCOLUMN tag. You define a wide range of row and column formatting options, as well as a query name, selection options, and so on. You use the CFGRIDCOLUMN tag to define individual columns in the grid.

Populating a grid from a query

To populate a grid from a query:

1. Open a new file named `grid1.cfm` in Studio.
2. Edit the file so that it appears as follows:

```
<CFQUERY NAME="empdata" DATASOURCE="CompanyInfo">
    SELECT * FROM Employees
</CFQUERY>

<CFFORM NAME="Form1" ACTION="submit.cfm" METHOD="Post">

    <CFGRID NAME="employee_grid" QUERY="empdata"
            SELECTMODE="single">
        <CFGRIDCOLUMN NAME="Employee_ID">
        <CFGRIDCOLUMN NAME="LastName">
        <CFGRIDCOLUMN NAME="Department_ID">
    </CFGRID>

<BR><INPUT TYPE="Submit" VALUE="Submit">
</CFFORM>
```

   Note Use the CFGRIDCOLUMN DISPLAY="No" attribute to hide columns you want to retrieve from a data source but not expose to an end user.

3. Save the file and view it in your browser.

Code Review

Code	Description
`<CFGRID NAME="employee_grid" QUERY="empdata"`	Create a grid named "employee_grid" and popluate it with the results of the query "empdata"
`SELECTMODE="single">`	Allow the user to select only one cell.
`<CFGRIDCOLUMN NAME="Employee_ID">`	Put the contents of the Employee_ID column in the query results in the first column of the grid
`<CFGRIDCOLUMN NAME="LastName">`	Put the contents of the LastName column in the query results in the second column of the grid
`<CFGRIDCOLUMN NAME="Department_ID">`	Put the contents of the Department_ID column in the query results in the third column of the grid

Note If you specify a CFGRID tag with a QUERY attribute defined and no corresponding CFGRIDITEM attributes the default grid that is created contains all the columns in the query.

Creating an Updateable Grid

You can build grids to allow users to edit data within them. Users can edit individual cell data, as well as insert, update, or delete rows. To enable grid editing, you specify SELECTMODE="EDIT" in the CFGRID tag and enable the INSERT or DELETE attributes in CFGRID.

You can then use either of two ways to use an updateable grid to make changes to your ColdFusion data sources.

- Create a page to which you pass the CFGRID form variables and in that page perform CFQUERY operations to update data source records.
- Pass grid edits to a page that includes the CFGRIDUPDATE tag, which passes data directly to the data source.

Although using CFQUERY gives you complete control over interactions with your data source, CFGRIDUPDATE provides a much simpler interface for operations that do not require the same level of control.

Editing data in a CFGRID

To enable grid editing, you use the SELECTMODE="EDIT" attribute. When enabled, a user can edit cell data and insert or delete grid rows. When a CFFORM containing a CFGRID is submitted, data about changes to grid cells are stored in one-dimensional arrays you can reference like any other ColdFusion array.

To make the grid editable:

1. Open the file `grid1.cfm` in Studio.
2. Edit the file so that it appears as follows:

```
<CFQUERY NAME="empdata" DATASOURCE="CompanyInfo">
    SELECT * FROM Employees
</CFQUERY>

<CFFORM NAME="GridForm"
    ACTION="handle_grid.cfm">

    <CFGRID NAME="employee_grid"
        HEIGHT=170
        WIDTH=400
        HSPACE=10
        VSPACE=6
        ALIGN="LEFT"
        SELECTCOLOR="white"
        SELECTMODE="edit"
        ROWHEADERS="YES"
        ROWHEADERWIDTH=25
        ROWHEADERALIGN="right"
        COLHEADERS="YES"
        QUERY="empdata"
        GRIDDATAALIGN="left"
        BGCOLOR="yellow"
        INSERT="YES"
        DELETE="YES"
        SORT="YES"
        MAXROWS=60>

        <CFGRIDCOLUMN NAME="Employee_ID"
            HEADER="Employee ID"
            WIDTH=80
            ITALIC="NO"
            HEADERALIGN="center"
            HEADERITALIC="NO"
            HEADERBOLD="YES"
            DISPLAY="NO">

        <CFGRIDCOLUMN NAME="LastName"
            HEADER="Last Name"
            WIDTH=80
            ITALIC="NO"
            HEADERALIGN="center"
```

```
            HEADERITALIC="NO"
            HEADERBOLD="YES"
            DISPLAY="YES"
            SELECT="YES">

        <CFGRIDCOLUMN NAME="Department_ID"
            HEADER="Department"
            WIDTH=240
            ITALIC="No"
            HEADERALIGN="center"
            HEADERITALIC="No"
            HEADERBOLD="Yes"
            BOLD="Yes"
            DISPLAY="Yes">

    </CFGRID>

<INPUT TYPE="Submit" VALUE="Submit">

</CFFORM>
```

3. Save the file as `grid2.cfm`.

To update the data source with CFQUERY:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Catch submitted grid values</TITLE>
</HEAD>
<BODY>

<H3>Grid values for FORM.employee_grid row updates</H3>

<CFIF IsDefined("form.employee_grid.rowstatus.action")>

    <CFLOOP INDEX = "Counter" FROM = "1" TO =
        #ArrayLen(form.employee_grid.rowstatus.action)#>

    <CFOUTPUT>
    The row action for #Counter# is:
    #form.employee_grid.rowstatus.action[Counter]#
    <BR><BR>
    </CFOUTPUT>

    <CFIF form.employee_grid.rowstatus.action[Counter] IS "D">

    <CFQUERY NAME="DeleteExistingEmployee"
        DATASOURCE="CompanyInfo">
        DELETE FROM Employees
        WHERE
Employee_ID=#form.employee_grid.original.Employee_ID[Counter]#
```

```
    </CFQUERY>

<CFELSEIF form.employee_grid.rowstatus.action[Counter] IS "U">

    <CFQUERY NAME="UpdateExistingEmployee"
        DATASOURCE="CompanyInfo">
        UPDATE Employees
        SET
        LastName='#form.employee_grid.LastName[Counter]#',
        Department_ID=#form.employee_grid.Department_ID[Counter]#
        WHERE

Employee_ID=#form.employee_grid.original.Employee_ID[Counter]#
    </CFQUERY>

<CFELSEIF form.employee_grid.rowstatus.action[Counter] IS "I">

    <CFQUERY NAME="InsertNewEmployee"
        DATASOURCE="CompanyInfo">
        INSERT into Employees
        (Employee_ID, LastName, Department_ID)
        VALUES (#form.employee_grid.Employee_ID[Counter]#,
        '#form.employee_grid.LastName[Counter]#',
#form.employee_grid.Department_ID[Counter]#)
    </CFQUERY>

    </CFIF>
    </CFLOOP>
</CFIF>

</BODY>
</HTML>
```

3. Save the file as `handle_grid.cfm`.
4. View `grid2.cfm` in your browser, make changes to the grid, and then submit them.

To update the data source with CFQUERY

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<CFGRIDUPDATE GRID="Employee_grid"
    DATASOURCE="CompanyInfo"
    TABLENAME="Employees"
    KEYONLY="NO">
```

3. Save the file as `handle_grid.cfm`.
4. View `grid2.cfm` in your browser, make changes to the grid, and then submit them.

How user edits are stored

The following arrays are created to keep track of edits to grid rows and cells:

Arrays Used to Store Grid Cell Edit Information	
Array reference	**Description**
gridname.colname [*row_index*]	Stores the new value of an edited grid cell
gridname.Original.*colname* [*row_index*]	Stores the original value of the edited grid cell
gridname.RowStatus.Action [*row_index*]	Stores the edit type made against the edited grid cell.

For example, you have an updateable CFGRID called "mygrid" consisting of two displayable columns, col1, col2, and one hidden column, col3. When an end user selects and changes data in a row, arrays are created to store the original values for all columns as well as the new column values for rows that have been updated, inserted, or deleted.

```
mygrid.col1[ row_index ]
mygrid.col2[ row_index ]
mygrid.col3[ row_index ]
mygrid.original.col1[ row_index ]
mygrid.original.col2[ row_index ]
mygrid.original.col3[ row_index ]
```

Where *row_index* is the array index containing the grid data.

If the end user makes a change to a single cell in col2, you can reference the edit operation, the original cell value, and the edited cell value in the following arrays:

```
<CFSET edittype = mygrid.RowStatus.Action[1]><BR>
<CFSET new_value = mygrid.col2[1]><BR>
<CFSET old_value = mygrid.original.col2[1]>
```

Multi-row edits

The use of arrays to track changes allows ColdFusion to manage changes to more than one row in a CFGRID. ColdFusion coordinates entries in the arrays used to store edit type (Update, Insert, or Delete), with arrays that store original grid data and edited grid data. For each grid cell edit, an entry is created in the RowStatus array, and corresponding entries are made in the arrays that store the new cell value and the original cell value.

Building Slider Bar Controls

You can use the CFSLIDER control to create a slider control and define a wide range of formatting options for slider label text, , colors for the groove in which the slider knob moves, label font name, size, boldface, italics, and color, as well as slider scale increments, range, positioning, and behavior.

As with CFTREE and CFGRID, input valication can be serviced with a JavaScript specified in the ONVALIDATE attribute.

To create a slider control:

1. Create a new file in Studio.
2. Modify the file so that it appears a follows:

```
<CFFORM NAME="Form1" ACTION="submit.cfm"
    METHOD="Post">

    <CFSLIDER NAME="myslider"
        GROOVECOLOR="black"
        BGCOLOR="white"
        TEXTCOLOR="black"
        FONT="Trebuchet MS"
        BOLD="yes"
        RANGE="0,1000"
        SCALE="10"
        VALUE="640"
        FONTSIZE="24"
        LABEL="Slider %value%"
        WIDTH="400">

</CFFORM>
```

3. Save the file as `slider.cfm` and view it in your browser.

CFSLIDER form variable

The value of the form variable passed from a CFSLIDER control to a ColdFusion application page is determined by the position of the slider on the scale. The form variable is passed as:

```
slider_name=slider_value
```

In the earlier example, the form variable would have been passed as:

```
myslider=slider_value
```

Building Text Entry Boxes

The CFTEXTINPUT tag is similar to the HTML INPUT=text tag. With CFTEXTINPUT, however, you can also specify font and alignment options, as well as enable input

validation methods using either a JavaScript or the VALIDATE attribute in CFTEXTINPUT.

The following example shows a basic CFTEXTINPUT control. This example validates a date entry, which means that a user must enter a valid date in the form *mm/dd/yy*. For a complete list of validation formats, refer to the *CFML Language Reference*.

```
<BR>Please enter a date:
<CFFORM NAME="Form1"
    ACTION="cfform_submit.cfm"
    METHOD="Post">

    <CFTEXTINPUT NAME="entertext"
        VALUE="mm/dd/yy"
        MAXLENGTH="10"
VALIDATE="date"
        FONT="Trebuchet MS">

    <BR>

    <INPUT TYPE="Submit"
           VALUE="Submit">

</CFFORM>
```

CFTEXTINPUT form variable

The value of the form variable passed from a CFTEXTINPUT control to a ColdFusion application page is determined by the entry in the CFTEXTINPUT control. The form variable is passed as:

```
textinput_name=textinput_value
```

In the example just above, the form variable would have been passed as:

```
entertext=textinput_value
```

So in the destination application page, the form variable is referenced as #entertext#

Building Drop-Down List Boxes

The drop-down list box you can create with CFSELECT is similar to the HTML SELECT tag. However, CFSELECT gives you more control over user inputs, error handling, and allows you to populate the selection list from a query.

When you populate a CFSELECT with data from a query, you only need to specify the name of the query that is supplying data for the CFSELECT and the query column name for each list element you want to display.

To populate a drop-down list box with query data using CFSELECT:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<CFQUERY NAME="getNames"
    DATASOURCE="CompanyInfo">
    SELECT * FROM Employees
</CFQUERY>

<CFFORM NAME="Form1" ACTION="submit.cfm"
    METHOD="Post">

    <CFSELECT NAME="employeeNames"
        QUERY="getNames"
        VALUE="Employee_ID"
        DISPLAY="FirstName"
        REQUIRED="yes"
        MULTIPLE="yes"
        SIZE="8">
    </CFSELECT>

    <BR><INPUT TYPE="Submit"
            VALUE="Submit">

</CFFORM>
```

3. Save the file as `selectbox.cfm` and view it in your browser.

Note that because the MULTIPLE attribute is used, the user can select multiple entries in the select box. Also, because the VALUE tag specifies the primary key for the Employee table, this data is used in the form variable that is passed to the application page specified in ACTION.

Embedding Java Applets

The CFAPPLET tag allows you to embed Java applets in a CFFORM. To use CFAPPLET, you must first register your Java applet using the ColdFusion Administrator Applets page. In the Administrator, you define the interface to the applet, encapsulating it so that each invocation of the CFAPPLET tag is very simple.

CFAPPLET offers several advantages over using the HTML APPLET tag:

- Return values — Since CFAPPLET requires a form field name attribute, you can avoid having to code additional JavaScript to capture the applet's return values. You can reference return values like any other ColdFusion form variable: `form.variablename`.
- Ease of use — Since the applet's interface is defined in the Administrator, each instance of the CFAPPLET tag in your pages only needs to reference the applet's name and specify a form variable name.

- Parameter options — You can override parameter values you defined in the Administrator by specifying the parameter value pair in CFAPPLET. Unless overridden, ColdFusion uses the parameter value pairs you defined in the Administrator.

When an applet is registered, enter just the applet source and the form variable name:

```
<CFAPPLET APPLETSOURCE="Calculator"
    NAME="calc_value">
```

By contrast, with the HTML APPLET tag, you'd have to invoke all the applet's parameters every time you wanted to use it in a ColdFusion page.

Registering a Java applet

Before you can use a Java applet in your ColdFusion pages, you must first register the applet in the Administrator.

To register a Java applet:

1. Open the ColdFusion Administrator by clicking on the Administrator icon in the ColdFusion Program group and entering the Administrator password (if required).
2. Click the Applets button to open the Registered Applets page.
3. Enter a name for the applet you want to register and click Register New Applet. Enter the information your applet requires, and choose the height, width, vertical and horizontal space, and alignment you want.

Applet registration fields are explained in the following table.

Applet Registration Fields	
Field	**Description**
Codebase	Enter the base URL of the applet: the directory that contains the applet components. The applet class files must be located within the web browser root directory. Example: `http://servername/classes`
Code	This is the name of the file that contains the applet subclass. The filename is relative to the codebase URL. The *.class file extension is not required.
Method	Enter the method name in the applet that returns a string value. You use this method name in the NAME attribute of the CFAPPLET tag to populate a form variable with the method's value. If the applet has no method, leave this field blank.

Applet Registration Fields (Continued)	
Field	**Description**
Height	Enter a measurement in pixels for the vertical space for the applet.
Width	Enter a measurement in pixels for the horizontal space for the applet.
Vspace	Enter a measurement in pixels for the space above and below the applet.
Hspace	Enter a measurement in pixels for the space on each side of the applet.
Align	Choose the alignment you want.
Java Not Supported Message	This message is displayed by browsers that do not support Java applets. If you want to override this message, you specify a different message in the CFAPPLET NOTSUPPORTED attribute.
Parameter Name	Enter a name for a required applet parameter. Your Java applet will typically provide the parameter name needed to use the applet. Enter each parameter in a separate parameter field.
Value	For every parameter you enter, define a default value. Your applet documentation will provide guidelines on valid entries.

Click Create to complete the process.

Using CFAPPLET to embed an applet

Once you've registered an applet, you can use the CFAPPLET tag to place the applet in a ColdFusion page. The CFAPPLET tag has two required attributes, APPLETSOURCE and NAME. Since the applet has been registered, and each applet parameter defined with a default value, you can invoke the applet with a very simple form of the CFAPPLET tag:

```
<CFAPPLET APPLETSOURCE="appletname" NAME="form_variable">
```

Overriding alignment and positioning values

To override any of the values defined in the Administrator for the applet, you can use the optional CFAPPLET parameters to specify custom values. For example, the following CFAPPLET tag specifies custom spacing and alignment values:

```
<CFAPPLET APPLETSOURCE="myapplet"
    NAME="applet1_var"
    HEIGHT=400
    WIDTH=200
    VSPACE=125
    HSPACE=125
    ALIGN="left">
```

Overriding parameter values

You can also override the values you assigned to applet parameters in the Administrator by providing new values for any parameter. Note that in order to override a parameter, you must have already defined the parameter and a default value for it in the ColdFusion Administrator Applets page.

```
<CFAPPLET APPLETSOURCE="myapplet"
    NAME="applet1_var"
    Param1="registered parameter"
    Param2="registered parameter">
```

Handling form variables from an applet

The CFAPPLET tag requires you to specify a form variable name for the applet. This variable, referenced like other ColdFusion form variables, `form.`*`variable_name`* holds the value the applet method provides when it is executed in the CFFORM.

Not all Java applets return values. Some, like many graphical widgets, do not return a specific value; they do their flipping, spinning, fading, exploding, and that's that. For this kind of applet, the method field in the Administrator remains empty. Other applets, however, do have a method that returns a value. You can only use one method for each applet you register. If an applet includes more than one method that you want to access, you can register the applet with a unique name an additional time for each method you want to use.

To reference a Java applet return value in your application page:

1. Specify the name of the method in the Register New Applet page of the ColdFusion Administrator.
2. Specify the method name in the NAME attribute of the CFAPPLET tag when you code your CFFORM.

When your page executes the applet, a form variable is created with the name you specified. If you don't specify a method, no form variable is created.

CHAPTER 11

Indexing and Searching Data

You can provide a full-text search capability for documents and data sources on a ColdFusion site by enabling the Verity search engine.

Contents

Searching a ColdFusion Web Site

Until now, you've searched for records in databases based on the value of particular fields using ODBC. However, to efficiently search through paragraphs of text or files of varying types requires full-text search capabilites. The Verity, Inc. search engine is bundled with ColdFusion to provide full-text indexing and searching.

The ColdFusion online documentation employs Verity to allow you to search the installed document set.

Here are some of the possible uses for Verity in ColdFusion:

- Index your Web site and provide a generalized search mechanism, such as a form interface, for executing searches.
- Index specific directories containing documents for subject-based searching.
- Index CFQUERY result sets, giving your end users the ability to search against the data. Since collections are made up of data optimized for retrieval, this method is much faster than performing multiple database queries to return the same data.
- Index CFLDAP and CFPOP query results.
- Manage and search collections generated outside of ColdFusion using native Verity tools. This additional capability requires only that the full path to the collection be specified in the index command.
- Index email generated by ColdFusion application pages and create a searching mechanism for the indexed messages.
- Build collections of inventory data and make those collections available for searching from your ColdFusion application pages.
- Support international users in a range of languages from both the CFINDEX and CFSEARCH tags.

Advantages of using Verity

Verity can index the output from queries so that you or an end user can search against the result sets. This has a clear advantage in speed of execution because pointers to the result sets are stored in a Verity index that is optimized for searching. You can reduce the programming overhead of query constructs by allowing users to construct their own queries and execute them directly. You need only be concerned with presenting the output to the client browser.

Verity can index database text fields, such as notes and product descriptions, that cannot be effectively indexed by native database tools.

When indexing collections containing documents in Adobe Acrobat (PDF) format, Verity scans for the document title (if one has been entered in Acrobat Exchange). The document title displays in the search results list.

Indexing Web pages returns the URL for each document. This is a valuable document management feature.

Online Verity training

A video titled "Creating Search Engines with Verity" is available at http://alive.allaire.com. The video gives an overview of the Verity implementation in ColdFusion and illustrates the development process with sample code.

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.

Supported File Types

The ColdFusion Verity implementation supports a wide array of document types. This means you can index Web pages, ColdFusion applications, and many binary document types and produce search results that include summaries of these documents.The following table lists the supported document types.

Supported File Types		
Documents	**Versions**	**Type**
Text files		
HTML, CFML, DBM, SGML, XML,	N/A	Text
ANSI, ASCII, Plain Text	N/A	Text
Word processors		
Adobe Acrobat (PDF)	all	Binary
Adobe FrameMaker (MIF)	all	Binary
Aplix Words	4.2	Binary
Corel WordPerfect for Windows	5.x 6, 7, 8	Binary
Corel WordPerfect for Macintosh	2, 3	Binary
Lotus Ami Pro	2, 3	Binary
Lotus Ami Pro Write Plus	all	Binary
Lotus Word Pro	96, 97	Binary
Microsoft Office	95, 97	Binary
MS Rich Text Format (RTF)	1.x, 2.0	Binary
MS Word for Windows	2, 6, 95, 97	Binary

Supported File Types (Continued)		
Documents	**Versions**	**Type**
MS Word for DOS	4, 5, 6	Binary
MS Word for Macintosh	4.0, 5.0, 6.0	Binary
MS Notepad, WordPad	all	Binary
MS Write, MS Works	all	Binary
XYWrite	4.12	Binary
Spreadsheets		
Corel QuattroPro	7, 8	Binary
Lotus 1-2-3 for DOS/Windows	2.0, 3.0, 4.0, 5.0, '96, '97	Binary
Lotus 1-2-3 for OS/2	2	Binary
MS Excel	3, 4, 5, '95, '97	Binary
MS Works	all	Binary
Presentation		
Corel Presentations	7.0, 8.0	Binary
Lotus Freelance	96, 97	Binary
MS PowerPoint	4.0, 95, 97	Binary

Support for International Languages

The ColdFusion International Language Search Pack can be purchased and installed to index data in any the following languages:

- Danish
- Dutch
- Finnish
- French
- German
- Italian
- Norwegian
- Portuguese

- Spanish
- Swedish

The default language for Verity collections is English. To index data in one of the supported languages, you must select the language from the drop-down list when you create a collection on the ColdFusion Administrator Verity page. You must then enter the selected language as a value of the LANGUAGE attribute in both the CFINDEX and CFSEARCH tags used against that collection.

To order the Language Pack, contact Allaire Customer Service or visit our online store at http://www.allaire.com/store. The default installation directory for the dictionaries is in `\cfusion\verity\common`.

Steps in Creating a Searchable Data Source

There are several steps in creating a searchable data source:

1. Create a collection.

   This can be done either through the ColdFusion Administrator or programmatically.

2. Populate and index the collection

   This involves selecting the data and generating the index.

3. Design a search interface and a results page so that users can access the searchable data source.

Creating a Collection

The Verity engine performs searches against *collections*. A collection is a special database created by Verity that contains pointers to the indexed data that you specify for that collection. ColdFusion's Verity implementation supports collections of three basic data types:

- Text files such as HTML pages and CFML templates.
- Binary documents (see the Supported File Types list).
- Result sets returned from CFQUERY, CFLDAP, and CFPOP queries.

You can build a collection from individual documents or an entire directory tree. Collections can be stored anywhere, so you have a lot of flexibility in accessing indexed data. This adds enormous value to any content-rich Web site.

ColdFusion provides twodifferent ways to create a Verity collection. You can:

- Make selections on the ColdFusion Administrator Verity page
- Code the CFCOLLECTION tag

Using the ColdFusion Administrator to create a collection

To create a new collection:

1. Open the ColdFusion Administrator Verity page.

   If you checked the option to install the ColdFusion Documentation, the documentation collection is listed by default. The Verity engine is used to search our online documents.
2. In the Add a Collection section, enter a name for the collection.
3. Enter a path for the location of the new collection.

   By default, new collections are added to `\Cfusion\Verity\Collections\`.
4. If you have an International Language Search Pack installed, you can select a language for the collection from the drop-down list.
5. Click Create a new collection, then click Apply.

   When the collection is created, the name and full path of the new collection appear in the Verity Collections list at the top of the page.

You can easily enable access to a collection on the network by creating a local reference (an alias) for that collection. It only needs to be a valid Verity collection; it doesn't matter whether it was created within ColdFusion or another tool.

To add an existing collection:

1. In the Add a Collection section, enter the collection alias.
2. Enter the full path to the collection.
3. Select Language if needed.
4. Click Map an existing collection.
5. Click Apply.

If the collection is subsequently moved, the alias path must be updated. The Delete command, when used on a mapped collection, only deletes the alias.

Creating a collection with the CFCOLLECTION tag

Creating and maintaining collections from a CFML application eliminates the need to access the ColdFusion Administrator. This can be an advantage when you need to schedule these tasks or to allow users to perform them without exposing the Administrator to users.

To create a simple collection form page:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
```

```
<HEAD>
    <TITLE>Collection Creation Input Form</TITLE>
</HEAD>

<BODY>
<H2>Specify a collection</H2>
<FORM ACTION="collectioncreateaction.cfm" METHOD="POST">
    <P>Collection name: <INPUT TYPE="text" NAME="CollectionName"
SIZE="25"></P>
    <P>What do you want to do with the collection?</P>
    <INPUT TYPE="radio"
        NAME="CollectionAction"
        VALUE="Create" checked>Create<BR>
    <INPUT TYPE="radio"
        NAME="CollectionAction"
        VALUE="Repair">Repair<BR>
    <INPUT TYPE="radio"
        NAME="CollectionAction"
        VALUE="Optimize">Optimize<BR>
    </P>
    <INPUT TYPE="submit"
        NAME="submit"
        VALUE="Submit">
</FORM>

</BODY>
</HTML>
```

3. Save the file as `collectioncreateform.cfm`.

Note that this file simply shows how the form variables are used and does not perform error checking.

To create a collection action page:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>CFCOLLECTION</TITLE>
</HEAD>

<BODY>
<H2>Collection creation</H2>

<CFOUTPUT>

    <CFSWITCH EXPRESSION=#FORM.CollectionAction#>
        <CFCASE VALUE="Create">
            <CFCOLLECTION ACTION="Create"
            COLLECTION="#FORM.CollectionName#"
            PATH="C:\CFUSION\Verity\Collections\">
            <P>The collection #FORM.CollectionName# is created.
        </CFCASE>

        <CFCASE VALUE="Repair">
            <CFCOLLECTION ACTION="REPAIR"
            COLLECTION="#FORM.CollectionName#">
            <P>The collection #FORM.CollectionName# is repaired.
        </CFCASE>

        <CFCASE VALUE="Optimize">
            <CFCOLLECTION ACTION="OPTIMIZE"
            COLLECTION="#FORM.CollectionName#">
            <P>The collection #FORM.CollectionName# is optimized.
        </CFCASE>

        <CFCASE VALUE="Delete">
            <CFCOLLECTION ACTION="DELETE"
            COLLECTION="#FORM.CollectionName#">
            <P>Collection deleted.
        </CFCASE>
    </CFSWITCH>
</CFOUTPUT>
</BODY>
</HTML>
```

3. Save the file as `collectioncreateaction.cfm`.
4. View the file `collectioncreateform.cfm` in your browser, enter values and submit the form.

Populating and Indexing a Collection

At this point, the new collection is just an empty shell. To populate the collection with indexed data, you can use either of two methods:

- The CF Administrator
- The CFINDEX tag

  You can use the Verity Wizard in ColdFusion Studio to create the templates to make yur documents searchable. To run the wizard, click File > New and select the Verity Wizard from the CFML tab of the New Document dialog.

Note You can index and search against Verity collections created outside of ColdFusion by using the EXTERNAL attribute of CFINDEX and CFSEARCH.

Selecting an indexing method

Use the following guidelines to determine which method to use.

Using the CF Administrator or CFINDEX	
Use the Administrator if	**Use the CFINDEX tag if**
You want to index document files.	You want to index ColdFusion query results.
The collection won't be updated very frequently.	You need to dynamically populate or update a collection from a ColdFusion application page.
You want to generate the collection without writing any CFML code.	Your collection needs to be updated frequently.
You want to generate a one-time collection.	Your collection needs to be updated by other people.

Using ColdFusion Administrator

To use ColdFusion Administrator to index a collection:

1. Select a collection name in the Verity Collections box.
2. Click Index to open the index page. The selected collection name appears at the top of the page.
3. Enter a single file type or multiple file types separated by commas.
4. Type in the directory path for the collection or click Browse Server and navigate to the directory in which to begin the index.

5. Check the Recursively index subdirectories box if you want to extend the indexing operation to all directories below the selected path.
6. Optionally, you can enter a Return URL to prepend to all indexed files. This allows you to easily create a link to any of the files in the index. A typical entry might be something like `http://localhost/wwwroot/`.
7. If the International Language Search Pack is installed, you can select one of the supported languages.
8. Click Update to begin the indexing process. The time required to generate the index depends on the number and size of the selected files in the path.

As you can see, this interface allows you to easily build a very specific index based on the file extension and path information you enter. In most cases, you do not need to change your server file structures to accommodate the generation of indices.

In your ColdFusion application, you can populate and search multiple collections, each of which can be designed to focus on a specific group of documents or queries, according to subject, document type, location, or any other logical grouping. Searches can be performed against multiple collections, giving you lots of flexibility in designing your search interface.

Using CFINDEX

To select which collection to index:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Select the Collection to Index</TITLE>
</HEAD>

<H2>Pick which index you want to build</H2>

<FORM METHOD="Post" ACTION="collectionindexaction.cfm">
    <P>Enter the collection you want to populate:
    <INPUT TYPE="text" NAME="IndexColl" SIZE="25" MAXLENGTH="35"></P>
    <P>Enter the location of the files in the collection:
    <INPUT TYPE="text" NAME="IndexDir" SIZE="50" MAXLENGTH="100"></P>

    <INPUT TYPE="submit" NAME="submit" VALUE="Index">

</FORM>

</BODY>
</HTML>
```

3. Save the file as `collectionindexform.cfm`

To use CFINDEX to index a collection:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>Creating Index</TITLE>
</HEAD>
<BODY>
<H2>Indexing Complete</H2>

<CFINDEX COLLECTION="#Form.IndexColl#"
    KEY="#Form.IndexDir#"
    ACTION="REFRESH"
    TYPE="PATH"
    URLPATH="#Form.IndexDir#"
    EXTENSIONS=".htm, .html"
    RECURSE="Yes"
    LANGUAGE="English">

<CFOUTPUT>
    The collection #Form.IndexColl# has been indexed.
</CFOUTPUT>
</BODY>
</HTML>
```

3. Save the file as `collectionindexaction.cfm`
4. View `collectionindexform.cfm` in your browser, enter values, and then click Index.

Building a Search Interface

Now that you've created and indexed a searchable data source, you need to build a search interface to allow users to access the data source. The CFSEARCH tag provides users with a set of operators and modifiers to create sophisticated query expressions. We'll explore these options in detail below, but first let's take a look at getting a basic search application up and running.

Using the Verity wizard in Studio

To quickly create a search application for an existing collection, click the File > New command in ColdFusion Studio and select the Verity Wizard in the CFML tab of the New Document dialog. The wizard creates a set of application pages based on the entries you make in the wizard dialogs.

You can customize the search interface by adding instructional text for users and applying styles to the form pages.

Basic search operations

To search the collection:

1. Create a new file in Stuio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Select the collection to search</TITLE>
</HEAD>

<BODY>
<H2>Search</H2>

<FORM METHOD="Post" ACTION="collectionsearchaction.cfm">
    <P>Enter the collection you want to search:
    <INPUT TYPE="text" NAME="collection" SIZE="25" MAXLENGTH="35"></
P>
    <P>Select the type of search:<BR>
    <INPUT TYPE=radio
        NAME=type
        VALUE=Simple checked> Simple<BR>
    <INPUT TYPE=radio
        NAME=type
        VALUE=Explicit> Explicit<BR>

    <P>Enter a search string:</P>
    <INPUT TYPE=text
        NAME=searchstring SIZE=50>

    <P><INPUT TYPE=submit
        NAME=search1
        VALUE="Search">
    <INPUT TYPE=reset
        VALUE="Reset">
</FORM>

</BODY>
</HTML>
```

3. Save the file as `collectionsearchform.cfm`.

To present the results of the search to the user:

1. Create a new file in Stuio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Search output template</TITLE>
</HEAD>
```

```
<BODY>
<CFSEARCH NAME="Search1"
    COLLECTION="#form.collection#"
    FORM TYPE="#form.type#"
    CRITERIA="#form.searchstring#">

<H2>Search Results</H2>

<CFOUTPUT>
    #Search1.RecordCount# found out of
    #Search1.RecordsSearched# searched.
</CFOUTPUT>

<HR NOSHADE>
    <CFOUTPUT QUERY="Search1">
        <A HREF="#Search1.URL#">#Search1.title#</A><BR>
    </CFOUTPUT>
<HR NOSHADE>
</BODY>
</HTML>
```

3. Save the file as `collectionsearchaction.cfm`.
4. View the file `collectionsearchform.cfm` in your browser, enter values in the form, then submit it.

Summarization

As part of the indexing process, Verity automatically produces a summary of every document file or query result set. The default summarization selects the best sentences, based on internal rules, up to a maximum of 500 characters. Summarization information is returned by default with every CFSEARCH operation. For more information on this topic, see the Allaire Knowledge Base article, "Custom1, Custom2 and Summary Fields" (ID# 1081) on our Web site at http://www.allaire.com/Support/KnowledgeBase/SearchForm.cfm.

To access the summary, invoke the property in the following form:

```
#search_query.Summary#
```

For example, in a search operation where the value of the NAME attribute is "mysearch" the following CFML outputs the summary of the search results:

```
<CFOUTPUT QUERY="mysearch">
    #Summary#<BR>
</CFOUTPUT>
```

For information on an advanced summarization technique, see the Allaire Knowledge Base article, "Synchronizing information stored in Verity Collection Document Fields with Corresponding Data from a Database" (ID# 1161) on our Web site at http://www.allaire.com/Support/KnowledgeBase/SearchForm.cfm.

CFSEARCH properties

Three properties are generated for each CFSEARCH query that provide information about a particular query:

- RecordCount — The total number of records returned by the query.
- CurrentRow — The current row of the query being processed by CFOUTPUT.
- RecordsSearched — The total number of records in the index that were searched. If no records were returned in the search, this property returns a null value.

Indexing database query results

Indexing the result set from a ColdFusion query involves an extra step not required when indexing documents. You need to code the query and output parameters, then point the CFINDEX tag at the result set from a CFQUERY, CFLDAP, or CFPOP query.

To index a ColdFusion query:

1. Create the collection on the ColdFusion Administrator Verity page.
2. Execute a query and output the data.
3. Populate the collection using the CFINDEX tag.

To populate a collection from a CFQUERY you specify a KEY, which corresponds to the primary key of the data source, and the BODY, the column in which you want to conduct searches. The following extract shows only the CFQUERY and CFINDEX parts of the process.

```
<!--- Select the entire table --->
<CFQUERY NAME="Messages"
    DATASOURCE="MyMail">
    SELECT *
        FROM Messages
</CFQUERY>

<!--- Output the result set --->
<CFOUTPUT QUERY="Messages">
    #Message_ID#, #Subject#, #Title#, #MessageText#

</CFOUTPUT>

<!--- Index the result set --->
<CFINDEX COLLECTION="DBINDEX"
    ACTION="UPDATE"
    TYPE="CUSTOM"
    BODY="MessageText"
    KEY="Message_ID"
    TITLE="Subject"
    QUERY="Messages">
```

This CFINDEX statement specifies the MessageText column as the core of the collection and names the table's primary key, the Message_ID column, as the KEY value. Note that the TITLE attribute names the Subject column. The TITLE attribute can be used to designate an output parameter.

To index more than one column in a collection, enter a comma-separated list of column names for values of the BODY attribute, such as:

```
BODY=FirstName,LastName,Company
```

Advantages of indexing a data source

The main advantage of performing searches against a Verity collection over using CFQUERY alone is that the database is indexed in a form that provides faster access. Use this technique instead of CFQUERY in the following cases:

- You want to index textual data. Verity collections containing textual data can be searched much more efficiently with CFINDEX than searching a database with CFQUERY.
- You want to give your users access to data without interacting directly with the data source itself.
- You want to improve the speed of queries.
- You want your end users to run queries but not update a database table.
- You do not want to expose your data source.

Indexing CFLDAP Query Results

The widespread use of the Lightweight Directory Access Protocol to build searchable directory structures, both internally and across the Web, provides ColdFusion developers with new opportunities to add value to the sites they create. Contact information or other data from an LDAP-accessible server can be indexed and searched by users. Remember to create the collection in the Administrator.

Two things to remember when creating an index from an LDAP query:

- Because LDAP structures vary greatly, you must know the server's directory schema and the exact name of every LDAP attribute you intend to use in a query.
- The records on an LDAP server can be subject to frequent change. You may want to re-index the collection before processing a search request.

In the example below, the search criterion is records with a telephone number in the 617 area code. Generally, LDAP servers use the Distinguished Name (dn) attribute as the unique identifier for each record, so that is used as the KEY value for the index.

```
<!--- Run the LDAP query --->
<CFLDAP NAME="OrgList"
    SERVER="myserver"
    ACTION="query"
    ATTRIBUTES="o, telephonenumber, dn, mail"
    SCOPE="onelevel"
    FILTER="(|(O=a*) (O=b*))"
    SORT="o"
    START="c=US">

<!--- Output query result set --->
<CFOUTPUT QUERY="OrgList">
    DN: #dn# <BR>
    O: #o# <BR>
    TELEPHONENUMBER: #telephonenumber# <BR>
    MAIL: #mail# <BR>
==============================<BR>
</CFOUTPUT>

<!--- Index the result set --->

<CFINDEX ACTION="update"
    COLLECTION="ldap_query"
    KEY="dn"
    TYPE="custom"
    TITLE="o"
    QUERY="OrgList"
    BODY="telephonenumber">

<!--- Search the collection --->
<!--- Use the wildcard * to contain the search string --->
<CFSEARCH COLLECTION="ldap_query"
    NAME="s_ldap"
    CRITERIA="*617*">

<!--- Output returned records --->
<CFOUTPUT QUERY="s_ldap">
    #Key#, #Title#, #Body# <BR>
</CFOUTPUT>
```

Indexing CFPOP Query Results

The contents of mail servers are generally quite volatile; specifically, the MessageNumber is reset as messages are added and deleted. To avoid mismatches between the unique MessageNumber identifiers on the server and in the Verity collection, it's a good idea to re-index the collection before processing a search.

As with the other query types, you need to provide a unique value for the KEY attribute and enter the data fields to index in the BODY attribute.

```
<!--- Run POP query --->
<CFPOP ACTION="getall"
    NAME="p_messages
    SERVER="mail.mycompany.com"
    USERNAME="user1"
    PASSWORD="user1">

<!--- Output POP query result set --->
<CFOUTPUT QUERY="p_messages">
    #MESSAGENUMBER# <BR>
    #FROM# <BR>
    #TO# <BR>
    #SUBJECT# <BR>
    #BODY# <BR>
========================<BR>

<!--- Index result set --->
<CFINDEX ACTION="update"
    COLLECTION="pop_query"
    KEY="messagenumber"
    TYPE="custom"
    TITLE="subject"
    QUERY="p_messages"

    BODY="body">

<!--- Search messages for the word "action" --->
<CFSEARCH COLLECTION="pop_query"
    NAME="s_messages"
    CRITERIA="action">

<!--- Output search result set --->
<CFOUTPUT QUERY=" s_messages">
    #Key#, #Title# <BR>
</CFOUTPUT>
```

The CFSEARCH code in the examples above uses the basic attributes needed to search a collection. The next section expands on the capabilities of this tag for more detailed input and output options.

Using Query Expressions

When you search a Verity collection, you use the CFSEARCH tag in a ColdFusion application page. Use the CRITERIA attribute to specify the query expression you want to pass to the search engine.

You can build two types of query expressions: simple and explicit. A simple query expression is typically a word or words. An explicit query expression can employ a number of operators and modifiers to refine the search. Although an explicit query can employ operators and modifiers, all aspects of the search must be explicitly invoked. A simple query expression is somewhat more powerful since it employs operators by default. You can assemble an explicit query expression programmatically or simply

pass a simple query expression to the search engine directly from an HTML input form.

The Verity query language provides many operators and modifiers for composing queries. The following search techniques can be used in searching a Verity collection:

- Word searches
- Proximity searches
- Concept–based
- Field searches in which documents are match based on matching predefined custom attributes
- Scoring operators

Simple query expressions

Simple queries allow end users to enter simple, comma-delimited strings and use wildcard characters. You can enter multiple words separated by commas, in which case the comma is treated like a logical OR. If you omit the commas, the query expression is treated as a phrase.

Ordinarily, operators are employed in explicit query expressions. Operators are normally surrounded by angle brackets < >. However, you can use the AND, OR, and NOT operators in a simple query without using angle brackets.

A simple query employs the STEM operator and the MANY modifier. STEM searches for words that derive from those entered in the query expression, so that entering "find" will return documents that contain "find," "finding," "finds," etc. The MANY modifier forces the documents returned in the search to be presented in a list based on a relevancy score.

Explicit query expressions

Explicit queries can be constructed using a variety of operators, which are described below. Most operators in an explicit query expression are surrounded by angle brackets < >. You can use the AND, OR, and NOT operators without angle brackets.

Expression syntax

You can use either simple or explicit syntax when stating simple query syntax. The syntax you use determines whether the search words you enter will be stemmed, and whether the words that are found will contribute to relevance-ranked scoring.

Simple syntax

When you use simple syntax, the search engine implicitly interprets single words as if they were modified by the MANY and STEM operators. By implicitly applying the MANY modifier, the search engine calculates each document's score based on the

density of the search term in the searched documents. The more frequent the occurrence of a word in a document, the higher the document's score.

As a result, the search engine ranks documents according to word density as it searches for the word you specify, as well as words that have the same stem. For example, "films," "filmed," and "filming" are stemmed variations of the word "film." To search for documents containing the word "film" and its stem words, you can enter the word "film" without modification. When documents are ranked by relevance, they appear in a list with the most relevant documents at the top.

Explicit syntax

When you use explicit syntax, the search engine interprets the search terms you enter as literals. For example, by entering the word "film" (including quotation marks) using explicit syntax, the stemmed versions of the word "film", "films," "filmed," and "filming" are ignored.

The following table shows all operators available for conducting searches of ColdFusion Verity collections.

Verity Search Operators		
<	CONTAINS	PHRASE
<=	ENDS	SENTENCE
=	MATCHES	STARTS
>	NEAR	STEM
>=	NEAR/N	SUBSTRING
Accrue	OR	WILDCARD
AND	PARAGRAPH	WORD

Special characters

A number of characters are handled in particular ways by the search engine.

Special Search Characters	
Characters	**Description**
, () [	These characters end a text token.
= > <!	These characters also end a text token. They are terminated by an associated end character.
' @ ' < {[!	These characters signify the start of a delimited token. They are terminated by an associated end character.

A backslash (\) removes special meaning from whatever character follows it. To enter a literal backslash in a query, use two in succession, such as this examples:

```
<FREETEXT>("\"Hello\", said Packard.")
"backslash (\\)"
```

Composing Search Expressions

The following rules apply for composing search expressions.

Precedence rules

Expressions are read from left to right. The AND operator takes precedence over OR operators. However, terms enclosed in parenteses are evaluated first. When the search engine encounters nested parentheses, it starts with the innermost term:

Prefix and infix notation

Search strings that use any operator other than evidence operators can be defined in prefix notation or infix notation. This means that either of the following expressions is valid:

- `AND (a,b)`

  This is prefix notation

- `a AND b`

  This is infix notation

When prefix notation is used, precedence is handled explicitly within the expression. The following example means: "Look for documents that contain b and c first, then documents that contain a":

```
OR (a, AND (b,c))
```

When infix notation is used, precedence is implicit in the expression. For example, the AND operator takes precedence over the OR operator.

Commas in expressions

If an expression includes two or more search terms within parentheses, a comma is required as a separator between each element. The following example means: Look for documents that contain any combination of a and b together. Note that in this example, angle brackets are used with the OR operator.

```
<OR> (a, b)
```

Delimiters in expressions

Angle brackets < >, double quotation marks " ", and backslashes \ are used to delimit various elements in a query expression.

Angle brackets for operators

Left and right angle brackets < > are reserved for designating operators and modifiers. They are optional for the AND, OR, and NOT operators, but required for all other operators.

Double quotation marks in expressions

You use double quotation marks to search for a word that is otherwise reserved as an operator, such as AND, OR, and NOT.

Backslashes in expressions

To include a backslash \ in a search, insert two backslashes for each backslash character you want to search for, such as `C:\\CFUSION\\BIN`.

Searching with Wildcards

This table shows the wildcard characters for searching Verity collections.

Verity Wildcard Characters	
Wildcard	**Description**
?	Question. Specifies any single alphanumeric character.
*	Asterisk. Specifies zero or more alphanumeric characters. Avoid using the asterisk as the first character in a search string. Asterisk is ignored in a set, [] or an alternative pattern {}.
[]	Square brackets. Specifies one of any character in a set, as in "sl[iau]m" which locates "slim," "slam," and "slum." Square brackets indicate an implied OR.
{}	Curly braces. Specifies one of each pattern separated by a comma, as in "hoist{s, ing, ed}" which locates "hoists," "hoisting," and "hoisted." Curly braces indicate an implied AND.
^	Caret. Specifies one of any character not in the set as in "sl[^ia]m" which locates "slum" but not "slim" or "slam."
-	Hyphen. Specifies a range of characters in a set as in "c[a-r]t" which locates every word beginning with "c," ending with "t," and containing any letter from "a" to "r."

Searching for wildcards as literals

To search for a wildcard character in your collection, you need to escape the character with a backslash (\). For example:

To match a literal asterisk, you precede the * with two backslashes: "a*"

To match a question mark or other wildcard character: "Checkers\?"

Searching for special characters as literals

The following non-alphanumeric characters must be preceded by a backslash character (\) in a search string:

- comma (,)
- left and right parentheses ()
- double quotation mark (")
- backslash (\)
- at sign (@)

- left curly brace ({)
- left bracket ([)
- less than sign (<)
- backquote (‘)

In addition to the backslash character, you can use paired backquotes (‘ ‘) to interpret special characters as literals. For example, to search for the wildcard string "a{b" you can surround the string with backquotes, as follows:

```
‘a{b‘
```

To search for a wildcard string that includes the literal backquote character (‘) you must use two backquotes together and surround the whole string in backquotes:

```
‘*n‘‘t‘
```

Note that you can use either paired backquotes or backslashes to escape special characters. There is no functional difference in the use of one or the other. For example, you can query for the term: <DDA> in the following ways:

```
\<DDA\> or ‘<DDA>‘
```

Operators and Modifiers

The power of the CFSEARCH tag is in the control it gives you over the Verity search engine. The engine offers users a high degree of specificity in setting search parameters.

Operators

An operator represents logic to be applied to a search element. This logic defines the qualifications a document must meet to be retrieved. Operators are used to refine your search or to influence the results in other ways. For example, you could construct an HTML form for conducting searches. In the form, a user could perform a search for a single term: server. You can refine your search by limiting the search scope in a number of ways. Operators are available for limiting a query to a sentence or paragraph, and you can search words based on proximity. The following operator types are available:

- **Evidence operators** — Used to specify basic and intelligent word searches.
- **Proximity operators** — For specifying the relative location of words in a document.
- **Relational operators** — Search fields in a collection.
- **Concept operators** — Used to identify a concept in a document by combining the meanings of search elements.
- **Score operators** — Allow you to manipulate the score returned by a search element. The score percentage display can optionally be set to as many as four decimal places.

- **Natural language operators** — Allow the use of natural language expressions in forming queries.

Ordinarily, you use operators in explicit searches. They are used in the following manner:

```
"<operator>search_string"
```

Evidence operators

Evidence operators can be used to specify either a basic word search or an intelligent word search. A basic word search finds documents that contain only the word or words specified in the query. An intelligent word search expands the query terms to create an expanded word list so that the search returns documents that contain variations of the query terms.

Documents retrieved using evidence operators are not ranked by relevance unless you use the MANY modifier.

Verity Evidence Operators	
Operator	**Description**
STEM	Expands the search to include the word you enter and its variations. The STEM operator is automatically implied in any SIMPLE query. For example, the EXPLICIT query expression `<STEM>believe` yields matches such as, "believe," "believing," "believer".'
WILDCARD	Matches wildcard characters included in search strings. Certain characters automatically indicate a wildcard specification, such as * and?. For example, the query expression `spam*` yields matches such as, "spam," "spammer," "spamming."
WORD	Performs a basic word search, selecting documents that include one or more instances of the specific word you enter. The WORD operator is automatically implied in any SIMPLE query.

Proximity operators

Proximity operators specify the relative location of specific words in the document. Specified words must be in the same phrase, paragraph, or sentence for a document to be retrieved. In the case of NEAR and NEAR/N operators, retrieved documents are ranked by relevance based on the proximity of the specified words. Proximity operators can be nested; phrases or words can appear within SENTENCE or PARAGRAPH operators, and SENTENCE operators can appear within PARAGRAPH operators.

The following table describes each operator.

Verity Proximity Operators	
Operator	**Description**
NEAR	Selects documents containing specified search terms. The closer the search terms are to one another within a document, the higher the document's score. The document with the smallest possible region containing all search terms always receives the highest score. Documents whose search terms are not within 1000 words of each other are not selected.

NEAR/*N*	Selects documents containing two or more search terms within *N* number of words of each other, where *N* is an integer between 1 and 1024 where NEAR/1 searches for two words that are next to each other. The closer the search terms are within a document, the higher the document's score.
	You can specify multiple search terms using multiple instances of NEAR/*N* as long as the value of N is the same:
	`commute <NEAR/10> bicycle <NEAR/10>`
	`train <NEAR/10>`

PARAGRAPH	Selects documents that include all of the words you specify within the same paragraph. To search for three or more words or phrases, you must use the PARAGRAPH operator between each word or phrase.

PHRASE	Selects documents that include a phrase you specify. A phrase is a grouping of two or more words that occur in a specific order. Examples of phrases:
	`mission oak`
	`"mission oak"`
	`mission <PHRASE> oak`
	`<PARAGRAPH> (mission, oak)`

SENTENCE	Selects documents that include all of the words you specify within the same sentence. Examples:
	`jazz <SENTENCE> musician`
	`<SENTENCE> (jazz, musician)`

Relational operators

Relational operators search document fields that have been defined in the collection. Documents containing specified field values are returned. Documents retrieved using relational operators are not ranked by relevance, and you cannot use the MANY modifier with relational operators.

The following operators are used for numeric and date comparisons.

Verity Numerical and Date Relational Operators	
Operator	**Description**
=	Equals
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Text comparison operators match words and parts of words. The following operators are used for text comparisons.

VerityText Comparison Operators	
Operator	**Description**
CONTAINS	Selects documents by matching the word or phrase you specify with the values stored in a specific document field. Documents are selected only if the search elements specified appear in the same sequential and contiguous order in the field value. For example, specifying "god" will match "God in heaven," "a god among men," or "good god" but not "godliness," or "gods."
MATCHES	Selects documents by matching the query string with values stored in a specific document field. Documents are selected only if the search elements specified match the field value exactly. If a partial match is found, a document is not selected. For example, specifying "god" will match a document field containing only "god" and will not match "gods," "godliness," or "a god among men."
STARTS	Selects documents by matching the character string you specify with the starting characters of the values stored in a specific document field.

VerityText Comparison Operators (Continued)	
Operator	**Description**
ENDS	Selects documents by matching the character string you specify with the ending characters of the values stored in a specific document field.
SUBSTRING	Selects documents by matching the query string you specify with any portion of the strings in a specific document field. For example, specifying "god" will match "godliness," "a god among men," "godforsaken," etc.

Document fields

The values you specify for the CFINDEX attributes TITLE, KEY, URL, and CUSTOM can be specified as document fields for use with relational operators in the CRITERIA attribute. Document fields are referenced in text comparison operators. They are identified as:

- CF_TITLE
- CF_KEY
- CF_URL
- CF_CUSTOM1
- CF_CUSTOM2

For more information on this topic, see the Allaire Knowledge Base article, "Using Document Fields To Narrow Down Searches" (ID# 1082) on our Web site at http://www.allaire.com/Support/KnowledgeBase/SearchForm.cfm.

The SUBSTRING operator

You can use the SUBSTRING operator to match a character string with data stored in a specified data source. In the following example, a data source called TEST1 contains the table YearPlaceText, which itself contains three columns: Year, Place, and Text. Year and Place make up the primary key. The following table shows the TEST1 schema.

YearPlaceText		
Year	**Place**	**Text**
1990	Utah	Text about Utah 1990
1990	Oregon	Text about Oregon 1990
1991	Utah	Text about Utah 1991

YearPlaceText (Continued)		
Year	**Place**	**Text**
1991	Oregon	Text about Oregon 1991
1992	Utah	Text about Utah 1992

The following application page matches records that have 1990 in the TEXT column and are in the Place Utah. The search is performed against the collection that contains the TEXT column and then is narrowed further by searching the string "Utah" in the CF_TITLE document field. Recall that document fields are defaults defined in every collection corresponding to the values you define for URL, TITLE, and KEY in the CFINDEX tag.

```
<CFQUERY NAME="GetText"
    DATASOURCE="TEST1">
    SELECT Year+Place
        AS Identifier, text
        FROM YearPlaceText
</CFQUERY>

<CFINDEX COLLECTION="testcollection"
    ACTION="Update"
    TYPE="Custom"
    TITLE="Identifier"
    KEY="Identifier"
    BODY="TEXT"
    QUERY="GetText">

<CFSEARCH NAME="GetText_Search"
    COLLECTION="testcollection"
    TYPE="Explicit"
    CRITERIA="1990 and CF_TITLE <SUBSTRING> Utah">
<CFOUTPUT>
    Record Counts: <BR>
    #GetText.RecordCount# <BR>
    #GetText_Search.RecordCount# <BR>
</CFOUTPUT>

<CFOUTPUT>
    Query Results --- Should be 5 rows <BR>
</CFOUTPUT>

<CFOUTPUT QUERY="Gettext">
    #Identifier# <BR>
</CFOUTPUT>

<CFOUTPUT>
    Search Results -- should be 1 row <BR>
```

```
</CFOUTPUT>

<CFOUTPUT QUERY="GetText_Search">
    #GetText_Search.TITLE# <BR>
</CFOUTPUT>
```

Concept operators

Concept operators combine the meaning of search elements to identify a concept in a document. Documents retrieved using concept operators are ranked by relevance. The following table describes each concept operator.

Verity Concept Operators	
Operator	**Description**
AND	Selects documents that contain all of the search elements you specify.
OR	Selects documents that show evidence of at least one of the search elements you specify.
ACCRUE	Selects documents that include at least one of the search elements you specify. Documents are ranked based on the number of search elements found.

Score operators

Score operators govern how the search engine calculates scores for retrieved documents. The maximum score a returned search element can have is 1.000. The score percentage display can optionally be set to as many as four decimal places.

When a score operator is used, the search engine first calculates a separate score for each search element found in a document, and then performs a mathematical operation on the individual element scores to arrive at the final score for each document.

Note that the document's score is available as a result column. The SCORE result column can be referenced to trap the relevancy score of any document retrieved. For example:

```
<CFOUTPUT>
    <A HREF="#Search1.URL#">#Search1.Title#</A><BR>
    Document Score=#Search1.SCORE#<BR>
</CFOUTPUT>
```

The following table lists the score operators.

Verity Score Operators	
Operator	**Description**
YESNO	Forces the score of an element to 1 if the element's score is non-zero: `<YESNO>mainframe` If the retrieval result of the search on "mainframe" is 0.75, the YESNO operator forces the result to 1. You can use YESNO to avoid relevance ranking.
PRODUCT	Multiplies the scores for documents matching a query. To arrive at a document's score, the search engine calculates a score for each search element and multiplies these scores together: `<PRODUCT>(computers, laptops)` The resulting score for each document is multiplied together.
SUM	Adds together the scores for documents matching a query, up to a maximum value of 1: `<SUM>(computers, laptops)` The resulting scores are added together.
COMPLEMENT	Calculates scores for documents matching a query by taking the complement (subtracting from 1) of the scores for the query's search elements. The new score is 1 minus the search element's original score. `<COMPLEMENT>computers` If the search element's original score is.785, the COMPLEMENT operator recalculates the score as.215.

Modifiers

Modifiers can be used with operators to further refine query expressions. You can specify case sensitivity in a query, or force the output to be ranked by relevancy. Modifiers include:

- CASE — Sets case sensitivity. Verity searches are case-insensitive for search text entered in all uppercase or all lowercase. Case sensitivity is turned on when mixed case characters are entered.
- MANY — Results are ranked by relevancy, which is determined by the number of times the search value is found in a document.
- NOT — Eliminates documents containing the specified words.
- ORDER — Returns documents only if they contain words in the listed order.

Search modifiers

Modifiers are combined with operators to change the standard behavior of an operator in some way. For example, you can use the CASE modifier with an operator to specify that you want to match the case of the search word.

Modifiers are as follows.

Verity Search Modifiers	
Modifier	**Description**
CASE	Specifies a case-sensitive search: <CASE>J[JAVA, java] Searches for "JAVA" and "Java." If a search contains a mixed-case string, the search request will be case-sensitive.
MANY	Counts the density of words, stemmed variations, or phrases in a document and produces a relevance-ranked score for retrieved documents. Can be used with the following operators: WORD WILDCARD STEM PHRASE SENTENCE PARAGRAPH <PARAGRAPH><MANY>javascript <AND> vbscript The MANY modifier cannot be used with the following: AND OR ACCRUE Relational operators

Verity Search Modifiers (Continued)	
Modifier	**Description**
NOT	Used to exclude documents that contain the specified word or phrase. Used only with the AND and OR operators. `Java <AND> programming <NOT> coffee`
ORDER	Used to specify that the search elements must occur on the same order in which they were specified in the query. Can be used with the following operators: PARAGRAPH SENTENCE NEAR/*N* Place the ORDER modifier before any operator: `<ORDER><PARAGRAPH>("server", "Java")`

Managing Collections

As with any data source, the maintenance requirements of a Verity collection are dictated by the amount, frequency, and type of changes that occur in the records. You can run maintenance routines directly from either the CFCOLLECTION or CFINDEX tags or via the Administrator Verity page. For more information on this topic, see the Allaire Knowledge Base article, "Maintaining Collections" (ID# 1080) on our Web site at http://www.allaire.com/Support/KnowledgeBase/SearchForm.cfm.

The easiest way to perform collection management tasks is to create a ColdFusion template that runs the operations, then add the task on the Administrator Scheduler page. The page presents a wide range of scheduling options.

Maintenance options

Choose an option based on the following function descriptions.

- **Repair** — Runs internal Verity routines to fix corrupted records. If you suspect a collection has become corrupted, it is probably safest to re-populate it.
- **Optimize** — Packs the indexed data for better performance. This is similar to database optimization. This procedure can be used as part of routine maintenance. The Optimize action is deprecated for CFINDEX except to maintain legacy code; the CFCOLLECTION tag is recommended instead. For more information on this command, see the Allaire Knowledge Base article, "How To Optimize Your Verity Collection" (ID# 416) on our Web site at http://www.allaire.com/Support/KnowledgeBase/SearchForm.cfm.
- **Purge** — Removes all data from a collection.

- **Delete** (when used as a CFINDEX ACTION) — Deletes the specified KEY value, or comma-separated values, from the collection.
- **Delete** (when used on the Administrator Verity page or in CFCOLLECTION) — Deletes the entire collection.
- **Update** — Re-populates the collection with changed records and new records and adds a key if one is not part of the collection. This operation does not delete records that have been deleted from the data source. To update a collection from the Administrator Verity main page, select a collection on the list, click Index, then click Update on the index page.
- **Refresh** (CFINDEX ACTION only) — Deletes all data and re-populates the collection.

Securing a collection

A couple of possible scenarios for restricting access to a Verity collection are:

- The ColdFusion Administrator may need to specify developer access to collections.
- A public site may need to limit user access to collections.

To restrict access to a collection, follow these steps:

1. Open the Advanced Server Security page of the ColdFusion Administrator and click the Use Advanced Server Security box.
2. Click the Security Contexts button.
3. Enter a name for the secured collection and click Add.
4. Optionally enter a description for the secured collection.
5. Click Collections on the Enable Security for Resource Types list.
6. Click Apply.

You can then develop an appropriate authentication interface to allow access to the secured collection.

CHAPTER 12

Using the Application Framework

The ColdFusion Web Application Framework is a powerful tool you can use to help structure your ColdFusion applications. This section describes how to create and use the `Application.cfm` file, the application page that controls the application framework.

Contents

Understanding the Web Application Framework

A ColdFusion application is a collection of application pages that work together. Applications can be as simple as a guest book or as sophisticated as a full Internet commerce system with catalog pages, shopping carts, and reporting. You can combine individual applications to create advanced Web systems.

You create a special template, named `Application.cfm`, which you place in the root directory of the application. All the other templates in the application are stored in directories below the application's root directory.

The ColdFusion Web Application Framework is based on four basic components:

- Application-level settings and functions
- Client state management
- Custom error handling
- Web server security integration

With these components, you can easily combine your ColdFusion application pages into sophisticated Web applications.

Application-level settings and functions in Application.cfm

ColdFusion offers application-level features that help you control settings, variables, and features available across the entire application. Once you have defined an application, you can use the application-level features in addition to all of the other features in ColdFusion.

Client state management

Because the Web is a stateless system, each connection a browser makes to a Web server is unique in the eyes of the Web server. However, within an application it is important to be able to keep track of users as they move through the pages within the application. This is the definition of client state management.

An application maintains client state by seamlessly tracking variables for a browser as the user moves from page to page within the application. This can be used in place of other methods for tracking client state such as using URL parameters, hidden form fields, and HTTP cookies.

ColdFusion creates a client record for each browser that requests an application page in an application in which client state management is enabled. The client record is identified by a unique token that is stored in an HTTP cookie in the user's browser.

The application can then define variables within the client record. These client variables are accessible as parameters in every application page that the client requests within the scope of the application.

Custom error handling

Using the CFERROR tag, you can display customized HTML pages when errors occur. This allows you to maintain a consistent look and feel within your application even when errors occur. It also allows you to optionally suppress the display of error information.

See "Generating Custom Error Messages (CFERROR)" on page 93 for more information.

Web server security integration

You can integrate your applications with the user authentication and security provided by your Web server. In addition, the ColdFusion Server offers a security framework that controls access to applications, pages, data sources, and users. You set the bounds of a security domain using the CFAUTHENTICATE tag.

See Chapter 17, "Application Security," on page 263 for more information.

Mapping Out an Application Framework

An important step in designing a ColdFusion application is mapping out its directory structure.

Before you start building the application, establish a root directory for the application. Application pages may be stored in subdirectories of the root directory.

When any ColdFusion application page is requested, ColdFusion searches up the page's directory tree for an `Application.cfm` file. When it is found, the `Application.cfm` code is logically included at the beginning of that page.

If it is not found, ColdFusion searches up the directory tree until it finds an `Application.cfm` file. If more than one `Application.cfm` file lives in the current directory tree, ColdFusion uses the first one it finds.

Just as the `Application.cfm` file is executed before each application page it governs, you can specify a file named `OnRequestEnd.cfm`, which is executed after each application page in the same application.

ColdFusion Server looks for the `OnRequestEnd.cfm` file in the same directory as the `Application.cfm` file of the current application page. The `OnRequestEnd.cfm` file will never be executed if it resides in another directory.

The `OnRequestEnd.cfm` file will not be executed if there is an error or an exception in the called page, or if the called page executes the CFABORT or CFEXIT tag.

Just as the `Application.cfm` file must be spelled with a capital A, you must spell the `OnRequestEnd.cfm` file with capital O, R, and E.

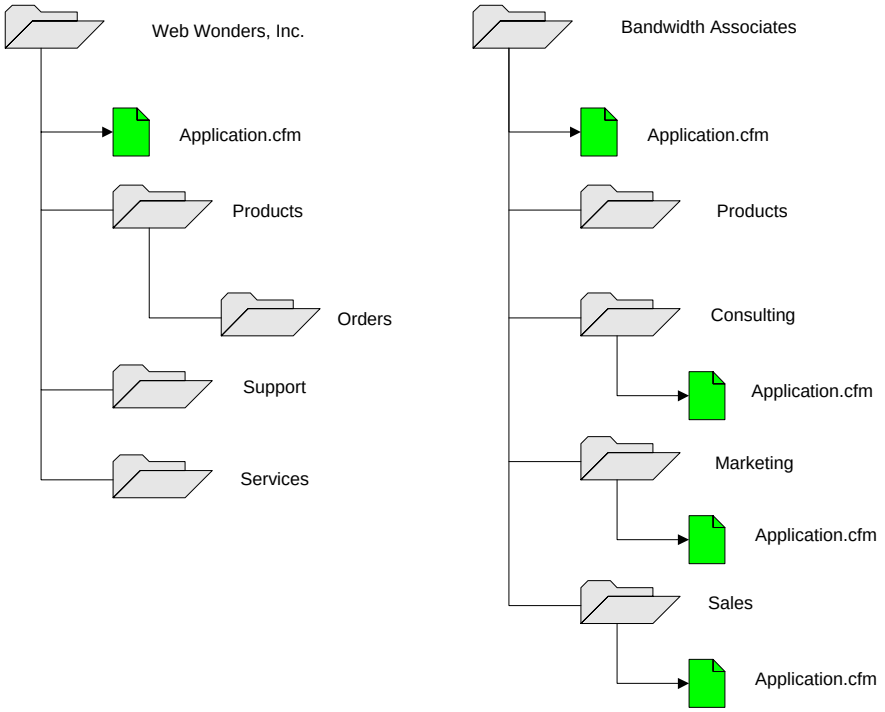
Defining a root directory for an application has a number of advantages:

- Development: The application is easier to develop and maintain because the application page files are well organized.
- Portability: The application can be more easily moved to another server or another part of a server without having to change any code in the application page files.
- Application-level Settings: Application pages that fall under the same root directory can share application-level settings and functions.
- Security: Application pages that fall under the same directory can share Web server security settings.

You can use a single `Application.cfm` file for your application, or use different `Application.cfm` files that govern individual sections of the application.

The directory trees below illustrate two approaches to implementing the Application Framework.

- In the first example, a company named Web Wonders, Inc. uses a single `Application.cfm` file installed in their application root directory to process all application page requests.
- The illustration on the right shows how Bandwidth Associates uses the settings in individual `Application.cfm` files to specify processing for ColdFusion applications at the departmental level. Only the Products application pages are processed using the settings in the root `Application.cfm` file. The Consulting, Marketing, and Sales directories each has its own `Application.cfm` file.



Behavior with CFINCLUDE

Only one `Application.cfm` file is ever processed for each ColdFusion application page. The presence of an `Application.cfm` file is an implicit CFINCLUDE. If it is present in the directory tree, there is no way not to include it. For this reason, it is the ideal location to set application-level variables.

When the requested application page has a CFINCLUDE tag pointing to an additional application page, ColdFusion does not initiate another search up the directory tree based on the included application page. This is an important behavior to understand. Upon opening a requested application page, ColdFusion searches for the `Application.cfm` file only once.

Creating the Application.cfm File

The special application-wide page called `Application.cfm` defines application-level settings and functions such as:

- The application name
- Client state management options
- Application and session variables

- Default variables
- Custom error pages
- Data sources
- Default style settings
- Exclusive locks
- Other application-level constants

Note Because UNIX is case sensitive, the application framework file must be spelled with an initial capital, `Application.cfm`, for applications that run on UNIX platforms.

Naming the application

In ColdFusion, you define an application by giving it a name using the CFAPPLICATION tag. By using the same application name in a CFAPPLICATION tag, you define a set of pages as being part of the same logical application.

Note The value you set for the NAME attribute in CFAPPLICATION is limited to 64 characters.

To name the application:

1. Open Studio and create a new file.
2. Modify the file so that it appears as follows:

```
<!--- This example illustrates CFAPPLICATION --->

<!--- Name the application --->
<CFAPPLICATION NAME="SearchApp">
```

3. Save the file as `Application.cfm` in the root directory of your application framework.

Setting up client state management options

If you want to enable client state management, you must do so on every page in an application. Because the `Application.cfm` file is included in all of the application's pages, you enable client management in the CFAPPLICATION tag, at the beginning of `Application.cfm`.

To enable client state management:

1. Open the file `Application.cfm` in Studio and modify it so that it appears as follows:

```
<!--- This example illustrates CFAPPLICATION --->

<!--- Name the application and enable client management--->
<CFAPPLICATION NAME="SearchApp"
CLIENTMANAGEMENT="Yes">
```

2. Save the file as `Application.cfm` in the root directory of your application framework.

Choosing a client variable storage method

Once you have enabled client state management, you then have to determine where you want to store client variables. The system-wide default is for client variables to be stored in the system registry. But your site administrator can choose to store them instead in a SQL database or in cookies.

There are two steps to change client variable storage: first, setting a client variable storage option in the Variables page of the ColdFusion Administrator, and then, noting the client variable storage location in the CFAPPLICATION tag. See *Administering ColdFusion Server* for information on using the ColdFusion Administrator.

You use the CLIENTSTORAGE attribute in the CFAPPLICATION tag to specify where you want to store client variables, providing one of three values:

- `Registry`
- The name of a configured client store
- `Cookie`

If no ClientStorage setting is specified, the default location, as noted in the ColdFusion Administrator Variables page, is used.

The following example shows how you enable client state management using a sample database called *mydatasource*.

To note the client variable storage method:

1. Open the file `Application.cfm` in Studio and modify it so that it appears as follows:

```
<!--- This example illustrates CFAPPLICATION --->

<!--- Name the application and enable client management--->
<CFAPPLICATION NAME="SearchApp"
CLIENTMANAGEMENT="Yes"
CLIENTSTORAGE="mydatasource">
```

2. Save the file as `Application.cfm` in the root directory of your application framework.

Note Client storage mechanisms are exclusive; when one storage type is in use, the values set in other storage options are unavailable.

Cookie storage

When you set `CLIENTSTORAGE="Cookie"` the cookie that ColdFusion creates has the application's name. Storing client data in a cookie is scalable to large numbers of clients, but this storage mechanism has some limitations. Chief among them is that if the client turns off cookies in the browser, client variables won't work.

Consider these additional limitations before implementing cookie storage for client variables:

- Netscape Navigator allows only 20 cookies from a particular host to be set. ColdFusion uses two of these cookies for CFID and CFTOKEN, and also creates a cookie named CFGLOBALS to hold global data about the client, such as HitCount, TimeCreated, and LastVisit. This limits you to 17 unique applications per host.
- Netscape Navigator sets a size limit of 4K bytes per cookie. ColdFusion encodes non-alphanumeric data in cookies with a URL encoding scheme that expands at a 3-1 ratio, which means you should not store large amounts of data per client. ColdFusion will throw an error if you try to store more than 4000 encoded bytes of data for a client.

Managing Client State in a Clustered Environment

To maintain your ColdFusion Web application's state in a clustered environment, you can use server-side client variables that get stored in a common, back-end repository that all Web servers in a multi-server clustered environment can access. Even though all state information will be stored in client variables in the repository, a mechanism must be used to identify specific client requests. This is typically accomplished by dropping a client-side identifier, such as a cookie, on the user's machine.

ColdFusion 4.5 provides several client variable attributes in the CFApplication tag that allow you to maintain application state across a cluster when using server-side client variables. These attributes enable client variable management and set CFID and CFTOKEN cookies at the domain level (for example, .allaire.com). If ID and token combinations already exist on each host in the cluster, ColdFusion migrates the host-level cookies on each cluster member to the single, common domain-level cookie. Following the setting or migration of host-level cookies to a domain-level cookie, ColdFusion creates a new cookie (CFMAGIC) that tells ColdFusion that domain cookies have been set.

This domain-level cookie allows a ColdFusion application to maintain specific client information across a server cluster.

Using Client State Management

When client state management is enabled for an application, you can use the system to keep track of any number of variables associated with a particular client.

Creating a client variable

To create a client variable and set the value of the parameter, use the CFSET or CFPARAM tag., for example:

```
<CFSET Client.FavoriteColor="Red">
```

Once a client variable has been set in this manner, it is available for use within any application page in your application that is accessed by the client for whom the variable is set.

The following example shows how to use the CFPARAM tag to check for the existence of a client parameter and to set a default value if the parameter does not already exist:

```
<CFPARAM NAME="Client.FavoriteColor" DEFAULT="Red">
```

Using Client Variables

A client variable is accessed using the same syntax as other types of variables, and can be used anywhere other ColdFusion variables are used.

To display the favorite color that has been set for a specific user, use the following code:

```
<CFOUTPUT>
    Your favorite color is #Client.FavoriteColor#.
</CFOUTPUT>
```

Standard client variables

In addition to storing custom client variables, the Client object has several standard parameters. These parameters can be useful in providing customized behavior depending on how often users visit your site and when they last visited. For example, the following code shows the date of a user's last visit to your site:

```
<CFOUTPUT>
    Welcome back to the Web SuperShop. Your last
    visit was on #DateFormat(Client.LastVisit)#.
</CFOUTPUT>
```

The standard Client object attributes are read-only (they can be accessed but not set by your application) and include CFID, CFToken, URLToken, HitCount, TimeCreated, and LastVisit.

Using client state management without cookies

You can use ColdFusion's client state management without cookies. However,this is not recommended. If you choose to maintain client state without cookies, you must ensure that every request carries CFID and CFTOKEN.

To maintain client state without cookies, set the SETCLIENTCOOKIES attribute of the CFAPPLICATION tag to No. Then, you must maintain client state in URLs. by passing

the client ID (CFID)and the client security token (CFTOKEN) between pages, either in hidden form fields or appended to URLs. You accomplish this using the variable Client.URLTOKEN or Session.URLTOKEN.

Note In ColdFusion, client state management is explicitly designed to work with cookies, the standard tool for identifying clients. Using client state management without cookies requires careful programming to ensure that the URLToken is always passed between application pages.

Getting a list of client variables

To obtain a list of the custom client parameters associated with a particular client, use the GetClientVariablesList function.

```
<CFOUTPUT>#GetClientVariablesList()#</CFOUTPUT>
```

The GetClientVariablesList function returns a comma-separated list of variable names defined for the application context declared by CFAPPLICATION, if any. The standard system-provided client variables (CFID, CFToken, URLToken, HitCount, TimeCreated, and LastVisit) are not returned in the list.

Deleting client variables

Unlike normal variables, client variables and their values persist over time. (In this fashion they are akin to cookies.) To delete a client variable, use the DeleteClientVariable function. For example:

```
<CFSET IsDeleteSuccessful=DeleteClientVariable("MyClientVariable")>
```

The DeleteClientVariable function operates only on variables within the scope declared by CFAPPLICATION, if any. See the *CFML Language Reference* for more information on this function.

Also, through the Variables page of the ColdFusion Administrator, you can edit the client variable storage to remove client variables after a set number of days. (The default value is 90 days when client variables are stored in the registry, ten days when stored in a data source.)

See *Administering ColdFusion Server* for more information about setting time-out values.

Note The system-provided client variables (CFID, CFToken, URLToken, HitCount, TimeCreated, and LastVisit) cannot be deleted.

Client variables with CFLOCATION behavior

When using CFLOCATION to redirect to a path that contains .DBM or .CFM, the Client.URLToken is automatically appended to the URL. This behavior can be suppressed by adding the attribute ADDTOKEN="No" to the CFLOCATION tag.

Variable caching

All client variable reads and writes are cached to help decrease the overhead of client state management operations. See *Administering ColdFusion Server* for information on variables and server clustering.

Exporting the client variable database

If your client variable database is stored in the system registry and you need to move it to another machine, you can export the registry key that stores your client variables and take it to your new server. The system registry allows you to export and import registry entries.

To export your client variable database from the registry:

1. Open the registry editor. In UNIX, use the program, `/<install_dir>/coldfusion/bin/regedit`.
2. Find and select the following key:
   `HKEY_LOCAL_MACHINE\SOFTWARE\Allaire\ColdFusion\CurrentVersion\Clients`
3. On the Registry menu, click Export Registry File.
4. Enter a name for the registry file.

Once you've created a registry file, you can take it to a new machine and import it by selecting Import Registry File on the Registry Editor Registry menu.

Application and Session Variables

In ColdFusion, you use variables to work around the Web's inherent statelessness. Session and application variables are persistent variable "scopes." You access these variables by prefacing the variable name with the scope name, for example: *Session.MyVariable* or *Application.MyVariable*. And because they are persistent, you can pass values between pages with a minimum of effort.

Enabling application and session variables

Session and application variables are similar in operation to client variables. Like client variables, they are enabled with the CFAPPLICATION tag. However, unlike client variables, which are stored in the system registry, a data source, or a cookie, application and session variables are always stored in the ColdFusion server's memory. This method offers obvious performance advantages. In addition, you can set time-out values for these variables either with CFAPPLICATION, or by specifying time-outs in the ColdFusion Administrator. You can also simply disable application and session variables entirely.

For information on setting time-outs for variables, See *Administering ColdFusion Server*.

Differentiating client, session, and application variables

This table shows the relationships among client, session, and application variables.:

Kinds of Variables					
Variable Type	**Application Names**	**Client IDs**	**Client Mgmt**	**Session Mgmt**	**Time-out**
Client	Optional	Required	Required	n/a	Optional
Session	Optional	Required	Required	Required	Optional
Application	Required	n/a	n/a	n/a	Optional

Note ColdFusion does not attempt to automatically evaluate application and session variables. You must use variable prefixes with these variables, as in Session.variablename or Application.variablename.

Using Session Variables

Use session variables when the variables are needed for a single site visit or set of requests. For example, you might use session variables to store a user's selections in a shopping cart application. (Use client variables when the variable is needed for future visits.)

What is a session?

A session refers to all the connections that a *single client* might make to a server in the course of viewing any pages associated with a given application. Sessions are specific to individual users. As a result, every user has a separate session and has access to a separate set of session variables.

This logical view of a session begins with the first connection by a client and ends (after a specified time-out period) after that client's last connection. However, because of the stateless nature of the Web, it's not always possible to define a precise point at which a session ends. In the real world, a session ends when the user finishes using an application and goes off to do something else. In most cases, however, a Web application has no way of knowing if a user is finished or if he's just lingering over a page.

You can impose some structure on session variable duration by specifying a time-out period. If the user does not access a page of the application within this time-out

period, ColdFusion interprets this as the end of the session and clears any variables associated with that session.

The default time-out for session variables is set to 20 minutes. In the Variables page of the ColdFusion Administrator, you can change this time-out value. See *Administering ColdFusion Server* for more information.

You can also set the time-out period for session variables inside a specific application (thereby overruling the Administrator default setting) by using the SESSIONTIMEOUT attribute of the CFAPPLICATION tag.

Storing session data in session variables

Session variables are designed to store session-level data. They are a convenient place to store information that all pages of your application might need during a user session. Using session variables, an application could initialize itself with user-specific data the first time a user hit a page of that application. This information could then remain available while that user continues to use that application. For example, information about a specific user's preferences could be retrieved from a database once, the first time a user hits any page of an application. This information would remain available throughout that user's session, thereby avoiding the overhead of retrieving the preferences again and again.

Session variables work exactly as client variables do, in that they require a client name (client ID) and are always scoped within that client ID. Session variables also work within the scope of an application name if one is supplied, in which case their scope will be the combination of the client ID and the application name.

To enable session variables, set `SESSIONMANAGEMENT="Yes"` in the CFAPPLICATION tag in your `Application.cfm` file. Note that when you turn on session management in the CFAPPLICATION tag, you must specify the application's name. Following is an example of turning on session management:

```
<!--- This example illustrates CFAPPLICATION --->

<!--- Name the application, and turn on
    session management --->
<CFAPPLICATION NAME="GetLeadApp" SESSIONMANAGEMENT="Yes">

<!--- set data source for this application --->
<CFSET dsn = "my_dsn">

<!--- set global error handling for this application --->
<CFERROR TYPE="REQUEST" TEMPLATE="request_err.cfm"
    MAILTO="webmaster@mysite.com">
<CFERROR TYPE="VALIDATION" TEMPLATE="val_err.cfm"
    MAILTO="webmaster@mysite.com">

<!--- set some global variables for this application
```

```
to be triggered on every page --->
<CFSET MainPage = "default.cfm">
<CFSET session.current_location = "Davis, Porter, Alewife">
<CFSET sm_location = "dpa">
<CFSET current_page = "#cgi.path_info#?#cgi.query_string#">
```

Using Application Variables

Application variables require an application name be associated with them and are always scoped within that application name.

Unlike client and session variables, however, application variables do not require that a client name (client ID) be associated with them. Thus, they are available to any clients that specify the same application name.

The name you establish in the CFAPPLICATION tag is accessible elsewhere in the application by using the Application.ApplicationName variable. For example, you would use this variable in the CFLOCK tag to restrict access to application variables to one request at a time.

Storing application data in application variables

Application variables are designed to store application-level data. They are a convenient place to store information that all pages of your application might need no matter who (what client) is running that application. Using application variables, an application could initialize itself, say, when the first user hit any page of that application. This information could then remain available indefinitely to all subsequent hits of any pages of that application, by all users, thereby avoiding the overhead of repeated initialization.

Because the data stored in application variables is available to all pages of an application and remains available until ColdFusion Server is shut down, application variables are very convenient. However, because all clients running an application see the same set of application variables, they are not useful for client-specific information. To target variables for specific clients, use session variables.

Application variable time-outs

Application variables have a specific lifetime, and this lifetime defines an "application." For example, when you access an application variable inside a specific application, the variable returns a value because your request occurs on a page declared in the CFAPPLICATION tag to be part of a single application.

The default time-out period for application variables is two days. In the Variables page of the ColdFusion Administrator, you can define time-out values for application and session variables. See *Administering ColdFusion Server* for more information.

You can set the time-out period for application variables within a specific application (thereby overriding the default setting in the ColdFusion Administrator) by using the APPLICATIONTIMEOUT attribute of the CFAPPLICATION tag.

If no clients access the application within the specified time-out period, ColdFusion Server destroys its application variables.

Tips for Using Session and Application Variables

In general, session and application variables are designed to hold information that you seldom write but read often. In most cases, the values of these variables are set once, most often when an application is first started (Application variables) or the first time a user begins using an application (Session variables). Then the values of these variables will be referenced many times throughout the life of the application or the course of a session.

When using application variables, keep in mind that these variables are shared by all instances of an application that might be running on a server. Because of this sharing, applications cannot assume that values saved in these variables will not be overwritten by other instances of the same application that might be simultaneously running on the server. Of course, this is not a problem if these variables are treated as "write-once, read-many," but can be a problem if they are written to indiscriminately.

Getting a list of application and session variables

The variable scope names "application" and "session" are registered as ColdFusion structures. This enables you to use the ColdFusion Structure functions to get a list of application and session variables. For example, you can use CFLOOP with the StructFind function to output a list of application and session variables defined for a specific application.

To find a list of client variables, you use the GetClientList function.

See the *CFML Language Reference* for more information on these functions.

Default Variables and Constants

It is often useful to set default variables and application-level constants in the `Application.cfm` file. For example you may want to designate:

- A data source you're using
- A domain name
- Style settings such as fonts or colors
- Other important application-level variables

Example: Application.cfm

The following example shows a complete `Application.cfm` file for the sample Products application:

```
<!--- Set application name and enable client
variables option, with client variables stored in
a data source called mycompany --->

<CFAPPLICATION NAME="Products"
    CLIENTMANAGEMENT="Yes"
    CLIENTSTORAGE="mycompany">

<!--- Install custom error pages --->

<CFERROR TYPE="REQUEST"
    TEMPLATE="requesterr.cfm"
    MAILTO="admin@company.com">

<CFERROR TYPE="VALIDATION"
    TEMPLATE="validationerr.cfm">

<!--- Set application constants --->

<CFSET HomePage="http://www.mycompany.com">
<CFSET PrimaryDataSource="CompanyDB">
```

Using CFLOCK for Exclusive Locking

The CFLOCK tag provides a means of implementing exclusive locking in ColdFusion applications. The reasons you use CFLOCK include :

- Protecting sections of code that manipulate shared data, such as session, application, and server variables.
- Ensuring that file updates do not fail because files are open for writing by other applications or ColdFusion tags.

Note Use anonymous locks to protect a portion of a template, for example a non-thread safe CFX. Use named locks to prevent parallel access to data.

How CFLOCK works

The CFLOCK tag can single-thread access to the CFML constructs in its body, so that the body of the tag can be executed by at most one request at a time. By default, a request executing inside a CFLOCK tag has an "exclusive lock" on the tag. No other requests are allowed to start executing inside the tag while a request has an exclusive lock. ColdFusion issues exclusive locks on a first-come first-serve basis.

However, ColdFusion offers provisions for allowing read-only access to locked code. The CFLOCK tag offers two modes of locking:

- **Exclusive locks** allow only one request to process the locked code.
- **Read-only locks** allow multiple requests to execute concurrently, provided that no exclusive locks are executing.

Note Unless you specify the TYPE attribute, the default lock is exclusive. You should minimize the use of exclusive locks. If you have performance-sensitive code inside CFLOCK tags, consider adding the TYPE="ReadOnly" attribute to CFLOCK tags that do not update shared data.

Using CFLOCK

ColdFusion Server is a multi-threaded web application server that can process multiple page requests at any given time. Use CFLOCK to guarantee that multiple concurrently executing requests do not manipulate shared data structures, files, or CFXs in an inconsistent manner.

Note the following:

- Using CFLOCK around CFML constructs that modify shared data ensures that the modifications occur one after the other and not all at the same time.
- Using CFLOCK around file manipulation constructs can guarantee that file updates do not fail due to files being open for writing by other applications or ColdFusion tags.
- Using CFLOCK around CFX invocations can guarantee that CFXs that are not implemented in a thread-safe manner can be safely invoked by ColdFusion. This usually only applies to CFXs developed in C++ using the CFAPI. Any C++ CFX that maintains and manipulates shared (global) data structures will have to be made thread-safe to safely work with ColdFusion. However, writing thread-safe C++ CFXs requires advanced knowledge. A CFML custom tag wrapper can be used around the CFX to make its invocation thread-safe.

Note CFLOCK uses a kernel-level synchronization object that is released automatically upon time-out and/or abnormal termination of the thread that owns it. Therefore, ColdFusion will never deadlock for an infinite period of time while processing a CFLOCK tag. However, very large time-outs can block request threads for long periods of time and thus radically decrease throughput. Always use the minimum time-out value allowed.

Avoiding deadlocks

Be sure to nest CFLOCK tags consistently. A potential cause of blocked request threads is inconsistent nesting of CFLOCK tags and inconsistent naming of locks. If you are nesting locks, you and everyone accessing the locked variables must consistently nest CFLOCK tags in the same order and use the same lock name for each scope. If everyone accessing locked variables does not adhere to these conventions, a deadlock can occur.

A deadlock is a state in which no request can execute the locked section of the page. Thus, all requests to the protected section of the page are blocked until there is a time-out. The following table shows one scenario that would cause a deadlock.

Deadlock Scenario	
User 1	**User 2**
Locks the session scope.	Locks the application scope.
Deadlock: Tries to lock application scope, but application scope is already locked by User 2.	**Deadlock:** Tries to lock session, but session is already locked by User 1.

Once a deadlock occurs neither of the users can do anything to break the deadlock, because the execution of their requests is blocked until the deadlock can be resolved by a lock time-out.

In addition, if you nest locks of different types, you can cause a deadlock. An example of this is nesting an exclusive lock inside a read lock of the same scope, or of the same name.

In order to avoid a deadlock, you and all who need to nest locks should do so in a well-specified order and name the locks consistently. In particular, if you need to lock access to the server, application, and session scopes, you must do so in the following order.

1. Lock the session scope. In the CFLOCK tag, specify thescope as "session."
2. Lock the application scope. In the CFLOCK tag, specify the scope as "application."
3. Lock the server scope. In the CFLOCK tag, specify the scope as "server."
4. Unlock the server scope.
5. Unlock the application scope.
6. Unlock the session scope.

Note You can skip any pair of lock/unlock steps in the list above if you don't need to lock a particular scope. For example, you can take out Steps 3 and 4 if you don't need to lock the server scope.

CFLOCK Examples

The following examples show how to use CFLOCK in a variety of situations.

Example with Application, Server, and Session Variables

This example shows how CFLOCK can be used to guarantee the consistency of data updates to variables in the Application, Server, and Session scopes.

The following sample code might be part of the `Application.cfm` file.

```
<HTML>
<HEAD>
    <TITLE>Define Session and Application Variables</TITLE>
</HEAD>

<H3>CFAPPLICATION Example</H3>

<P>CFAPPLICATION defines scoping for a
ColdFusion applicationand enables or disables
the storing of client and/or session variables.
This tag is placed in the Application.cfm file
for the current application.

<CFAPPLICATION NAME="ETurtle"
    SESSIONTIMEOUT=CreateTimeSpan("60")
    SESSIONMANAGEMENT="yes">

<!--- Initialize the session and application
variables that will be used by E-Turtleneck. Use
the session lock for the session variables.
The member variable sessionID creates the
session name for you. --->

<CFLOCK SCOPE="Session"
    TIMEOUT="30" TYPE ="Exclusive">
    <CFIF NOT IsDefined("session.size")>
        <CFSET session.size = "">
    </CFIF>
    <CFIF NOT IsDefined("session.color")>
        <CFSET session.color = "">
    </CFIF>
</CFLOCK>

<!--- Use the application lock for the
application variable. This variable keeps
track of the total number of turtlenecks sold.
The application lock should have the same name
as specified in the CFAPPLICATION tag. --->

<CFLOCK Scope="Application"
    TIMEOUT="30"
    TYPE="Exclusive">
    <CFIF NOT IsDefined("application.number")>
        <CFSET application.number = 1>
    </CFIF>
</CFLOCK>

<CFLOCK SCOPE="Application"
    TIMEOUT="30"
    TYPE ="ReadOnly">
```

```
    <CFOUTPUT>
    E-Turtleneck is proud to say that we have sold
    #application.number# turtlenecks to date.
    </CFOUTPUT>
</CFLOCK>
```

Tip In general, you should limit lock scopes. When locking variables, queries, and arrays (anything other than structures), you can copy to a local variable in the CFLOCK block, then reference the local variable.

The remaining sample code would appear inside the application page where customers place orders.

```
<HTML>
<HEAD>
<TITLE>CFLOCK Example</TITLE>
</HEAD>

<BODY>
<H3>CFLOCK Example</H3>

<CFIF IsDefined("form.submit")>

<!--- Lock session variables --->
<CFLOCK SCOPE="Session"
    TIMEOUT="30" TYPE="ReadOnly">
    <CFOUTPUT>Thank you for shopping E-Turtleneck.
    Today you have chosen a turtleneck in size
    <B>#form.size#</B> and in the color <B>#form.color#</B>.
    Your order number is #session.sessionID#.
    </CFOUTPUT>
</CFLOCK>

<!--- Lock session variables to assign form values to them.
To lock session variables, you should get the session ID
with the sessionID member variable. --->

<CFLOCK SCOPE="Session"
    TIMEOUT="30"
    TYPE="Exclusive">
    <CFPARAM Name=session.size Default=#form.size#>
    <CFPARAM Name=session.color Default=#form.color#>
</CFLOCK>

<!--- Lock application variable application.number to
find the total number of turtlenecks sold. If you don't
know the name of the application, you can use the member
variable applicationName to find it.--->

<CFLOCK SCOPE="Application"
    TIMEOUT="30" TYPE="Exclusive">
    <CFSET application.number=application.number + 1>
</CFLOCK>
```

```
<!--- Show the form only if it has not been submitted. --->
<CFELSE>
<FORM ACTION="cflock.cfm" METHOD="Post">

<P> Congratulations! You have just selected
the longest wearing, most comfortable turtleneck
in the world. Please indicate the color and size
you want to buy.</P>

<TABLE CELLSPACING="2" CELLPADDING="2" BORDER="0">
<TR>
<TD>Select a color.</TD>
<TD><SELECT TYPE="Text" NAME="color">
        <OPTION>red
        <OPTION>white
        <OPTION>blue
        <OPTION>turquoise
        <OPTION>black
        <OPTION>forest green
        </SELECT>
    </TD>
</TR>
<TR>
    <TD>Select a size.</TD>
    <TD><SELECT TYPE="Text" NAME="size">
        <OPTION>small
        <OPTION>medium
        <OPTION>large
        <OPTION>xlarge
        </SELECT>
    </TD>
</TR>
<TR>
    <TD></TD>
    <TD><INPUT TYPE="Submit" NAME="submit" VALUE="Submit">
    </TD>
</TR>
</TABLE>
</FORM>
</CFIF>

</BODY>
</HTML>
```

Example of synchronizing access to a file system

The following example demonstrates how to use CFLOCK to synchronize access to a file system. The CFLOCK tag protects a CFFILE tag from attempting to append data to a file already open for writing by the same tag executing on another request.

Note that if an append operation takes more that one minute, a request waiting to obtain an exclusive lock to the critical section may time out. Also, note the use of a

dynamic value for the NAME attribute to allow protection of a file with any given name.

```
<CFLOCK NAME=#FileName# TIMEOUT=60 TYPE="Exclusive">
    <CFFILE ACTION="Append"
        FILE=#FileName#
        OUTPUT=#TextToAppend#>
</CFLOCK>
```

Example of protecting ColdFusion Extensions

This example illustrates how a custom tag wrapper can be built around CFXs that are not thread-safe. The wrapper simply forwards attributes to the non thread-safe CFX that is used inside a CFLOCK tag. An anonymous lock is used here because this is the only place from which the CFX will be invoked.

```
<CFPARAM NAME="Attributes.AttributeOne" Default="">
<CFPARAM NAME="Attributes.AttributeTwo" Default="">
<CFPARAM NAME="Attributes.AttributeThree" Default="">

<CFLOCK TIMEOUT=10 TYPE="Exclusive">
    <CFX_NOT_THREAD_SAFE AttributeOne=#Attributes.AttributeOne#
        AttributeTwo=#Attributes.AttributeTwo#
        AttributeThree=#Attributes.AttributeThree#>
</CFLOCK>
```

Note This example assumes that this is the only instance this CFX is used in the application. To lock a non-thread safe CFX that used multiple times in an application, used named locking rather than anonymous locking, specifying the same name for each lock.

For more information

See the *CFML Language Reference* for more information on using CFLOCK.

CHAPTER 13

Sending and Receiving Email

You can add interactive email features to your ColdFusion applications, providing complete two-way interface to mail servers via the CFMAIL tag and the CFPOP tag. The boom in Internet mail services makes ColdFusion's enhanced email capability a vital link to your users.

Contents

Using ColdFusion with Mail Servers

Adding email to your ColdFusion applications lets you respond automatically to user requests. You can use email in your ColdFusion applications in many different ways. These are just a few examples:

- Trigger email messages based on users' requests or orders.
- Allow users to request and receive additional information or documents through email.
- Confirm customer information based on order entries or updates.
- Send invoices or reminders, using information pulled from database queries.

ColdFusion offers several ways to integrate email into your applications. For sending email, you generally use the Simple Mail Transfer Protocol (SMTP). For receiving mail, you use the Post Office Protocol (POP) to retrieve email from the mail server. To use email messaging in your ColdFusion applications you must have access to an SMTP server and/or a valid POP account.

In your ColdFusion application pages, you use the CFMAIL and CFPOP tags to send and receive mail respectively. The following sections describe ColdFusion email features and offer examples of these tags.

Sending Email Messages

Before you set up ColdFusion to send email messages, you must have access to an SMTP email server. Also, before you run application pages that refer to the email server, you may want to configure the ColdFusion Administrator to use the SMTP server so that you don't have to hard-code it in your application.

To configure ColdFusion for email:

1. Open the Mail page in the ColdFusion Administrator.
2. In the Mail Server box, enter the address of the SMTP mail server you want ColdFusion to use.
3. Normally, you leave the Server Port and Connection Timeout settings at their default values, unless you need different settings.
4. Click Apply to save the settings.
5. To verify server settings, click the Verify button to make sure ColdFusion can access your mail server.

See *Administering ColdFusion Server* for more information on the Administrator's mail settings.

Sending SMTP mail with CFMAIL

The CFMAIL tag provides support for sending SMTP email from within ColdFusion applications. The CFMAIL tag is similar to the CFOUTPUT tag, except that CFMAIL outputs the generated text as SMTP mail messages rather than to a page. You can use all the attributes and commands that you use with CFOUTPUT with CFMAIL as well.

To send a simple email message:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Sending a simple email</TITLE>
</HEAD>

<BODY>
<H1>Sample email</H1>
<CFMAIL
    FROM="Sender@Company.com"
    TO="#URL.email#"
    SUBJECT="Sample email"
>
This is a sample email to show basic email capability.

</CFMAIL>

The email was sent.

</BODY>
</HTML>
```

3. Save the file as `sendmail.cfm` in `myapps` under the Web root directory.
4. Open your browser and enter the URL that contains the file. Replace `myname@mycompany.com` with you email address. For example,

   `http://localhost/myapps/sendmail.cfm?email=myname@mycompany.com`

The template sends the email to you, through your SMTP server.

Samples uses of CFMAIL

An application page with the CFMAIL tag dynamically generates email messages based on the tag's settings. Some of the things you can accomplish with CFMAIL are:

- Send a mail message whose recipient and contents are determined by data the user enters in an HTML form.
- Use a query to send a mail message to a database-driven list of recipients.

- Use a query to send a customized mail message, such as a billing statement to a list of recipients that is dynamically populated from a database.
- Send a MIME file attachment along with a mail message.

Sending form-based email

In the example below, the contents of a customer inquiry form submittal are forwarded to the marketing department. Note that the same application page could also insert the customer inquiry into the database.

```
<CFMAIL FROM="#Form.EMailAddress#"
    TO="marketing@allaire.com"
    SUBJECT="Customer Inquiry">

A customer inquiry was posted to our Web site:

Name: #Form.FirstName# #Form.LastName#
Subject: #Form.Subject#

#Form.InquiryText#

</CFMAIL>
```

Sending query-based email

In the example below, a query ("ProductRequests") is run to retrieve a list of the customers who have inquired about a product over the last seven days. This list is then sent, with an appropriate header and footer, to the marketing department:

```
<CFMAIL QUERY="ProductRequests"
    FROM="webmaster@allaire.com"
    TO="marketing@allaire.com"
    SUBJECT="ColdFusion status report">

Here is a list of people who have inquired about
Allaire ColdFusion over the last seven days:

<CFOUTPUT>
#ProductRequests.FirstName# #ProductRequests.LastName#
(#ProductRequests.Company#) - #ProductRequests.EMailAddress#
</CFOUTPUT>

Regards,
The WebMaster
webmaster@allaire.com

</CFMAIL>
```

Note the use of the nested CFOUTPUT tag to present a dynamic list embedded within a normal CFMAIL message. The text within the CFOUTPUT is repeated for each row in

the "ProductRequests" query, while the text above and below it serve as the header and footer (respectively) for the mail message.

Sending email to multiple recipients

In the following example, a query ("CFBetaTesters") is run to retrieve a list of people who are beta testing ColdFusion. This query is then used to send a notification to each of these testers that a new version of the beta release is available:

```
<CFMAIL QUERY="CFBetaTesters"
    FROM="beta@allaire.com"
    TO="#TesterEMail#"
    SUBJECT="ColdFusion Beta Four Available">

To all ColdFusion beta testers:

ColdFusion Beta Four is now available
for downloading from the Allaire site.
The URL for the download is:

http://beta.allaire.com

Regards,
ColdFusion Technical Support
beta@allaire.com

</CFMAIL>
```

Note that in this example, the contents of the CFMAIL tag are not dynamic, that is, they don't use any # delimited dynamic parameters. What is dynamic is the list of email addresses to which the message is sent. Note the use of the "TesterEMail" column from the "CFBetaTesters" query in the TO attribute.

Customizing Email for Multiple Recipients

In the following example, a query ("GetCustomers") is run to retrieve the contact information for a list of customers. This query is then used to send an email to each customer asking them to verify that their contact information is still valid:

```
<CFMAIL QUERY="GetCustomers"
    FROM="service@allaire.com"
    TO="#EMail#"
    SUBJECT="Contact Info Verification">

Dear #FirstName# -

We'd like to verify that our customer
database has the most up-to-date contact
information for your firm. Our current
information is as follows:
```

```
Company Name: #Company#
Contact: #FirstName# #LastName#

Address:
    #Address1#
    #Address2#
    #City#, #State# #Zip#

Phone: #Phone#
Fax: #Fax#
Home Page: #HomePageURL#

Please let us know if any of the above
information has changed, or if we need to
get in touch with someone else in your
organization regarding this request.

Thanks,
Allaire Customer Service
service@allaire.com

</CFMAIL>
```

Note that in the TO attribute of CFMAIL, the #Email# query column causes one message to be sent to the address listed in each row of the query. Also note the use of the other query columns (FirstName, LastName, etc.) within the CFMAIL section to customize the contents of the message for each recipient.

Attaching a MIME file

You use the CFMAILPARAM tag to attach a file or add a header to a mail message. In the following example, a MIME-encoded file is sent along with an email message:

```
<CFMAIL FROM="abeecho@allaire.com"
    TO="bobm@supercomputer.com"
    SUBJECT="File you requested"
>

Dear Bob,

Here is a copy of the file you requested.

Regards,
A. Beech

<CFMAILPARAM FILE="c:\photos\asdl_photo.jpg">

</CFMAIL>
```

Advanced Sending Options

The ColdFusion implementation of SMTP mail uses a spooled architecture. This means that when a CFMAIL tag is processed in an application page, the messages generated are not sent immediately. Instead, they are spooled to disk and processed in the background. This architecture has two distinct advantages:

1. End users of your application are not required to wait for SMTP processing to complete before a page returns to them. This is especially useful when a user action causes more than a handful of messages to be sent.
2. Messages sent using CFMAIL are delivered reliably, even in the presence of unanticipated events like power outages or server crashes.

In most cases, spooled messages are processed immediately by ColdFusion and delivery occurs almost instantly. If, however, ColdFusion is either extremely busy or has a large existing queue of messages, delivery could occur some time after the request is submitted.

Sending mail as HTML

Most newer Internet mail applications are capable of reading and interpreting HTML code in a mail message. The CFMAIL tag allows you to specify the message type as HTML. The TYPE attribute, which only accepts HTML as an argument, informs the receiving email client that the message has embedded HTML tags that need to be processed. This feature is only useful when sending messages to mail clients that understand HTML.

Error logging and undelivered messages

All errors that occur during the processing of SMTP messages are logged to the file `errors.log` in the ColdFusion log directory. Error log entries contain the date and time of the error as well as diagnostic information on why the error occurred.

All messages not delivered because of an error are written to the `\cfusion\mail\undelivr` directory. The error log entry corresponding to the undelivered message contains the name of the file written to the `undelivr` directory.

See *Administering ColdFusion Server* for more information about the mail logging settings in the ColdFusion Administrator.

Receiving Email Messages

CFPOP, the Post Office Protocol tag, expands the ColdFusion developer's ability to add Internet mail client features and email consolidation to applications. While a conventional mail client provides an adequate interface for personal mail, there are many cases where an alternative interface to some mailboxes is desirable. CFPOP is a tool to develop targeted mail clients to suit the specific needs of a wide range of applications.

Use CFPOP in applications when you want to receive email. Here are two instances where implementing POP mail makes sense:

- If your site has generic mailboxes that are read by more than one person (*sales@yourcompany.com*), it may be more efficient to construct a ColdFusion mail front-end to supplement individual user mail clients.
- In many applications, the processing of mail can be automated when the mail is formatted to serve a particular purpose. For example, when subscribing to a list server.

See the *CFML Language Reference* for more information on CFPOP syntax and variables.

Using CFPOP

To implement the CFPOP tag in your ColdFusion application:

1. Choose which mail boxes you want to access within your ColdFusion application.
2. Determine what mail message components you need to process: message header, message body, attachments, etc.
3. Decide if you need to store the retrieved messages in a database.
4. Decide if you need to delete messages from the POP server once you've retrieved them.
5. Incorporate the CFPOP tag in your application and create a user interface for accessing a given mailbox.
6. Build an application page to handle the output. Retrieved messages can include ASCII characters that do not display properly in the browser.

   You use the CFOUTPUT tag with the HTMLCodeFormat and HTMLEditFormat functions to control output to the browser. Note the use of these functions in the examples.

CFPOP query variables

Two variables are returned for each CFPOP query that provide record number information:

- RecordCount: The total number of records returned by the query.
- CurrentRow: The current row of the query being processed by CFOUTPUT in a query-driven loop.

You can reference these properties in a CFOUTPUT tag by prefixing the query variable with the query name in the NAME attribute of CFPOP:

```
<CFOUTPUT>
This operation returned #Sample.RecordCount# messages.
</CFOUTPUT>
```

Handling POP Mail

This section gives an example of each of the following usages:

- Retrieving only message headers
- Retrieving a message body
- Retrieving attachments
- Deleting messages

Returning only message headers

The header includes:

- DATE
- FROM
- MESSAGENUMBER
- REPLYTO
- SUBJECT
- CC
- TO

To retrieve only the message header:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>POP Mail Message Header Example</TITLE>
</HEAD>

<BODY>
<H2>This example retrieves message
header information:</H2>

<CFPOP SERVER="mail.company.com"
    USERNAME=#username#
    PASSWORD=#password#
    ACTION="GetHeaderOnly"
    NAME="Sample">

<CFOUTPUT QUERY="Sample">
    MessageNumber: #HTMLEditFormat(Sample.MESSAGENUMBER)# <BR>
    To: #HTMLEditFormat(Sample.TO)# <BR>
    From: #HTMLEditFormat(Sample.FROM)# <BR>
    Subject: #HTMLEditFormat(Sample.SUBJECT)# <BR>
    Date: #HTMLEditFormat(Sample.DATE)# <BR>
```

```
    Cc: #HTMLEditFormat(Sample.CC)# <BR>
    ReplyTo: #HTMLEditFormat(Sample.REPLYTO)# <BR>
</CFOUTPUT>

</BODY>
</HTML>
```

3. Change the following line so that it refers to a valid POP mail server, as well as a valid user name and password:

```
<CFPOP SERVER="mail.company.com"
    USERNAME=#username#
    PASSWORD=#password#
```

4. Save the file as `hdronly.cfm` in `myapps` under the Web root directory.

This code retrieves the message headers and stores them in a CFPOP result set called *Sample*.

You can enclose header information in HTML coding and use the ColdFusion function HTMLCodeFormat to replace HTML tags with escaped characters, such as `>` for > and `<` for <.

In addition, you can process the date returned by CFPOP with ParseDateTime, which accepts an argument for converting POP date/time objects into GMT (Greenwich Mean Time).

See the *CFML Language Reference* for information on these ColdFusion functions.

You can reference any of these columns in a CFOUTPUT tag, as the following example shows.

```
<CFOUTPUT>
    #ParseDateTime(queryname.date, "POP")#
    #HTMLCodeFormat(queryname.from)#
    #HTMLCodeFormat(queryname.messagenumber)#
</CFOUTPUT>
```

Returning an entire message

When you use the CFPOP tag with ACTION="GetAll", ColdFusion returns the same columns returned with GETHEADERONLY, as well as two additional columns, BODY and HEADER.

To retrieve an entire message:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>POP Mail Message Body Example</TITLE>
</HEAD>

<BODY>
<H2>This example adds retrieval of
```

```
the message body:</H2>

<CFPOP SERVER="mail.company.com"
    USERNAME=#username#
    PASSWORD=#password#
    ACTION="GetAll"
    NAME="Sample">

<CFOUTPUT QUERY="Sample">
    MessageNumber: #HTMLEditFormat(Sample.MESSAGENUMBER)# <BR>
    To: #HTMLEditFormat(Sample.TO)# <BR>
    From: #HTMLEditFormat(Sample.FROM)# <BR>
    Subject: #HTMLEditFormat(Sample.SUBJECT)# <BR>
    Date: #HTMLEditFormat(Sample.DATE)# <BR>
    Cc: #HTMLEditFormat(Sample.CC)# <BR>
    ReplyTo: #HTMLEditFormat(Sample.REPLYTO)# <BR>
    Body: #HTMLCodeFormat(Sample.BODY)# <BR>
    Header: #HTMLCodeFormat(Sample.HEADER)# <BR>
</CFOUTPUT>

</BODY>
</HTML>
```

3. Change the following line so that it refers to a valid POP mail server, as well as a valid user name and password:

```
<CFPOP SERVER="mail.company.com"
    USERNAME=#username#
    PASSWORD=#password#
```

4. Save the file as `hdrbody.cfm` in `myapps` under the Web root directory.

Returning attachments with messages

When you use the CFPOP tag with ACTION="GetAll", and add the ATTACHMENTPATH attribute, ColdFusion returns two additional columns:

- ATTACHMENTS contains a a tab-separated list of all source attachment names.
- ATTACHMENTFILES contains a tab-separated list of the actual temporary filenames written to the server. Use the CFFILE tag to delete the temporary files.

Not all messages have attachments. If a message has no attachments, both ATTACHMENTS and ATTACHMENTFILES will be equal to an empty string.

To retrieve all parts of a message including attachments:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>POP Mail Message Attachment Example</TITLE>
</HEAD>
```

```
<BODY>
<H2>This example retrieves message header,
body, and all attachments:</H2>

<CFPOP SERVER="mail.company.com"
    USERNAME=#username#
    PASSWORD=#password#
    ACTION="GetAll"
    ATTACHMENTPATH="c:\attachdir"
    NAME="Sample">

<CFOUTPUT QUERY="Sample">
    MessageNumber: #HTMLEditFormat(Sample.MESSAGENUMBER)# <BR>
    To: #HTMLEditFormat(Sample.TO)# <BR>
    From: #HTMLEditFormat(Sample.FROM)# <BR>
    Subject: #HTMLEditFormat(Sample.SUBJECT)# <BR>
    Date: #HTMLEditFormat(Sample.DATE)# <BR>
    Cc: #HTMLEditFormat(Sample.CC)# <BR>
    ReplyTo: #HTMLEditFormat(Sample.REPLYTO)# <BR>
    Attachments: #HTMLEditFormat(Sample.ATTACHMENTS)# <BR>
    Attachment Files: #HTMLEditFormat(Sample.ATTACHMENTFILES)# <BR>
    Body: #HTMLCodeFormat(Sample.BODY)# <BR>
    Header: #HTMLCodeFormat(Sample.HEADER)# <BR>
</CFOUTPUT>

</BODY>
</HTML>
```

3. Change the following line so that it refers to a valid POP mail server, as well as a valid user name and password:

```
<CFPOP SERVER="mail.company.com"
    USERNAME=#username#
    PASSWORD=#password#
```

4. Save the file as `attach.cfm` in `myapps` under the Web root directory.

Note To avoid duplicate file names when saving attachments, use the GENERATEDUNIQUEFILENAMES attribute of CFPOP and set it to Yes.

Deleting messages

By default, retrieved messages are not deleted from the POP mail server. If you want to delete retrieved messages, you must set the ACTION attribute to Delete.

Note Once a message is deleted, it's gone for good.

The MESSAGENUMBER attribute returned by all CFPOP retrievals contains the message number you need to pass back to the POP mail server to have the corresponding message deleted. A few notes:

Note Message numbers are reassigned at the end of every POP mail server communication that contains a delete action. For example, if four messages are retrieved from a POP mail server, the message numbers

returned will be 1,2,3,4. If messages 1 and 2 are then deleted within a single CFPOP tag, messages 3 and 4 will be assigned message numbers 1 and 2, respectively.

To delete messages:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
<TITLE>POP Mail Message Delete Example</TITLE>
</HEAD>

<BODY>
<H2>This example deletes messages:</H2>

<CFPOP SERVER="mail.company.com"
    USERNAME=#username#
    PASSWORD=#password#
    ACTION="Delete"
    MESSAGENUMBER="1,2,3">

</BODY>
</HTML>
```

3. Change the following line so that it refers to a valid POP mail server, as well as a valid user name and password:

```
<CFPOP SERVER="mail.company.com"
    USERNAME=#username#
    PASSWORD=#password#
```

4. Save the file as `msgdel.cfm` in `myapps` under the Web root directory.

CHAPTER 14

Managing Files on the Server

The CFFILE, CFDIRECTORY, and CFCONTENT tags handle browser/server file management tasks. To perform server-to-server operations, use the CFFTP tag.

Contents

Using CFFILE

The CFFILE tag gives you the ability to work with files on your server in a number of ways:

- Uploading files from a client to the Web server using an HTML form.
- Moving, renaming, copying, or deleting files on the server.
- Reading, writing, or appending to text files on the server.

The required attributes depend on the ACTION specified. For example, if ACTION="WRITE", ColdFusion expects the attributes associated with writing a text file.

Note Consider the security and logical structure of directories on the server before allowing users access to them.

Uploading Files

File uploading requires that you create two files:

- An HTML form to enter file upload information
- An action page containing the file upload code

To create an HTML file to specify file upload information:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Specify File to Upload</TITLE>
</HEAD>

<BODY>
<H2>Specify File to Upload</H2>
<FORM ACTION="uploadfileaction.cfm"
ENCTYPE="multipart/form-data"
        METHOD="post">
<P>Enter the complete path and filename of the file to upload:
    <INPUT TYPE="file"
            NAME="FiletoUpload"
            SIZE="45">
</P>
    <INPUT TYPE="submit"
            VALUE"Upload">
</FORM>
</BODY>
</HTML>
```

3. Save the file as `uploadfileform.cfm` in `myapps` under the Web root directory.

Code Review

Code	Description
<FORM ACTION="uploadfileaction.cfm" ENCTYPE="multipart/form-data" METHOD="post">	Create a form that contains file selection fields for upload by the user.
<INPUT TYPE="file" NAME="FiletoUpload" SIZE="45">	Allow the user to input a field. (The File input type automatically includes a Browse button to allow the user to look for the file instead of entering the entire path and file name.)

HTML forms can be designed in most browsers to give users the ability to upload files. Setting the HTML INPUT tag type to "file" instructs the browser to prepare to read and transmit a file from the user's system to your server. Setting the ENCTYPE FORM attribute to "multipart/form-data" tells the server that the form submission contains an uploaded file.

The user can enter a file path or browse the system and pick a file to send.

To create an action page to upload the file:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Upload File</TITLE>
</HEAD>

<body>
<H2>Upload File</H2>

<CFFILE ACTION="UPLOAD"
        DESTINATION="c:\inetpub\wwwroot\HR\"
        NAMECONFLICT="Overwrite"
        FILEFIELD="FiletoUpload">


<CFOUTPUT>
You uploaded the file #File.ClientFileName#.#File.ClientFileExt#
successfully to
#File.ServerDirectory#\#File.ServerFileName#.#File.ServerFileExt#.
</CFOUTPUT>

</BODY>
</HTML>
```

3. Change the following line to point to an appropriate location on your server:
   `DESTINATION="c:\inetpub\wwwroot\HR\"`
4. Save the file as `uploadfileaction.cfm` in `myapps` under the Web root directory.
5. View `uploadfileform.cfm` in your browser, enter values and submit the form.
6. The file you specified is uploaded.

Code Review

Code	Description
`<CFFILE ACTION="UPLOAD"`	Prepare to upload a file to the server.
`DESTINATION="c:\inetpub\wwwroot\HR\"`	Specify the destination of the file.
`NAMECONFLICT="Overwrite"`	If the file already exists, overwrite it.
`FILEFIELD="FiletoUpload">`	Specify the name of the file to upload. **Note that you do not enclose the variable in pound signs.**

Note This example performs no error checking and does not incorporate any security measures. Before deploying an application that performs file uploads, be sure to incorporate both error handling and security.

Resolving conflicting file names

When a file is saved to the server, there is a risk that another file may already exist with the same name. In the event of this occurrence, there are a number of actions you can take using the NAMECONFLICT attribute. For example, you can specify the parameter `NAMECONFLICT="MAKEUNIQUE"` in the CFFILE tag to create a unique file name while keeping the file extension the same.

Controlling the type of file uploaded

For some applications, you might want to restrict the type of file that is uploaded. For example, you may not want to accept graphic files in a document library.

The ACCEPT attribute is used to restrict the type of file that will be allowed in an upload. When an ACCEPT qualifier is present, the uploaded file's MIME content type must match the criteria specified or an error will occur. ACCEPT takes a comma-separated list of MIME data names, optionally with wildcards.

A file's MIME type is determined by the browser. Common types, like "image/gif" and "text/plain", are registered in your browser.

Note Not all browsers support MIME type associations.

Example: Restricting file types

This CFFILE specification will only save an image file that is in the GIF format:

```
<CFFILE ACTION="Upload"
    FILEFIELD="UploadFile"
    DESTINATION="c:\uploads\MyImage.GIF"
    NAMECONFLICT="OVERWRITE"
    ACCEPT="image/gif">
```

This CFFILE specification will only save an image file that is either a GIF or a JPEG:

```
<CFFILE ACTION="Upload"
    FILEFIELD="UploadFile"
    DESTINATION="c:\uploads\MyImage.GIF"
    NAMECONFLICT="OVERWRITE"
    ACCEPT="image/gif, image/jpeg">
```

This CFFILE specification will only save an image file, but the format doesn't matter:

```
<CFFILE ACTION="Upload"
    FILEFIELD="UploadFile"
    DESTINATION="c:\uploads\MyImage.GIF"
    NAMECONFLICT="OVERWRITE"
    ACCEPT="image/*">
```

Note Any file will be saved if ACCEPT is omitted, left empty, or contains "*/*".

Setting File and Directory Attributes

File attributes in Windows are defined using the CFFILE ATTRIBUTES attribute. In UNIX, file and directory permissions are defined using the CFFILE and CFDIRECTORY MODE attribute.

UNIX

In UNIX, you can set permissions on files and directories for owner, group, and other. Values for the MODE attribute correspond to octal values for the UNIX `chmod` command:

- 4 = Read only
- 2 = Read/write
- 1 = Read/write/execute

You enter permissions values in the MODE attribute for each type of user: owner, group, other in that order. For example to assign read permissions for all:

```
MODE=444
```

To give a file or directory owner read/write/execute permissions and read only permissions for everyone else:

```
MODE=744
```

Windows

In Windows, you can set the following file attributes:

- ReadOnly
- Temporary
- Archive
- Hidden
- System
- Normal

If ATTRIBUTES is not used, the file's existing attributes are maintained. If Normal is specified as well as any other attributes, Normal is overridden by whatever other attribute is specified.

Example: Setting file attribute

This example sets the archive bit for the uploaded file:

```
<CFFILE ACTION="Copy"
    SOURCE="c:\files\upload\keymemo.doc"
    DESTINATION="c:\files\backup\"
    ATTRIBUTES="Archive">
```

Note Be sure to include the traililng slash (\) in the source and destination file names.

Evaluating the Results of a File Upload

After a file upload is completed, you can retrieve status information using file upload variables. This status information includes a wide range of data about the file, such as the file's name and the directory where it was saved.

Although you can use either the File or CFFILE prefix, for file uplodad status variables, CFFILE is preferred, for example, `CFFILE.ClientDirectory`. (The `File` prefix is retained for backward compatibility.) The file status variables can be used anywhere that ColdFusion variables are used.

The following file upload status variables are available after an upload.

File Upload Variables	
Parameter	**Description**
AttemptedServerFile	Initial name ColdFusion used attempting to save a file, for example, `myfile.txt`. See "Resolving conflicting file names" above.
ClientDirectory	Directory location of the file uploaded from the client's system.
ClientFile	Name of the file uploaded from the client's system, such as `myfile.txt`.
ClientFileExt	Extension of the uploaded file on the client's system without a period, for example, `txt` not `.txt`.
ClientFileName	Filename without an extension of the uploaded file on the client's system.
ContentSubType	MIME content subtype of the saved file, such as `gif` for `image/gif`.
ContentType	MIME content type of the saved file, such as `image` for `image/gif`.
DateLastAccessed	Date and time the uploaded file was last accessed.
FileExisted	Indicates (Yes or No) whether or not the file already existed with the same path.
FileSize	Size of the uploaded file.
FileWasAppended	Indicates (Yes or No) whether or not ColdFusion appended the uploaded file to an existing file.
FileWasOverwritten	Indicates (Yes or No) whether or not ColdFusion overwrote a file.
FileWasRenamed	Indicates (Yes or No) whether or not the uploaded file was renamed to avoid a name conflict.
FileWasSaved	Indicates (Yes or No) whether or not ColdFusion saved a file.
OldFileSize	Size of a file that was overwritten in the file upload operation.
ServerDirectory	Directory of the file actually saved on the server.
ServeFile	Filename of the file actually saved on the server.

File Upload Variables (Continued)	
Parameter	**Description**
ServerFileExt	Extension of the uploaded file on the server, without a period, for example, `txt` not `.txt`.
ServerFileName	Filename, without an extension, of the uploaded file on the server.
TimeCreated	Time the uploaded file was created.
TimeLastModified	Date and time of the last modification to the uploaded file.

Use the File prefix to refer to these variables, for example, #CFFILE.FileExisted#.

Note File status variables are read-only. They are set to the results of the most recent CFFILE operation. If two CFFILE tags execute, the results of the first are overwritten by the subsequent CFFILE operation.

Moving, Renaming, Copying, and Deleting Server Files

With CFFILE, you can create application pages to manage files on your Web server. You can use the tag to move files from one directory to another, rename files, copy a file, or delete a file.

The examples below show static values for many of the attributes. However, the value of all or part of any attribute in a CFFILE tag can be a dynamic parameter. This makes CFFILE a very powerful tool.

Examples of moving, renaming, copying, and deleting server files	
Action	**Example code**

Move a file	`<CFFILE ACTION="Move"`
	`SOURCE="c:\files\upload\KeyMemo.doc"`
	`DESTINATION="c:\files\memo\">`

Rename a file	`<CFFILE ACTION="Rename"`
	`SOURCE="c:\files\memo\KeyMemo.doc"`
	`DESTINATION="c:\files\memo\OldMemo.doc">`

Copy a file	`<CFFILE ACTION="Copy"`
	`SOURCE="c:\files\upload\KeyMemo.doc"`
	`DESTINATION="c:\files\backup\">`

Delete a file	`<CFFILE ACTION="Delete"` `FILE="c:\files\upload\oldfile.txt">`

Reading, Writing, and Appending to a Text File

In addition to managing files on the server, you can use CFFILE to read, create, and modify text files.

This gives you the ability to

- Create log files.
- Generate static HTML documents.
- Use text files to store information that can be brought into Web pages.

Reading a text file

You can use CFFILE to read an existing text file. The file is read into a dynamic parameter which you can use anywhere in the application page. For example, you could read a text file and then insert its contents into a database. Or you could read a text file and then use one of the find and replace functions to modify the contents.

To read a text file:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Read a Text File</TITLE>
</HEAD>

<BODY>
Ready to read the file:<BR>
<CFFILE ACTION="Read"
FILE="C:\inetpub\wwwroot\mine\message.txt"
VARIABLE="Message">

<CFOUTPUT>
    #Message#
</CFOUTPUT>
</BODY>
</HTML>
```

3. Replace `c:\inetpub\wwwroot\mine\message.txt` with the location and name of a text file on your server.
4. Save the file as `readtext.cfm` and view it in your browser.

Writing a text file

You can use CFFILE to write a text file based on dynamic content. For example, you could create static HTML files or log actions in a text file.

To create a form in which to enter data for a text file:

1. Open a new file in Studio.
2. Modify the file so that it appears a follows:

```
<HTML>
<HEAD>
    <TITLE>Put Information into a Text File</TITLE>
</HEAD>

<BODY>
<H2>Put Information into a Text File</H2>

<FORM ACTION="writetextfileaction.cfm" METHOD="POST">
    <p>Enter you name: <INPUT TYPE="text" NAME="Name" SIZE="25">
    <p>Enter you the name of the file: <INPUT TYPE="text"
NAME="FileName" SIZE="25">
    <p>Enter your message:</p>
    <INPUT TYPE="textarea" NAME="message"cols=45 rows=6>
    </p>
    <INPUT TYPE="submit" NAME="submit" VALUE="Submit">
</FORM>

</BODY>
</HTML>
```

3. Save the file as `writetextfileform.cfm` in `myapps` under the Web root directory.

To write a text file:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>Untitled</TITLE>
</HEAD>
<BODY>
<CFFILE ACTION="Write"
    FILE="C:\inetpub\wwwroot\mine\#form.filename#"
    OUTPUT="Created By: #Form.Name#
    #Form.Message# ">
</BODY>
</HTML>
```

3. Modify the path `C:\inetpub\wwwroot\mine\` to point to a path on your server.
4. Save the file as `writetextfileaction.cfm`.
5. View the file `writetextfileform.cfm` in your browser, enter values, and submit the form.

The text file is written to the location you specified.

You can use `CFFILE ACTION="Append"` to append additional text to the end of an existing text file, for example, when creating log files.

Performing Directory Operations

Use the CFDIRECTORY tag to return file information from a specified directory and to create, delete, and rename directories.

As with CFFILE, ColdFusion administrators can disable CFDIRECTORY processing in the ColdFusion Administrator Tags page. See the *CFML Language Reference* for details on the syntax of this tag.

Returning file information

When using the ACTION=LIST, CFDIRECTORY returns five result columns you can reference in your CFOUTPUT:

- Name — Directory entry name.
- Size — Directory entry size.
- Type — File type: F or D for File or Directory.
- DateLastModified — Date an entry was last modified.
- Attributes — File attributes, if applicable.
- Mode — (Solaris only) The octal value representing the permissions setting for the specified directory. For information about octal values, refer to the man pages for the `chmod` shell command.

To view directory information:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HTML>
<HEAD>
    <TITLE>List Directory Information</TITLE>
</HEAD>

<BODY>
<H2>List Directory Information</H2>
<CFDIRECTORY
    DIRECTORY="c:\inetpub\wwwroot\mine"
    NAME="mydirectory"
    SORT="size ASC, name DESC, datelastmodified">

<TABLE>
<TR>
    <TH>Name</TH>
    <TH>Size</TH>
    <TH>Type</TH>
    <TH>Modified</TH>
    <TH>Attributes</TH>
    <TH>Mode</TH>
</TR>
```

```
<CFOUTPUT QUERY="mydirectory">
<TR>
    <TD>#mydirectory.name#</TD>
    <TD>#mydirectory.size#</TD>
    <TD>#mydirectory.type#</TD>
    <TD>#mydirectory.datelastmodified#</TD>
    <TD>#mydirectory.attributes#</TD>
    <TD>#mydirectory.mode#</TD>
</TR>
</CFOUTPUT>
</TABLE>

</BODY>
</HTML>
```

3. Modify the line `DIRECTORY="c:\inetpub\wwwroot\mine"`so that it points to a directory on your server.
4. Save the file as `directoryinfo.cfm` and view it in your browser.

CHAPTER 15

Interacting with Remote Servers

This chapter describes how ColdFusion wraps the complexity of Hypertext Transfer Protocol communications in a simplified tag syntax that allows you to easily extend your site's offerings across the Web.

Contents

Using CFHTTP to Interact with the Web

The CFHTTP tag is one of the more powerful tags in the CFML tag set. You can use one of two methods to interact with a remote server using the CFHTTP tag: Get or Post. The Get method is a one-way transaction in which CFHTTP retrieves an object. By comparison, the Post method is a two-way transaction in which CFHTTP passes variables to a ColdFusion page or CGI program which then returns data, usually processing what was received.

Allaire Alive

A video titled, "Creating Web Agents" is available at http://alive.allaire.com. It gives an overview of HTTP and covers the use of CFHTTP for creating automated processes such as:

- Search agents
- Transaction agents
- Messaging agents

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.

Using the CFHTTP Get Method

You use Get to retrieve text and binary files from a specified server. The examples below illustrate a few common GET operations.

To retrieve a file and store it in a variable:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<CFHTTP METHOD="Get"
    URL="http://www.allaire.com/index.cfm"
    RESOLVEURL="Yes">

<CFOUTPUT>
    #CFHTTP.FileContent# <BR>
</CFOUTPUT>
```

3. Save the file as `getwebpage.cfm` in `myapps` under your Web root directory and view it in your browser.

Code Review

Code	Description
<CFHTTP METHOD="Get" URL="http://www.allaire.com/index.cfm" RESOLVEURL="Yes">	Get the page specified in the URL and make the links absolute instead of relative..
<CFOUTPUT> #CFHTTP.FileContent# </CFOUTPUT>	Display the page, which is stored in the variable CFHTTP.FileContent, in the browser.

To get a Web page and save it in a file:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<CFHTTP
    METHOD = "get"
    URL="http://www.allaire.com/index.cfm"
    PATH="c:\mine"
    FILE="allaireindex.cfm">
```

3. Change the path from `c:\mine` to point to a path on your hard drive.
4. Save the file as `savewebpage.cfm` and view it in your browser.

Code Review

Code	Description
<CFHTTP METHOD = "get" URL="http://www.allaire.com/index.cfm" PATH="c:\mine" FILE="allaireindex.cfm">	Get the page specified in the URL and save it in the file specified in PATH and FILE. Note that when the PATH and FILE attributes are used, the RESOLVEURL attribute is ignored, even if present.

To get a binary file and save it:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<CFHTTP
    METHOD="Get"
    URL="http://maximus/downloads/quakestuff/q2_test.zip"
    PATH="c:\quake2\install"
    FILE="quake2beta.zip">

<CFOUTPUT>
    #CFHTTP.MimeType#
</CFOUTPUT>
```

3. Change the URL to point to a binary file you want to download.
4. Change the path to point to a path on your hard drive.
5. Save the file as `savebinary.cfm` in `myapps` under your Web root directory and view it in your browser.

Code Review

Code	Description

`<CFHTTP`
`  METHOD="Get"`
`  URL="http://maximus/downloads/quakestuff/q2_test.zip"`
`  PATH="c:\quake2\install"`
`  FILE="quake2beta.zip">`
	Get a binary file and save it in the PATH and FILE specified.

`<CFOUTPUT>`
`  #CFHTTP.MimeType#`
`</CFOUTPUT>`
	Display the MIME type of the file.

Creating a Query from a Text File

Using the CFHTTP Get operation, you can create a query object from a delimited text file. This is a powerful means for processing and handling generated text files. Once the query object is created, it is very simple to reference columns in the query and perform other ColdFusion operations on the data.

Text files are processed in the following manner:

- You specify a delimiter with the DELIMITER attribute. If data in a field includes the delimiter character, it must be quoted or qualified with some other character, which you specify with the TEXTQUALIFIER attribute.
- The first row of a text file is always interpreted as column headings, so that row is skipped. If the first row doesn't contain column headings, you'll need to use the COLUMNS attribute to specify headings so that you don't lose the first row data. You can also use the COLUMNS attribute to specify alternate heading text. Just make sure that you enter an alternate for every column of data in the text file.

- When duplicate column heading names are encountered, ColdFusion adds an underscore character to the duplicate column name to make it unique. For example, if two CustomerID columns are found, the second is renamed "CustomerID_".

To create a query from a text file:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<!--- The text file consists of six columns --->
<!--- separated by commas. --->
<!--- The rows are --->
<!--- OrderID,OrderNum,OrderDate --->
<!--- ShipDate,ShipName,ShipAddress --->
<!--- This example accepts the first row --->
<!--- of the text file as the column names --->

<CFHTTP METHOD="Get"
    URL="http://127.0.0.1/orders/june/orders.txt"
    NAME="juneorders"
    DELIMITER=","
    TEXTQUALIFIER="""">

<CFOUTPUT QUERY="juneorders">
    OrderID: #OrderID#<BR>
    Order Number: #OrderNum#<BR>
    Order Date: #OrderDate#<BR>
</CFOUTPUT>

<!--- You can substitute different column names --->
<!--- by using the COLUMNS attribute --->

<CFHTTP METHOD="Get"
    URL="http://127.0.0.1/orders/june/orders.txt"
    NAME="juneorders"

    COLUMNS="ID, Number,Date"
    DELIMITER=","
    TEXTQUALIFIER="""">

<CFOUTPUT QUERY="juneorders">
    Order ID: #ID#<BR>
    Order Number: #Number#<BR>
    Order Date: #Date#<BR>
</CFOUTPUT>
```

3. Substitue the URL with the location of your text file.
4. Substitue the name of a text file and the column headers to those in your text file.
5. Save the file as `querytextfile.cfm` in `myapps` under your Web root directory and view it in your browser.

Using the CFHTTP Post Method

Use the Post method to send cookie, form field, CGI, URL, and file variables to a specified ColdFusion page or CGI program. For Post operations, you must use the CFHTTPPARAM tag for each variable you want to post. Unlike the Get method, Post passes data to a specified ColdFusion page or to some executable that interprets the variables being sent and returns data.

For example, when you build an HTML form using the Post method, you specify the name of the program to which form data will be passed. Using the Post method in CFHTTP is exactly the same.

To pass variables to a ColdFusion page:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<CFHTTP METHOD="Post"
    URL="http://127.0.0.1/dwa_code/server.cfm"
    USERNAME="user1"
    PASSWORD="user1pwd">

    <CFHTTPPARAM TYPE="Cookie"
        VALUE="cookiemonster"
        NAME="mycookie6">
    <CFHTTPPARAM TYPE="CGI"
        VALUE="cgivar "
        NAME="mycgi">
    <CFHTTPPARAM TYPE="URL"
        VALUE="theurl"
        NAME="myurl">
    <CFHTTPPARAM TYPE="Formfield"
        VALUE="wbfreuh@allaire.com"
        NAME="emailaddress">
    <CFHTTPPARAM TYPE="File"
        NAME="myfile"
        FILE="c:\temp\cyberlogo.gif">
</CFHTTP>

<CFOUTPUT>
    #CFHTTP.filecontent#
    #CFHTTP.mimetype#
</CFOUTPUT>
```

3. Replace the URL with one on your server.
4. Save the file as `server.cfm` in `myapps` under your Web root directory.

To view the variables:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
You have POSTed to me.<BR>
<CFFILE DESTINATION="c:\temp\junk"
    NAMECONFLICT="Overwrite"
    FILEFIELD="myfile"
    ACTION="Upload"
    ATTRIBUTES="Normal">

<CFOUTPUT>
    The URL variable is: #url.myurl# <BR>
    The Cookie variable is: #cookie.mycookie6# <BR>
    The CGI variable is: #cgi.mycgi#. <BR>
    The Formfield variable is: #form.myformfield#. <BR>
</CFOUTPUT>
```

3. Replace `c:\temp\junk` with a path and filename on your hard drive.
4. Save the file as `posttest.cfm` in `myapps` under your Web root directory.

This example uses the CFFILE tag to upload the contents of the file variable to `c:\temp\junk.`

It passes the five supported variable types to the page specified in the URL attribute. The page that receives this data is also shown. It returns the value of the variables which appears in the client's browser. This example uses the CFFILE tag in the page that receives the Posted variables to upload the contents of the file variable to `c:\temp\junk.`

The CFOUTPUT section in `posttest.cfm` references the `CFHTTP.FileContent` variable, which is used to display the output from the `server.cfm` file. If the `CFHTTP.FileContents` variable were left out, the browser output would be limited to the contents of the `posttest.cfm` file.

To return resuls of a CGI program:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<CFHTTP METHOD="Post"
    URL="http://www.thatsite.com/search.exe"
    RESOLVEURL="Yes">

    <CFHTTPPARAM TYPE="Formfield"
        NAME="search"
        VALUE="hello">

</CFHTTP>

<CFOUTPUT>
    #CFHTTP.MimeType#<BR>
    Length: #len(cfhttp.filecontent)# <BR>
    Content: #htmlcodeformat(cfhttp.filecontent)#<BR>
</CFOUTPUT>
```

3. Save the file as `getcgivars.cfm` in `myapps` under your Web root directory.

This example runs a CGI program, `search.exe`, that searches the site and returns the hits on the value specified in VALUE.

Using Secure Sockets Layer (SSL) with CFHTTP

When using Secure Sockets Layer (SSL) to transmit secured transactions via CFHTTP, you need to be aware of limitations on its use caused by a bug in one of the components of Microsoft's InetSDK. The problem occurs on Windows NT and should not affect Windows 95/98 machines.

CFHTTP uses the InetSDK to conduct all HTTP and HTTPS transactions and relies on the WinInet DLL and Schannel DLL for its SSL implementation. The WinInet bug constrains storage of SSL certificate information to the user level. This means that WinInet does not interrogate the registry for certificate information if the client that loads the DLL is a service. As a result, a CFHTTP request to an `https:// url,` will fail if ColdFusion is running as a service.

While we strongly recommend that the ColdFusion Server be run as a service, a workaround for this SSL/WinInet problem is available. The workaround is to run ColdFusion as a desktop application when SSL is needed. In this way, WinInet will write to and read from the registry appropriately when negotiating certificate information.

To run ColdFusion as a desktop application:

1. From the Windows NT Start menu in, select Run.
2. Type the following (assuming that your installation of CF is in the default location): `c:\cfusion\bin\cfserver –DESKTOP`
3. The ColdFusion icon should appear in the Windows Task Bar.

When running the ColdFusion server as a desktop application rather than as a service, keep the following in mind:

- Access the server from the Window Control Panel Services dialog.
- The server must be cycled manually by loading and unloading the ColdFusion Application Server process.
- The server cannot be stopped or started from the ColdFusion Administrator.
- If the server goes down, the Executive will restart it as a service, not as a desktop application, and all subsequent SSL transactions will fail.

To determine whether the encryption key size conforms to export laws:

1. Right mouse click on Schannel.dll.
2. Select Properties.
3. Click the Version tab.

   If the Description field reads "PCT / SSL Security Provider (Export Version)", a 40-bit key was used.

Performing File Operations with CFFTP

The CFFTP tag allows you to perform tasks on remote servers via the File Transfer Protocol (FTP). CFFTP allows you to cache connections for batch file transfers.

Note In order to use CFFTP, make sure CFOBJECT is enabled on the Basic Security page of the ColdFusion Administrator.

For server/browser operations, use the CFFILE, CFCONTENT, and CFDIRECTORY tags.

Note CFFTP is a COM object and is not supported in Microsoft Windows NT 3.51.

Using CFFTP involves two distinct types of operations, connecting and transferring files. For a complete list of attributes, see the *CFML Language Reference* .

To open an FTP connection and retrieve a file listing:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<--- open FTP connection --->
<CFFTP CONNECTION=FTP
    USERNAME="betauser"
    PASSWORD="monroe"
    SERVER="beta.company.com"
    ACTION="Open"
    STOPONERROR="Yes">

<--- get current directory name --->
<CFFTP CONNECTION=FTP
    ACTION="GetCurrentDir"
    STOPONERROR="Yes">

<--- output directory name --->
<CFOUTPUT>
    FTP directory listing of #cfftp.returnvalue#.<p>
</CFOUTPUT>

<--- get directory info --->
<CFFTP CONNECTION=FTP
    ACTION="listdir"
    DIRECTORY="/*."
    NAME="q"
    STOPONERROR="Yes">

<--- output dirlist results --->
<HR>
<P>FTP Directory Listing:</P>

<CFTABLE QUERY="q" HTMLTABLE>
```

```
        <CFCOL HEADER="<B>Name</B>" TEXT="#name#">
        <CFCOL HEADER="<B>Path</B>" TEXT="#path#">
        <CFCOL HEADER="<B>URL</B>" TEXT="#url#">
        <CFCOL HEADER="<B>Length</B>" TEXT="#length#">
        <CFCOL HEADER="<B>LastModified</B>"
        TEXT="Date(Format#lastmodified#)">
        <CFCOL HEADER="<B>IsDirectory</B>"
            TEXT="#isdirectory#">
    </CFTABLE>
```

3. Change `beta.company.com` to the name of a server you have permission to FTP to.
4. Change `betauser` and `monroe` to a valid username and password.

   To establish an anonymous connection enter "anonymous" as the username and an email address (by convention) for the password.
5. Save the file as `ftpconnect.cfm` in `myapps` under your Web root directory.

Once you've established a connection with CFFTP, you can reuse the connection to perform additional FTP operations. When you access an already active FTP connection, you don't need to re-specify the username, password, or server. In this case, make sure that when you use frames, only one frame uses the connection object.

Caching connections across multiple pages

CFFTP caching is maintained only in the current page unless you explicitly assign a CFFTP connection to a variable with application or session scope. Assigning a CFFTP connection to an application variable could cause problems, since multiple users could access the same connection object at the same time. Creating a session variable for a CFFTP connection makes the most sense.

You cache a connection object for a session by assigning the connection name to a session variable:

Example: Caching a connection

```
<CFFTP ACTION=connect
    USERNAME="anonymous"
    PASSWORD="me@home.com"
    SERVER="ftp.eclipse.com"
    CONNECTION="Session.myconnection">
```

In this example, the connection cache remains available to other pages within the current session. Of course, you need to be sure that you've enabled session variables in your application first.

Note Changes to a cached connection, such as changing RETRYCOUNT or TIMEOUT values, may require re-establishing the connection.

Connection caching actions and attributes

The following table shows which CFFTP attributes are required for CFFTP actions when employing connection caching. If connection caching is not used, the connection attributes USERNAME, PASSWORD, and SERVER must be specified.

CFFTP Required Attributes by Action			
Action	**Attributes**	**Action**	**Attributes**
Open	none	Rename	EXISTING NEW
Close	none	Remove	SERVER ITEM
ChangeDir	DIRECTORY	GetCurrentDir	none
CreateDir	DIRECTORY	GetCurrentURL	none
ListDir	NAME DIRECTORY	ExistsDir	DIRECTORY
GetFile	LOCALFILE REMOTEFILE	ExistsFile	REMOTEFILE
PutFile	LOCALFILE REMOTEFILE	Exists	ITEM

Moving Complex Data Structures Across the Web with WDDX

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

Additionally, server-to-browser and browser-to-server JavaScript data exchanges can be instantiated using WDDX. Server data can be transferred to the browser and converted to JavaScript objects, while JavaScript data generated on the browser can be serialized, which involves translating the native data structures into an abstract representation in XML, and transferred to the application server. Conversely, you can deserialize WDDX XML into a native data structure.

This functionality is encapsulated in the CFWDDX tag.

While WDDX is a valuable tool for ColdFusion developers, it's utility is not limited to CFML. WDDX serialization of common programming data structures such as arrays, record sets, and structures enables data communication, via HTTP, across a range of languages and platforms.

The best source of information about WDDX is http://www.wddx.org/. This site, sponsored by Allaire Corporation, offers a free download of the WDDX SDK and a number of resources, including a WDDX FAQ and a developer forum.

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, Perl, Java, Python, and COM.

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.

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.

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.

WDDX is based on XML, which is a W3C Recommendation. Other W3C efforts now in the works will have obvious application to WDDX when they are completed, most importantly, the XML-Schema proposal. 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.

Time zone processing

Because producers and consumers of WDDX packets can be in geographically dispersed locations, using time zone information during the serialization and deserialization phases becomes critical for correct date-time processing.

All of Allaire's WDDX serializers (CFML, COM, and JS) have an attribute/property `useTimezoneInfo` that specifies whether time zone information should be used in the serialization process. The default value is true.

In the CFML implementation, `useTimezoneInfo` is a property of the CFWDDX Action=Cfml2WDDX tag. In the COM implementation, `useTimezoneInfo` is a property of the IWDDXSerializer interface provided by the object `WDDX.Serializer.1`. In the JS implementation `useTimezoneInfo` (note the case-sensitivity of JS) is a property of the WDDXSerializer object.

Date-time values in WDDX are represented using a subset of the ISO8601 format. Time zone information is represented as an hour/minute offset from UTC, for example, "1998-9-8T12:6:26-4:0".

During WDDX deserialization to CFML and COM time zone information is automatically taken into account and all date-time values are converted to local time. In this way, UTC is taken out of the picture entirely and developers do not need to worry about the details of time zone conversions.

However, during deserialization to JavaScript expressions, time zone information is not taken into account. Complications arise because of the difficulty of knowing the time zone of the browser.

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 Deserializer object would create a structure similar to what would be created directly by this JavaScript object declaration:

Comparison of JavaScript object and deserialized XML	
JavaScript	**CFML**

`x = new Object();`
`x.a = "Property a";`
`x.b = "Property b";`
	`x = structNew();`
	`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
- Including the WDDX JavaScript utility classes
- 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 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>

<!--- Cache the JavaScript so that subsequent requests will --->
<!--- use the cached version rather than making additional --->
<!--- requests to the server --->

<SCRIPT LANGUAGE="JavaScript"
    SRC="/CFIDE/scripts/wddx.js"></SCRIPT>

<!--- 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'>
```

```
    <!--- Dump the recordset --->
    q.dump(true);

</SCRIPT>
```

Note To see how CFWDDX Action="cfml2js" works, view the source to the page.

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 with columns firstName and lastName
    var personInfo = new WddxRecordset(new Array("firstName",
    "lastName"));

    // Add next record to end of personInfo recordset
    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" BALUE="Next" onclick="doNext()">
    <INPUT TYPE="button" BALUE="Serialize"
    onclick="serializeData(personInfo, document.personForm.wddxPacket)">
    <INPUT TYPE="submit" BALUE="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 16

Connecting to LDAP Directories

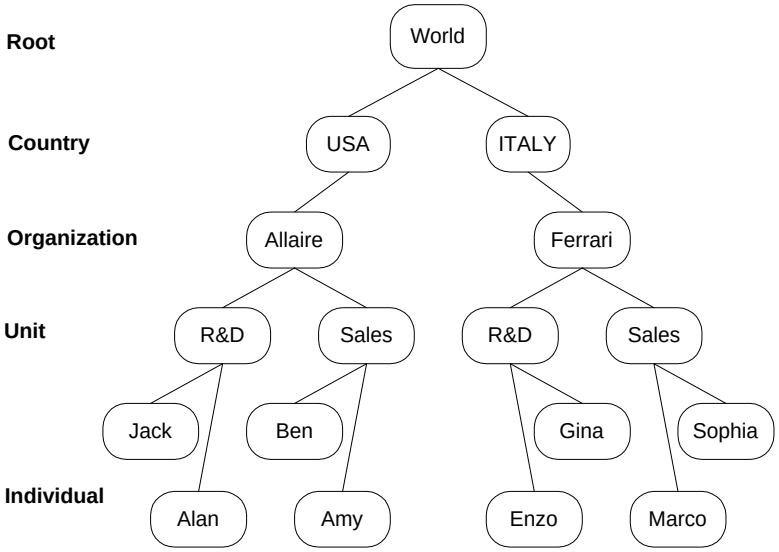
Support for the Lightweight Directory Access Protocol (LDAP) API in CFML is part of Allaire's commitment to open networking standards.

Contents

What is LDAP?

LDAP (Lightweight Directory Access Protocol) is a protocol that enables organizations to arrange directory information in a hierarchy. Note that in this case, "directory" refers to a collection of information something like a telephone directory, not a collection of files in a folder on a disk drive.

An LDAP directory is, in essence, a database, which is usually a hierarchical structure, (although this is not a requirement). It offers performance advantages over conventional databases, and its operations are familiar to database users. 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.

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. You can pass the original login credentials to other servers that you may connect to when resolving a referral.

References

Extensions to the LDAP protocol are ongoing and it is widely supported in the Internet community. 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/.

ColdFusion Support for LDAP

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.

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.

Working with LDAP Directories

The CFLDAP tag allows you to work with LDAP directories in the following ways. You can:

- View the directory schema
- Search an LDAP directory
- Process the results of querying an LDAP directory (OUTPUT)

Viewing the Directory Schema

LDAP 3.0 now supports access to a directory's schema information as part of a special entry in the root DN. You can access this information using a ColdFusion query.

To view the schema for an LDAP directory:

1. Create a new file in Studio.
2. Modify the file so that it appears as follows:

```
<HEAD>
    <TITLE>LDAP schema</TITLE>
</HEAD>

<BODY>
<CFLDAP
    NAME="EntryList"
    SERVER="testldap.company.com"
    ACTION="QUERY"
    ATTRIBUTES="dn, subschemasubentry"
    SCOPE="BASE"
    FILTER="objectclass=*"
    START=""
>

<CFOUTPUT QUERY="EntryList">
    DN: Root DSE<BR>
    subschemaSubEntry: #subschemasubentry#<BR><BR>
</CFOUTPUT>

<P><P><P>
Use that DN to get the schema attributes...
<P>

<CFLDAP NAME="EntryList2"
    SERVER="testldap.company.com"
    ACTION="Query"
```

```
    ATTRIBUTES="dn, objectclasses, attributetypes"
    SCOPE="BASE"
    FILTER="objectclass=*"
    START=#EntryList.subschemasubentry#
>

<CFOUTPUT QUERY="EntryList2">
    DN=#dn#<BR>
    objectClasses: #objectclasses#<BR><BR><BR>
    attribute Types: #attributetypes#<BR><BR>
</CFOUTPUT>

</BODY>
</HTML>
```

3. Change the SERVER from `testldap.company.com` to a valid LDAP server.
4. Save the template as `testldap.cfm` in `myapps` under your Web root directory and view it in your browser.

Note To be able to the schema for an LDAP server, the server must support LDAP 3.0.

Querying an LDAP Directory

CFLDAP allows you to search an LDAP directory and output the results of your query on a page. You can sort query results and return them to the browser or perform further processing with CFOUTPUT, CFREPORT, and related tags.

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

CFLDAP Filter Operators (Continued)	
Operator	**Example**
*	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.

The following 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."

To query an LDAP directory:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<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 beginning 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>
```

3. Save the page as `ldapadd.cfm` and view it in your browser.

Updating an LDAP Directory

Entries can be added, modified, and deleted. Remote administration of an LDAP server is one possible using one of these options.

The following example runs a cycle of LDAP actions by first adding a new record, then querying the LDAP directory and generating a form for the output, and finally deleting the new record.

To add a new record:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<!--- add a new record (Joe Smith) --->

<CFLDAP
    SERVER="myserver"
    USERNAME="uid=kvaughan, ou=People, o=airius.com"
    PASSWORD="bribery"
    ACTION="ADD"
    ATTRIBUTES="objectclass=top, person, organizationalPerson
        inetOrgPerson; cn=Joe Smith;
        sn=Smith; mail=jSmith@airius.com;
        telephonenumber=+1 408 555 2128; ou=Human Resources"
        DN="uid=jSmith, ou=People, o=airius.com">
```

```
<!--- query the LDAP server --->

<CFLDAP Name="AriusList"
    SERVER="myserver"
    ACTION="QUERY"
    ATTRIBUTES="cn,mail,telephonenumber"
    SCOPE="SUBTREE"
    FILTER="ou=Human Resources"
    SORT="cn ASC"
    START="o=airius.com">

<!--- generate a form page for query output --->

<H3> Human Resources Directory for Arius</H3>

<CFFORM ACTION="ariusform_action.cfm">

    <CFGRID NAME="ariusgrid" width="350" query="AriusList"
        insert="No" delete="No" sort="no" bold="No" italic="No"
        appendkey="No" highlighthref="No" griddataalign="LEFT"
        gridlines="no" rowheaders="no" rowheaderalign="LEFT"
        rowheaderitalic="No" rowheaderbold="No" colheaders="yes"
        colheaderalign="LEFT" colheaderitalic="No"
colheaderbold="yes"
        selectmode="BROWSE" picturebar="no">

        <CFGRIDCOLUMN NAME="cn" HEADER="Name">
        <CFGRIDCOLUMN NAME="mail" HEADER="eMail Address">
        <CFGRIDCOLUMN NAME="telephonenumber" HEADER="Phone">
    </CFGRID><BR>

</CFFORM>

<!---delete record --->

<CFLDAP
    SERVER="myserver"
    USERNAME="uid=kvaughan, ou=People, o=airius.com"
    PASSWORD="bribery"
    ACTION="DELETE"
    DN="uid=jSmith, ou=People, o=airius.com">
```

3. Change `myserver` to a valid LDAP server.
4. Change the `uid` to a valid user id.
5. Save the page as `ldapadd.cfm` and view it in your browser.

To modify a record by adding an attribute:

This example illustrates modifying a record by adding an attribute value to the existing values. This is a necessary step to overcome the limitations of the MODIFY attribute.

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<!--- modify a record, preserving
    other existing attributes --->

<!--- You must include the existing attribute
values plus the new one you want to add. In this
case we are adding a unique member gfarmer to
the Accounting Managers. If we did not include
the existing the existing unique members scarter
and tmorris then they would no longer be unique
members. The modify really is doing a replace on
this attribute. For the next release of ColdFusion
we will provide an option to just update the attribute.
Multiple values for a single attribute are separated
by a comma.  If a single attribute value contains a
comma you must escape it by adding an extra comma. For
example the uniquemember value uid=scarter,ou=groups,
o=airius.com must be entered as uid=scarter,,ou=groups,,
o=airius.com Be careful when you do this modify or you
can remove attribute values you did not intend to! --->

<!--- ATTRIBUTES="uniquemember=uid=scarter,,ou=People,,o=airius.com,
uid=tmorris,,ou=People,,o=airius.com,
uid=gfarmer,,ou=People,,o=airius.com" --->

<CFLDAP SERVER="myserver"
    ACTION="Modify"
    USERNAME="uid=kvaughan, ou=People, o=airius.com"
    PASSWORD="bribery"
    ATTRIBUTES="uniquemember=uid=scarter,,ou=People,,o=airius.com,
        uid=tmorris,,ou=People,,o=airius.com,
        id=gfarmer,,ou=People,,o=airius.com"
        DN="cn=Accounting Managers, ou=groups; o=airius.com">
```

3. Change `myserver` to a valid LDAP server.
4. Change the `uid` to a valid user id.
5. Save the page as `ldapaddattr.cfm` and view it in your browser.

To insert or update an entry:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<!--- 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="myserver"
        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="myserver"
            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="myserver"
            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="myserver"
    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>
```

3. Change `myserver` to a valid LDAP server.

4. Change the `uid` to a valid user id.
5. Save the page as `ldapchangeattr.cfm` and view it in your browser.

To delete an entry:

1. Open a new file in Studio.
2. Modify the file so that it appears as follows:

```
<!--- If the delete parameter is sent
then run this update --->
<CFIF IsDefined(dn)>
    <CFLDAP Name="LDAPDelete"
        SERVER="myserver"
        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="myserver"
    ACTION="Query"
    ATTRIBUTES="dn,cn, sn"
    SCOPE="SUBTREE"
    SORT="cn ASC"
    FILTER="(cn>=A)"
    START="o=Ace Industry, c=US"
    TIMEOUT=30>
```

3. Change `myserver` to a valid LDAP server.
4. Change the `uid` to a valid user id.
5. Save the page as `ldapdeleteattr.cfm` and view it in your browser.

Creating searchable CFLDAP output

An example of building and searching a Verity collection from LDAP data can be found in "Indexing CFLDAP Query Results" on page 163.

CHAPTER 17

Application Security

ColdFusion 4.5 supports several levels of Advanced security. This chapter teaches you how to deploy user security, which is controlled by the ColdFusion developer and offers runtime security for ColdFusion applications. It also describes the Remote Development Services security feature, which authenticates developers accessing server resources through ColdFusion Studio before giving them 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

ColdFusion Server Professional and Enterprise editions include Advanced Security features that provide scalable, granular security for building and deploying your ColdFusion applications:

- **Application development** — System administrators can control access to files, data sources and administration for each developer on your team. Coordinate team development on shared servers with the assurance that sensitive data and applications are secure.
- **Application deployment**— Create complex rules to programmatically control access to functionality within applications. Confine applications to secure areas that can flexibly restrict the access applications have to directories, components, databases or other resources on the server.

This chapter describes the ColdFusion Server features that let you integrate a total security solution into your applications.

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.

When you're working in ColdFusion Studio, you access these ColdFusion resources remotely, opening `*.cfm` files or accessing data sources. RDS security authenticates you and grants access only to the resources appropriate to your login. Authentication is carried out against the NT domain server, an ODBC data source, or an LDAP directory specified in the ColdFusion Administrator as part of a security context.

There are two ways to implement RDS security services:

- **Basic Security** — Requires developers in ColdFusion Studio to supply a password which, when authenticated, permits access to RDS Services: file browsing, editing, database operations, debugging, and so on.
- **Advanced Security** — Allows ColdFusion Administrators to restrict or permit access to file systems and data sources based on security contexts and policies established in the Advanced Security page of the ColdFusion Administrator.

Your company or ISP ColdFusion Server administrator configures RDS security so that it best meets the needs of your group.

For detailed information about setting up RDS security, See *Administering ColdFusion Server*.

Overview of User Security

User security authenticates users when they log into a ColdFusion application, and then assigns privileges based on group membership or other criteria that you determine. For example, suppose you've used ColdFusion to build and host your company's intranet. The Human Resources department maintains a page on the intranet where all employees can access timely information about the company, like the latest company policies, upcoming events, and job postings. You'd want everyone to be able to read the information, but you'd only want certain authorized HR employees to be able to add, update, or delete information. In addition, you might want to let employees view customized information about their salaries, job levels, and performance reviews. You certainly wouldn't want one employee to view sensitive information about another employee, but you'd want managers to be able to see, and possibly update, information about their direct reports. User security authenticates and authorizes users each time they try to access or work with sensitive data.

User security is made up of two components:

- Security contexts, configured in the ColdFusion Administrator, on the Advanced Security page. A security context provides the framework against which to authenticate and authorize users.
- Code you write in your application pages that checks against a security context to see if a user is allowed to access a particular resource and then takes

Before you can implement user security in your applications, you must make sure that your ColdFusion administrator has installed Advanced security on the server and has configured the appropriate security framework for your application. After the security framework is in place, you can code security features into your ColdFusion applications. For complete information about installing Advanced security and setting up a security framework, See *Administering ColdFusion Server*.

Using Advanced Security in Application Pages

Advanced security makes it easier for developers to enforce application security. After your administrator sets up the appropriate security contexts for your application, you can start using ColdFusion security tags and functions to authenticate users and see if they've been authorized for the part of the application they're trying to access.

This section describes how to use security tags and functions to authenticate users and provide or withhold resources according to the security context's rules.

- Include CFAUTHENTICATE on any application page where you want to authenticate users — that is, to make sure users are who they say they are. (You can also use CFAUTHENTICATE your application's `Application.cfm` file.) Pass the authentication information to subsequent pages where you want to 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.

- Use the IsAuthenticated function to check if the current user is authenticated.
- Use the IsAuthorized function to check if the user is authorized to access resources. This function lets developers offer or deny access to protected resources based on a user's authorization level, which is determined by already-established security contexts.
- Use the CFIMPERSONATE tag wherever you want to provide a greater level of access than is otherwise assigned to a particular user.

Read the section "Example of User Authentication and Authorization" on page 270 to see code examples that show how these tags functions work in ColdFusion applications.

To learn about syntax and usage for the CFAUTHENTICATE and CFIMPERSONATE tags, and the IsAuthenticated and IsAuthorized functions, See the *CFML Language Reference*.

Encrypting application pages

For an added measure of security, you can encrypt strings in your applications using the Encrypt and Decrypt functions. See the *CFML Language Reference* for descriptions of these functions.

Using the CFAUTHENTICATE tag

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 ColdFusion 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, ColdFusion throws a SECURITY exception. 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, if the user logs in successfully, display the originally-requested page.

Go to the section “Example of User Authentication and Authorization” on page 270 to see a longer code example.

Authentication and Authorization Functions

Once you've used CFAUTHENTICATE to check if the user is defined for a particular security context, you can use the following security functions throughout your applications any time you need to authenticate or authorize a user:

- IsAuthenticated checks if the current session has been authenticated by the CFAUTHENTICATE tag.
- IsAuthorized checks if the authenticated user has access to the named resource, based on rules defined in the security context for which the user has been authenticated.

Using the IsAuthenticated Function

The IsAuthenticated function checks whether a CFAUTHENTICATE tag has been successfully executed for the current request. If not, it looks for the CFAUTH cookie to determine if the user is authenticated or not. If you don't set a CFAUTH cookie with CFAUTHENTICATE, you must call CFAUTHENTICATE for every request in the application.

The IsAuthenticated function returns TRUE if the user has been authenticated for the current request; otherwise, it returns FALSE.

If you enter an optional security context parameter for IsAuthenticated, then it returns true if the user is authenticated in the named security context; otherwise it returns false.

```
IsAuthenticated("security_context_name")
```

Using the IsAuthorized Function

Once a user is authenticated, you can use the IsAuthorized function to check which resources the user is allowed to access. You define d authorization levels when you create security policies on the Advanced Security page of the ColdFusion Administrator.

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 data source resource called orders, use this syntax:

```
IsAuthorized("Datasource", "orders", "update")
```

In this example, 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 does not check user authorization unless 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.

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 "Exception handling strategies" on page 100

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>
```

Using the CFIMPERSONATE Tag

CFIMPERSONATE gives ColdFusion developers a way to execute a segment of code CFIMPERSONATE is useful when you want to briefly grant a type of access that you'd normally withold. Suppose you're an internet service provider (ISP) who hosts ColdFusion development services. You provide a set of custom tags that let your customers add features like hit counters, guest books, and message boards to the ColdFusion applications they create. To provide this type of functionality, you'd also need to provide access to some resources that you'd probably rather keep protected. Using CFIMPERSONATE provides access to these resources in a safe manner by wrapping the functionality in a custom tag. For example, as an ISP, you definitely wouldn't want your customers to access the CFFILE tag on your servers. However, if you provided your customers with a hit counter, you'd need to let them read specific, system-maintained files, in this case, the file that contains number of hits to the customer's homepage. You'd provide the hit-counter in a custom tag that would use the CFFILE tag. To ensure that the custom tag can access the CFFILE tag, it needs a way to impersonate a trusted user while the tag is executing and then to revert back to the non-trusted user once the trusted piece of code has completed execution.

The CFIMPERSONATE tag has the following 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 ColdFusion Administrator.

- USERNAME — The username of the user to impersonate.
- PASSWORD — The password of the user to impersonate.
- TYPE — Indicates the type of impersonation to implement, application-level or operating-system-level. Application-level impersonation lets you assume the rights assigned to a ColdFusion user by a specified security context. Operating-system-level impersonation lets you assume the rights assigned to a Windows NT user by a specified Windows NT Domain. (Operating-system-level impersonation is not currently available for UNIX.)

In addition, CFIMPERSONATE has one optional attribute:

- THROWONFAILURE — Indicates whether ColdFusion throws an exception of type Security if authentication fails. Default is Yes.

Example

The following example reads a protected file because the ColdFusion user "pfoley" has been granted access to the file by the security context "MyContext." If the user cannot be authenticated, ColdFusion throws a SECURITY exception.

```
<CFIMPERSONATE SECURITYCONTEXT="MyContext"
    USERNAME="pfoley"
    PASSWORD="admin"
    TYPE= "CF"
    THROWONFAILURE= "Yes">

    <CFFILE FILE="#readFile#" ACTION="read" VARIABLE="text">
    <CFOUTPUT>
        The file contains the following text:<BR>#text#<BR>
    </CFOUTPUT>

</CFIMPERSONATE>
```

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>
```

CHAPTER 18

Building Custom CFAPI Tags

For some applications, building executables to run with ColdFusion is the best solution. Perhaps the application requirements go beyond what is currently feasible in CFML. Or perhaps application performance can be improved for certain types of processing.

To meet these types of requirements, you can use the ColdFusion Extension Application Programming Interface (CFXAPI) to access ColdFusion functions.

While this chapter documents custom tag development using Microsoft Visual C++, or Java it is currently also possible to develop them in Inprise's Delphi.

Contents

What Are CFX Tags?

CFX tags are custom tags written against the ColdFusion Application Programming Interface. Generally, you create a CFX if you want to do something that's not possible in CFML, or if you want to improve performance of a task in CFML that's repetitive. Unlike CFML custom tags, CFXs are implemented as DLL files and can:

- Handle any number of custom attributes.
- Use and manipulate ColdFusion queries for custom formatting.
- Generate ColdFusion queries for interfacing with non-ODBC based information sources.
- Dynamically generate HTML to be returned to the client.
- Set variables within the ColdFusion application page from which they are called.
- Throw exceptions that result in standard ColdFusion error messages.

You can build CFXs using C++ or Java. Then, to be able to use the CFX, you have to register it in the ColdFusion Administrator.

Before You Begin Developing CFX Tags in C++

Sample C++ CFXs

Before you begin development of a CFX tag in C++, you may want to study the two CFX tags that are included to give you additional insight into working with the CFAPI. The two example tags are:

- `CFFX_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/cfx/examples` directory. On Solaris, look in `/<installdirectory>/coldfusion/cfx/examples`.

Setting Up Your C++ Development Environment

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\cfx\include`) to your list of global include paths. On UNIX, you will need `-I <includepath>` on your compile line (see the Makefile directory list example).

Using the Tag Wizard to create CFXs in C++

On Windows NT, you can get a start in developing CFXs by using the ColdFusion Tag Wizard. To use the wizard, the CFXAPI Tag Development Kit must be installed (it is by default), and the setup routine must detect Microsoft Visual C++ on the system.

The wizard generates a DLL file with a basic tag structure containing a single procedure. By modifying and testing this tag, you can quickly learn how to work within the API.

To build a CFX tag:

1. In Visual C++, select File > New, then click the Projects tab.
2. Select ColdFusion Tag Wizard and enter a tag name of the form `CFX_MyNewTag` in the Project name box. Click OK to open the wizard.
3. Enter the new tag name as the name of the custom tag.
4. You can optionally add text that will appear as comments in the tag's code.
5. Select an MFC usage option and click Finish to generate the code.
6. In Visual C++, select Build > Build `CFX_MyNewTag` to create the DLL file.

The next step is to make ColdFusion aware of the new tag by registering it. See "Registering CFXs" on page 289.

Compiling C++ CFXs

CFX tags built on Windows NT and UNIX must be thread safe. CFXs for Solaris should be compiled with the `-mt` switch on the Sun compiler.

Debugging C++ CFXs

Once a debug session is configured, you can run your custom tag from within the debugger, set breakpoints, single-step, and so on.

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`.

On UNIX

Solaris

You can debug custom tags on UNIX using the dbx debugger. You should shut down 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.

HP-UX 10.20

You can debug custom tags on UNIX using HP's DDE debugger. You should shut down ColdFusion using the stop script.

Set the environment variables, including `SHLIB_PATH` and `CFHOME` as they are set in the start script. You should then be able to run the cfserver executable under the DDE debugger and set break points in your CFX code. You may need to set a break point in main ("stop in main") so the debugger loads the symbols for your CFX before you can set breakpoints in your code.

Before You Begin Developing CFX Tags in Java

Because the methods and syntax are similar, if you are familiar with creating CFXs using C++ you will be productive creating CFXs in Java almost immediately. Even if you have never used the C++ based API, you will find that the Java implementation is extremely easy to learn and work with.

Sample Java CFXs

Before you begin developing a CFX tag in Java, you may want to study sample CFX tags. The Java source files for the examples can be found in the `examples` subdirectory of the main installation directory. The example tags are:

- `HelloColdFusion` - Prints a personalized greeting. Demonstrates the minimal implementation required to create a CFX.
- `ZipBrowser` - Retrieves the contents of a zip archive. Demonstrates generating a ColdFusion query and returning it to the calling page.
- `ServerDateTime` - Retrieves the date and time from a network server. Demonstrates attribute validation, using numeric attributes, and setting variables within the calling page.
- `OutputQuery` - Outputs a ColdFusion query in an HTML table. Demonstrates handling a ColdFusion query as input, throwing exceptions, and generating dynamic output.

- `HelloWorldGraphic` - Generates a "Hello World!" graphic in JPEG format. Demonstrates how to dynamically create and return graphics from a Java CFX.

Setting Up Your Development Environment to Develop CFXs in Java

You can use a wide range of Java development environments to build Java CFXs, including the Java Development Kit which you can download from Sun at

`http://www.javasoft.com/products/jdk/1.2/index.html`

Although you can use just the basic JDK, it is highly recommended that you use one of the commercial Java IDEs that provide an integrated environment for development, debugging, project management, and access to documentation. If you don't already have a Java development environment, we recommend that you try Symantec Visual Café, for which a 30 day free trial is available at

`http://www.symantec.com/domain/cafe/vcafe30.html`

Configuring the Class Path

To configure your development environment to build Java CFXs, you need to make sure that the supporting classes are visible to your Java compiler. These classes are located in the `classes\cfx.jar` archive. The full path is `<coldfusioninstalldir>/Java/classes` Consult your Java development tool's documentation to determine how to configure the compiler class path for your particular environment.

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

1. It contains the supporting classes required for developing and deploying Java CFXs. This is the `com.allaire.cfx` package located in the `cfx.jar` archive.
2. It supports a feature that allows Java CFXs located within it to be reloaded every time they are changed. Although this is not the default behavior for other Java classes, it is very useful during an iterative development and testing cycle.

Allaire strongly recommends that when you create new Java CFXs, you develop and deploy them within the `classes` directory. Following this guideline will dramatically simplify your development, debugging, and testing processes.

Once you are finished with development and testing, you can then deploy your Java CFX anywhere on the class path visible to the ColdFusion embedded JVM. See "Java Customization and Configuration" on page 289 for more details on customizing the class path.

Writing a Java CFX

To create a Java CFX, you simply create a class which implements the `CustomTag` interface. This interface contains one method, `processRequest`, which is passed `Request` and `Response` objects that are then used to do the work of the tag.

To create a Java CFX:

1. Create a new source file in your editor.
2. Enter the code, for example, the code below illustrates the creation of a very simple Java CFX named `SimpleJavaCFX` that writes a text string back to the calling page:

```
import com.allaire.cfx.* ;

public class HelloColdFusion implements CustomTag
{
   public void processRequest( Request request, Response response )
        throws Exception
   {
        String strName = request.getAttribute( "NAME" ) ;
        response.write( "Hello, " + strName ) ;
   }
}
```

3. Save the file as `HelloColdFusion.java` in the `classes` subdirectory
4. Compile the java source file into a class file using the java compiler. If you are using the command line tools bundled with the JDK, you do this using the following command line, which you execute from within the `classes` directory:

```
javac -classpath cfx.jar HelloColdFusion.java
```

Note The above command will only work if the java compiler (`javac.exe`) is in your path. If it is not in your path, specify the fully qualified path, for example:

`c:\jdk12\bin\javac` on Windows NT, or `/usr/java/bin/javac` on Solaris

If you receive errors during compilation, check the source code to make sure you have entered it correctly. If no errors occur, you have just successfully written your first Java CFX!

As you can see, implementing the basic `CustomTag` interface is very straightforward. This is all a Java class has to do to be callable from a CFML page.

Processing Requests

Implementing a Java CFX requires interaction with the `Request` and `Response` objects passed to the `processRequest` method. In addition, CFXs that need to work with ColdFusion queries will also interface with the `Query` object. The `com.allaire.cfx` package, located in the `classes/cfx.jar` archive contains the `Request`, `Response`, and `Query` objects.

A basic overview of each of these object types is provided below. To see a complete example Java CFX that uses `Request`, `Response`, and `Query` objects, see the "ZipBrowser Example" on page 284.

Request Object

Passed to the `processRequest` method of the `CustomTag` interface. Provides methods for retrieving attributes passed to the tag, including queries, and reading global tag settings.

Methods Used by Request Object	
Method	**Description**
`attributeExists`	Checks if the attribute was passed to this tag
`getAttribute`	Retrieves the value of the passed attribute
`getIntAttribute`	Retrieves the value of the passed attribute as an integer
`getAttributeList`	Retrieves a list of all attributes passed to the tag
`getQuery`	Retrieves the query that was passed to this tag, if any
`getSetting`	Retrieves the value of a global custom tag setting
`debug`	Checks if the tag contains the DEBUG attribute

Response Object

Passed to the `processRequest` method of the `CustomTag` interface. Provides methods for writing output, generating queries, and setting variables within the calling page.

Methods Used by Response Object	
Method	**Description**
`write`	Outputs text into the calling page
`setVariable`	Sets a variable in the calling page
`addQuery`	Adds a query to the calling page
`writeDebug`	Outputs text into the debug stream

Query Object

Provides an interface for working with ColdFusion queries, including methods for retrieving name, row count, and column names as well as methods for getting and setting data elements..

Methods Used by Query Object	
Method	**Description**
`getName`	Retrieves the name of the query
`getRowCount`	Retrieves the number of rows in the query
`getColumns`	Retrieves the names of the query columns
`getData`	Retrieves a data element from the query
`addRows`	Adds a new row to the query
`setData`	Sets a data element within the query

For detailed reference information on each of these interfaces see the "Java CFX Reference" on page 311.

Java CFX Class Loading

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

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

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

Automatic Class Reloading

You can determine how the server treats changed Java CFX class files by using the RELOAD (?) . The allowable values for the RELOAD attribute are as follows:.

Allowable Values of RELOAD Attribute	
Value	**Description**
Auto	Automatically reload Java CFX and dependent classes within the classes directory whenever the CFX class file changes. Does not reload if a dependent class file changes without the CFX class file changing as well.
Always	Always reload Java CFX and dependent classes within the classes directory. Ensures a reload even if a dependent class changes, but the CFX class file itself does not change.
Never	Never reload Java CFX classes. Load them once per server lifetime.

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

Note The RELOAD attribute applies only to class files located in the classes directory. Classes located on the Java class path are loaded once per server lifetime and can only be reloaded by stopping and restarting ColdFusion Server.

Disabling Automatic Reloading for Deployment

Automatic class reloading is an essential feature for iterative development and testing. However, because it must continually check to see whether Java CFX class files have changed, performance may decrease slightly. Therefore, when you move from development into deployment, Allaire ecommends that you globally disable automatic class reloading. You can do this by modifying the coldfusion.cfx.class.reload setting of the config/jvm.init file as follows:

```
coldfusion.cfx.class.reload=no
```

For additional details on modifying JVM configuration file settings, see "Java Customization and Configuration" on page 289.

Life cycle of Java CFXs

A new instance of the Java CFX object is created for each invocation of the Java CFX tag. This means that it is safe to store per-request instance data within the members of

your CustomTag object. If you wish to store data and/or objects that are accessible to all instances of your CustomTag you should use static data members.

Calling the CFX from a ColdFusion Template

You call Java CFXs from within ColdFusion templates by using the name of the CFX. The following CFML template calls the HelloColdFusion custom tag:

```
<HTML>
<BODY>
    <CFX_HelloColdFusion NAME="Les">
</BODY>
</HTML>
```

To test the CFX:

1. Create a new source file in your editor and enter the code displayed above.
2. Save the file in a directory configured to serve ColdFusion templates. For example, you might save the file as c:\inetpub\wwwroot\cfdocs\testjavacfx.cfm on Windows NT or /home/docroot/cfdocs/testjavacfx.cfm on UNIX.
3. Request the template from your web browser using the appropriate URL, for example

   http://localhost/cfdocs/testjavacfx.cfm

ColdFusion processes the template and returns a page that displays the text "Hello, Robert." If an error is returned instead, check the source code to make sure you have entered it correctly.

ZipBrowser Example

The following example illustrates the use of the Request, Response, and Query objects. The example uses the java.util.zip package to implement a Java CFX called ZipBrowser, which is a zip file browsing tag.

The fully qualified path of the zip archive to browse is specified using the ARCHIVE attribute. The name of the query to return to the calling page is specified using the NAME attribute. The query returned contains three columns: Name, Size, and Compressed.

For example, to query an archive at the path c:\logfiles.zip for its contents and to output the results you would use the following CFML code:

```
<CFX_ZipBrowser
      ARCHIVE="c:\logfiles.zip"
      NAME="LogFiles" >

<CFOUTPUT QUERY="LogFiles">
#Name#,  #Size#, #Compressed# <BR>
</CFOUTPUT>
```

The Java implementation of `ZipBrowser` is as follows:

```
import com.allaire.cfx.* ;
import java.util.Hashtable ;
import java.io.FileInputStream ;
import java.util.zip.* ;

public class ZipBrowser implements CustomTag
{
   public void processRequest( Request request, Response response )
      throws Exception
   {
      // validate that required attributes were passed
      if (  !request.attributeExists( "ARCHIVE" ) ||
            !request.attributeExists( "NAME" ) )
      {
         throw new Exception(
            "Missing attribute (ARCHIVE and NAME are both " +
            "required attributes for this tag)" ) ;
      }
     // get attribute values
      String strArchive = request.getAttribute( "ARCHIVE" ) ;
      String strName = request.getAttribute( "NAME" ) ;

    // create a query to use for returning the list of files
      String[] columns = { "Name", "Size", "Compressed" } ;
      int iName = 1, iSize = 2, iCompressed = 3 ;
      Query files = response.addQuery( strName, columns ) ;

    // read the zip file and build a query from its contents
      ZipInputStream zin =
         new ZipInputStream( new FileInputStream(strArchive) ) ;
      ZipEntry entry ;
      while ( ( entry = zin.getNextEntry()) != null )
      {
         // add a row to the results
         int iRow = files.addRow() ;

         // populate the row with data
         files.setData( iRow, iName,
            entry.getName() ) ;
         files.setData( iRow, iSize,
            String.valueOf(entry.getSize()) ) ;
         files.setData( iRow, iCompressed,
            String.valueOf(entry.getCompressedSize()) ) ;

         // finish up with entry
         zin.closeEntry() ;
      }

      // close the archive
      zin.close() ;
   }
}
```

Approaches to Debugging Java CFXs

Java CFXs are not standalone applications that run in their own process like typical Java applications. Rather, they are created and invoked from an existing process — ColdFusion Server. This makes debugging Java CFXs more difficult because it is not possible to use an interactive debugger to debug Java classes that have been loaded by another process.

To overcome this limitation, you can use one of two techniques:

- Debug the CFX while it is running within ColdFusion Server by outputting debug information as needed. See "Outputting Debug Information" on page 286 for details.
- Debug the request in an interactive debugger offline from ColdFusion Server using the special `com.allaire.cfx` debugging classes. See "Using the Debugging Classes" on page 286 for more information.

Outputting Debug Information

Before using interactive debuggers became the norm, programmers typically debugged their programs by inserting output statements in their programs to indicate information such as variable values and control paths taken. Often, when a new platform emerges, this technique comes back into vogue while programmers wait for more sophisticated debugging technology to be brought to the platform.

If you need to debug a Java CFX while running against a live production server, this is the technique you must use. In addition to simply outputting debug text using the `Response.write` method, you can also use the `DEBUG` attribute in your Java CFX. This attribute flags the CFX that the request is running in debug mode and therefore should output additional extended debug information. For example, to call the `HelloColdFusion` CFX in debug mode, you would use the following CFML code:

```
<CFX_HelloColdFusion" NAME="Robert" DEBUG="On">
```

To determine if a CFX has been invoked with the `DEBUG` attribute, you use the `Request.debug` method. To write debug output which will be printed in a special debug block after the tag finishes executing, you use the `Response.writeDebug` method. See the "Java CFX Reference" on page 311 for details on using these methods.

Using the Debugging Classes

To develop and debug Java CFXs in isolation from the ColdFusion Server, you use three special debugging classes that are included in the `com.allaire.cfx` package. These classes enable you to simulate a call to the `processRequest` method of your CFX within the context of the interactive debugger of a Java development environment. The three debugging classes are:

- `DebugRequest` — An implementation of the `Request` interface that enables you to initialize the request with custom attributes, settings, and a query.

- `DebugResponse` — An implementation of the `Response` interface that enables you to print the results of a request once it has completed.
- `DebugQuery` — An implementation of the `Query` interface that enables you to initialize a query with a name, columns, and a data set.

To use the debugging classes:

1. Create a `main` method for your Java CFX class. This method will be used as the testbed for your CFX.
2. Within the `main` method, initialize a `DebugRequest` and `DebugResponse`, and a `DebugQuery` if appropriate, with the attributes and data you want to use for your test.
3. Create an instance of your Java CFX and call its `processRequest` method, passing in the `DebugRequest` and `DebugResponse` objects.
4. Call the `DebugResponse.printResults` method to output the results of the request, including content generated, variables set, queries created, and so forth.

Once you have implemented a `main` method as described above, you can debug your Java CFX using an interactive, single-step debugger. Just specify your Java CFX class as the `main` class, set breakpoints as appropriate, and begin debugging.

Debugging Classes Example

The following example demonstrates the use of the debugging classes.

```
import java.util.Hashtable ;
import com.allaire.cfx.* ;

public class OutputQuery implements CustomTag
{
   // debugger testbed for OutputQuery
   public static void main(String[] argv)
   {
      try
      {
         // initialize attributes
         Hashtable attributes = new Hashtable() ;
         attributes.put( "HEADER",  "Yes" ) ;
         attributes.put( "BORDER", "3" ) ;

         // initialize query

         String[] columns =
            { "FIRSTNAME", "LASTNAME", "TITLE" } ;

         String[][] data =  {
            { "Stephen", "Cheng", "Vice President" },
            { "Joe", "Berrey", "Intern" },
            { "Adam", "Lipinski", "Director" },
            { "Lynne", "Teague", "Developer" } }  ;
```

```
            DebugQuery query =
               new DebugQuery( "Employees", columns, data ) ;


           // create tag, process debug request, and print results
            OutputQuery tag = new OutputQuery() ;
            DebugRequest request = new DebugRequest( attributes, query ) ;
            DebugResponse response = new DebugResponse() ;
            tag.processRequest( request, response ) ;
            response.printResults() ;
         }
         catch( Throwable e )
         {
            e.printStackTrace() ;
         }
      }

      public void processRequest( Request request ) throws Exception
      {
         // ...code for processing the request...
      }
}
```

Debugging Classes Reference

The specific constructors and methods supported by the `DebugRequest`, `DebugResponse`, and `DebugQuery` classes are as follows. Note that these classes also support the other methods of the `Request`, `Response`, and `Query` interfaces, respectively.

DebugRequest

```
// initialize a debug request with attributes
public DebugRequest( Hashtable attributes ) ;

// initialize a debug request with attributes and a query
public DebugRequest( Hashtable attributes, Query query ) ;

// initialize a debug request with attributes, a query, and settings
public DebugRequest( Hashtable attributes, Query query,
                     Hashtable settings ) ;
```

DebugResponse

```
// initialize a debug response
public DebugResponse() ;

// print the results of processing
public void printResults() ;
```

DebugQuery

```
// initialize a query with name and columns
public DebugQuery( String name, String[] columns )
   throws IllegalArgumentException ;

// initialize a query with name, columns, and data
public DebugQuery( String name, String[] columns, String[][] data )
    throws IllegalArgumentException ;
```

Java Customization and Configuration

You use the ColdFusion Administrator to customize your Java development environment, such as customizing the Class Path, Java system properties, and specifying an alternate JVM.

Implementing C++ CFX Tags

CFX tags built in C++ use the tag request object, represented by the C++ class CCFXRequest. This object represents a request made from an application page to a custom tag. A pointer to an instance of a request object is passed to the main procedure of a custom tag. The methods available from the request object allow the custom tag to accomplish its work. See the "C++ CFX Reference" on page 293 for a detailed description of the CFXAPI classes and members.

Implementing Java CFX Tags

Implementing a Java CFX requires interaction with the `Request` and `Response` objects passed to the `processRequest` method. In addition, CFXs that need to work with ColdFusion queries will also interface with the `Query` object. See the "Java CFX Reference" on page 311 for a detailed description of CFX Java object types.

Registering CFXs

To use a CFX tag in your ColdFusion applications, first register it in the Extensions, CFX Tags page in the ColdFusion Administrator.

To register the tag in the CF Administrator:

1. In the CF Administrator, open the Extensions > CFX Tags page.
2. Enter `CFX_MyNewTag` in the Tag name and, optionally, a description.
3. Select the type of tag (either C++ or Java).
4. Click Add to open the New CFX Tag page.

To register a C++ CFX:

1. In the CF Administrator, open the Extensions > CFX Tags page.
2. Enter `CFX_MyNewTag` in the Tag name and, optionally, a description.
3. Select the type of tag (either C++ or Java).
4. Click Add to open the New CFX Tag page.
5. If the Server library (DLL) field is empty, enter the file path.
6. Accept the default Procedure entry.
7. Un-check the Keep library loaded box while developing the tag. When the tag is ready for production use, you can check this option to keep the DLL in memory for improved performance.
8. Click Add.

You can now call the tag from a ColdFusion template.

To register a Java CFX:

1. In the CF Administrator, open the Extensions > CFX Tags page.
2. Enter `CFX_MyNewTag` in the Tag name and, optionally, a description.
3. Select the type of tag (either C++ or Java).
4. Click Add to open the New CFX Tag page.
5. Enter the Class name.
6. Click Add.

You can now call the tag from a ColdFusion template.

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.

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.

On **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:

Registration Entries for C++ CFXs	
Entry	**Value**
`Hive`	`HKEY_LOCAL_MACHINE`
`Key`	`SOFTWARE\Allaire\ColdFusion\CurrentVersion\CustomTags\`*`TagName`*
`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).

Registration Entries for Java CFXs	
Entry	**Value**
`Hive`	`HKEY_LOCAL_MACHINE`
`Key`	`SOFTWARE\Allaire\ColdFusion\CurrentVersion\CustomTags\`*`TagName`*
`ClassName`	The name of the Class to call.
`Description`	A description of the tag's functionality for browsing by end users.

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.

On Windows NT, use Regedit to import custom tags to the registry. The ColdFusion regedit utility (in the bin) performs the same function on UNIX.

To import a C++ custom tag:

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
```

To import a Java custom tag:

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]
"ClassName"="ProcessTagRequest"
"Description"="Sample CFX tag."
```

2. In the install script, import the registry entry by including the following command in the install script:

```
regedit importfilename
```

C++ CFX Reference

Below is a listing of CFXAPI classes and members. Individual members are described fully in the following sections.

Class	Members
CCFXException Class	CCFXException::GetError CCFXException::GetDiagnostics
CCFXQuery Class	CCFXQuery::AddRow CCFXQuery::GetColumns CCFXQuery::GetData CCFXQuery::GetName CCFXQuery::GetRowCount CCFXQuery::SetData CCFXQuery::SetQueryString CCFXQuery::SetTotalTime
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
CCFXStringSet Class	CCFXStringSet::AddString CCFXStringSet::GetCount CCFXStringSet::GetIndexForString CCFXStringSet::GetString

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 CCFXRequest Class, CCFXQuery Class, and CCFXStringSet Class. Your ColdFusion Extension code must therefore be written to handle exceptions of this type. (See the CCFXRequest::ThrowException and CCFXRequest::ReThrowException tags for details on doing this correctly.)

Class members

`virtual LPCSTR GetError()`

The CCFXException::GetError function returns a general error message.

`virtual LPCSTR GetDiagnostic()`

The CCFXException::GetDiagnostics function returns detailed error information.

CCFXException::GetError

This function provides basic user output for exception that occur during processing.

CCFXException::GetDiagnostics

This function provides detailed user output for exception that occur during processing.

Example

This code block shows how both functions work with ThrowException and ReThrowException.

```
// Write output back to the user here...
pRequest->Write( "Hello from CFX_FOO2!" ) ;
pRequest->ThrowException("User Error", "You goof'd...");

// Output optional debug info
if ( pRequest->Debug() )
{
    pRequest->WriteDebug( "Debug info..." ) ;
}
}

// Catch Cold Fusion exceptions & re-raise them
catch( CCFXException* e )
{
// This is how you would pull the error information
LPCTSTR strError = e->GetError();
LPCTSTR strDiagnostic = e->GetDiagnostics();
```

```
pRequest->ReThrowException( e ) ;
}

// Catch ALL other exceptions and throw them as
// Cold Fusion exceptions (DO NOT REMOVE! --
// this prevents the server from crashing in
// case of an unexpected exception)
catch( ... )
{
    pRequest->ThrowException(
        "Error occurred in tag CFX_FOO2",
        "Unexpected error occurred while processing tag." ) ;
}
}
```

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()`

CCFXQuery::AddRow adds a new row to the query.

`virtual CCFXStringSet* GetColumns`

CCFXQuery::GetColumns retrieves a list of the query's column names.

`virtual LPCSTR GetData( int iRow, int iColumn )`

CCFXQuery::GetData retrieves a data element from a row and column of the query.

`virtual LPCSTR GetName()`

CCFXQuery::GetName retrieves the name of the query.

`virtual int GetRowCount()`

CCFXQuery::GetRowCount retrieves the number of rows in the query.

`virtual void SetData( int iRow, int iColumn, LPCSTR lpszData )`

CCFXQuery::SetData sets a data element within a row and column of the query.

`virtual void SetQueryString( LPCSTR lpszQuery )`

CCFXQuery::SetQueryString sets the query string that will displayed along with query debug output.

`virtual void SetTotalTime( DWORD dwMilliseconds )`

CCFXQuery::SetTotalTime 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( iRow, iCity, "Minneapolis" ) ;
pQuery->SetData( iRow, iState, "MN" ) ;
pQuery->SetData( iRow, iZip, "55345" ) ;

// Second row
iRow = pQuery->AddRow() ;
pQuery->SetData( iRow, iCity, "St. Paul" ) ;
pQuery->SetData( iRow, iState, "MN" ) ;
pQuery->SetData( iRow, iZip, "55105" ) ;
```

CCFXQuery::GetColumns

```
CCFXStringSet* CCFXQuery::GetColumns(void)
```

Retrieves a list of the column names contained in the query.

Returns an object of CCFXStringSet Class that 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 freed automatically 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 query

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 *iColumn*)

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 CCFXQuery::GetRowCount. You can determine the number of columns in the query by retrieving the list of columns using CCFXQuery::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` *`iColumn`*`, 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 CCFXQuery::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

This is a deprecated function and should not be used.

CCFXQuery::SetTotalTime

This is a deprecated function and should not be used.

CCFXRequest Class

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 that 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 )`

CCFXRequest::AttributeExists checks to see whether the attribute was passed to the tag.

`virtual LPCSTR GetAttribute( LPCSTR lpszName )`

CCFXRequest::GetAttribute retrieves the value of the passed attribute.

`virtual CCFXStringSet* GetAttributeList()`

CCFXRequest::GetAttributeList retrieves a list of all attribute names passed to the tag.

`virtual CCFXQuery* GetQuery()`

CCFXRequest::GetQuery retrieves the query that was passed to the tag.

`virtual LPCSTR GetSetting( LPCSTR lpszSettingName )`

CCFXRequest::GetSetting retrieves the value of a custom tag setting.

`virtual void Write( LPCSTR lpszOutput )`

CCFXRequest::Write writes text output back to the user.

`virtual void SetVariable( LPCSTR lpszName, LPCSTR lpszValue )`

CCFXRequest::SetVariable sets a variable in the template that contains this tag.

`virtual CCFXQuery* AddQuery( LPCSTR lpszName, CCFXStringSet* pColumns )`

CCFXRequest::AddQuery adds a query to the template that contains this tag.

`virtual BOOL Debug()`

CCFXRequest::Debug checks whether the tag contains the `DEBUG` attribute.

`virtual void WriteDebug( LPCSTR lpszOutput )`

CCFXRequest::WriteDebug writes text output into the debug stream.

`virtual CCFXStringSet* CreateStringSet()`

CCFXRequest::CreateStringSet allocates and returns a new `CCFXStringSet` instance.

`virtual void ThrowException( LPCSTR lpszError, LPCSTR lpszDiagnostics )`

CCFXRequest::ThrowException throws an exception and ends processing of this request.

`virtual void ReThrowException( CCFXException* e )`

CCFXRequest::ReThrowException re-throws an exception that has been caught.

`virtual void SetCustomData( LPVOID lpvData )`

CCFXRequest::SetCustomData sets custom (tag specific) data to carry along with the request.

`virtual LPVOID GetCustomData()`

CCFXRequest::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 CFML tags (for example, `CFOUTPUT` or `CFTABLE`) within the template. Note that after calling `AddQuery`, the query exists but is empty (that is, it has 0 rows). To populate the query with data, you should call the CCFXQuery::AddRow and CCFXQuery::SetData functions.

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 freed automatically 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 that 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 instance. Note that string sets should always be created using this function as opposed to directly using the 'new' operator.

Returns an object of CCFXStringSet Class. You are not responsible for freeing the memory allocated for the returned string set (it will be freed automatically 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 CCFXRequest::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 CCFXRequest::GetAttribute.

Returns an object of class CCFXStringSet Class 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 freed automatically 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 CCFXRequest::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 CFQUERY tag or another custom tag). The QUERY attribute is optional and should only be used by tags that need to process an existing data set.

Returns an object of the CCFXQuery Class 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 freed automatically by ColdFusion after the request is completed).

Example

The following example retrieves the query that 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 pointer using CCFXRequest::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 that could throw an exception...

}
catch( CCFXException* e )
{
        ...Do appropriate resource cleanup here...

        // Re-throw the exception
        pRequest->ReThrowException( e ) ;
}
catch( ... )
{
        // Something nasty happened

        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

CCFXRequest::ReThrowException 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 CCFXRequest::Debug.)

`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" ) ;
}
```

CCFXStringSet Class

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 CCFXRequest::CreateStringSet.

Class members

`virtual int AddString( LPCSTR lpszString )`

CCFXStringSet::AddString adds a string to the end of the list.

`virtual int GetCount()`

CCFXStringSet::GetCount gets the number of strings contained in the list.

`virtual LPCSTR GetString( int iIndex )`

CCFXStringSet::GetString gets the string located at the passed index.

`virtual int GetIndexForString( LPCSTR lpszString )`

CCFXStringSet::GetIndexForString 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 CCFXStringSet::GetString 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 CCFXStringSet::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 CCFXStringSet::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>" ) ;
}
```

Java CFX Reference

Contents

Interface CustomTag

```
public abstract interface CustomTag
```

Interface for implementing custom tags.

Classes that implement this interface can be specified in the CLASS attribute of the Java CFX tag. For example, if I have a class `MyCustomTag` which implements this interface then the following CFML code could be used to call the MyCustomTag.processRequest method:

```
<CFX_MyCustomTag">
```

Additional attributes may also be passed to the Java CFX tag. The values of these attributes are available via the Request object passed to the processRequest method.

Method Summary		
`void`	processRequest(Request request, Response response)	Processes a request originating from the `CFX_mycustomtag` tag.

Method Detail

processRequest

```
public void processRequest(Request request,
                           Response response)
                    throws Exception
```

Processes a request originating from the Java CFX tag.

Parameters:

`request` — Parameters (attributes, query, etc.) for this request

`response` — Interface for generating response to request (output, variables, queries, etc.)

Throws:

`Exception` — If an unexpected error occurs while processing the request.

Interface Query

public abstract interface Query

Interface to a query used or created by a CustomTag. A query contains tabular data organized by named columns and rows

Method Summary		
`int`	`addRow()`	Adds a new row to the query.
`int`	`getColumnIndex(String name)`	Retrieves the index of a column given its name.
`String[]`	`getColumns()`	Retrieves a list of the column names contained in the query.
`String`	`getData(int iRow, int iCol)`	Retrieves a data element from a row and column of the query.
`String`	`getName()`	Retrieves the name of the query.
`int`	`getRowCount()`	Retrieves the number of rows in the query.
`void`	`setData(int iRow, int iCol, String data)`	Sets a data element within a row and column of the query.

Method Detail

getName

`public String getName()`

Retrieves the name of the query.

The following example retrieves the name of the query and writes it back to the user:

```
Query query = request.getQuery() ;
response.write( "The query name is: " + query.getName() ) ;
```

Returns:

The name of the query.

getRowCount

```
public int getRowCount()
```

Retrieves the number of rows in the query.

The following example retrieves the number of rows in a query and writes it back to the user:

```
Query query = request.getQuery() ;
int rows = query.getRowCount() ;
response.write( "The number of rows in the query is "
                  + Integer.toString(rows) ) ;
```

Returns:

The number of rows contained in the query.

getColumnIndex

```
public int getColumnIndex(String name)
```

Retrieves the index of a column given its name.

The following example retrieves the index of the EMAIL column and uses it to output a list of the addresses contained in the column:

```
// Get the index of the EMAIL column
int iEMail = query.getColumnIndex( "EMAIL" ) ;

// Iterate over the query and output list of addresses
int nRows = query.getRowCount() ;
for( int iRow=1; iRow<=nRows; iRow++ )
{
   response.write( query.getData( iRow, iEMail ) + "<BR>" ) ;
}
```

Parameters:

`name` — Name of column to get index of (lookup is case insensitive)

Returns:

The index of the column (returns -1 if no such column exists).

See Also:

`getColumns, getData`

getColumns

```
public String[] getColumns()
```

Retrieves a list of the column names contained in the query.

The following example retrieves the array of columns and then iterates over the list, writing each column name back to the user:

```
// Get the list of columns from the query
String[] columns = query.getColumns() ;
int nNumColumns = columns.length ;

// Print the list of columns to the user
response.write( "Columns in query: " ) ;
for( int i=0; i<nNumColumns; i++ )
{
   response.write( columns[i] + " " ) ;
}
```

Returns:

An array of strings containing the names of the columns in the query.

getData

```
public String getData(int iRow,
                      int iCol)
               throws IndexOutOfBoundsException
```

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 columns in the query by calling getColumns.

The following example iterates over the rows of the query and writes the data back to the user in a simple, space-delimited format:

```
int iRow, iCol ;
int nNumCols = query.getColumns().length ;
int nNumRows = query.getRowCount() ;
for ( iRow=1; iRow<=nNumRows; iRow++ )
{
   for ( iCol=1; iCol<=nNumCols; iCol++ )
   {
      response.write( query.getData( iRow, iCol ) + " " ) ;
   }
   response.write( "<BR>" ) ;
}
```

Parameters:

`iRow` — Row to retrieve data from (1-based)

`iCol` — Column to retrieve data from (1-based)

Returns:

The value of the requested data element.

Throws:

`IndexOutOfBoundsException` - If an invalid index is passed to the method.

See Also:

`setData, addRow`

addRow

```
public int addRow()
```

Adds a new row to the query. Call this method each time you want to append a row to the query.

The following example demonstrates the addition of 2 rows to a query that has 3 columns ('City', 'State', and 'Zip'):

```
// Define column indexes
int iCity = 1, iState = 2, iZip = 3 ;

// First row
int iRow = query.addRow() ;
query.setData( iRow, iCity, "Minneapolis" ) ;
query.setData( iRow, iState, "MN" ) ;
query.setData( iRow, iZip, "55345" ) ;

// Second row
iRow = query.addRow() ;
query.setData( iRow, iCity, "St. Paul" ) ;
query.setData( iRow, iState, "MN" ) ;
query.setData( iRow, iZip, "55105" ) ;
```

Returns:

The index of the row that was appended to the query.

See Also:

`setData, getData`

setData

```
public void setData(int iRow,
                    int iCol,
                    String data)
             throws IndexOutOfBoundsException
```

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.

The following example demonstrates the addition of 2 rows to a query that has 3 columns ('City', 'State', and 'Zip'):

```
// Define column indexes
```

```
int iCity = 1, iState = 2, iZip = 3 ;

// First row
int iRow = query.addRow() ;
query.setData( iRow, iCity, "Minneapolis" ) ;
query.setData( iRow, iState, "MN" ) ;
query.setData( iRow, iZip, "55345" ) ;

// Second row
iRow = query.addRow() ;
query.setData( iRow, iCity, "St. Paul" ) ;
query.setData( iRow, iState, "MN" ) ;
query.setData( iRow, iZip, "55105" ) ;
```

Parameters:

`iRow` — Row of data element to set (1-based)

`iCol` — Column of data element to set (1-based)

`data` — New value for data element

Throws:

`IndexOutOfBoundsException` — If an invalid index is passed to the method.

See Also:

`getData, addRow`

Interface Request

`public abstract interface Request`

Interface to a request made to a CustomTag. This interface includes methods for retrieving attributes passed to the tag (including queries) and reading global tag settings.

Method Summary		
`boolean`	`attributeExists(String name)`	Checks to see whether the attribute was passed to this tag.
`boolean`	`debug()`	Checks whether the tag contains the DEBUG attribute.
`String`	`getAttribute(String name)`	Retrieves the value of the passed attribute.

Method Summary		
String	getAttributeList()	Retrieves a list of all attributes passed to the tag.
int	getIntAttribute(String name)	Retrieves the value of the passed attribute as an integer.
int	getIntAttribute(String name, int def)	Retrieves the value of the passed attribute as an integer (returns default if the attribute does not exist or is not a valid number).
Query	getQuery()	Retrieves the query that was passed to this tag.
String	getSetting(String name)	Retrieves the value of a global custom tag setting.

Method Detail

attributeExists

```
public boolean attributeExists(String name)
```

Checks to see whether the attribute was passed to this tag.

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 ( ! request.attributeExists("DESTINATION") )
{
   throw new Exception(
      "Missing DESTINATION parameter",
      "You must pass a DESTINATION parameter in "
      "order for this tag to work correctly." ) ;
} ;
```

Parameters:

`name` — Name of the attribute to check (case insenstive)

Returns:

Returns `true` if the attribute is available otherwise returns `false`.

See Also:

`getAttribute, getAttributeList`

getAttribute

```
public String getAttribute(String name)
```

Retrieves the value of the passed attribute. Returns an empty string if the attribute does not exist (use attributeExists to test whether an attribute was passed to the tag). Use getAttribute(String,String) to return a default value rather than an empty string.

The following example retrieves an attribute named DESTINATION and writes its value back to the user:

```
String strDestination = request.getAttribute("DESTINATION") ;
response.write( "The destination is: " + strDestination ) ;
```

Parameters:

`name` — The attribute to retrieve (case insensitive)

Returns:

The value of the attribute passed to the tag. If no attribute of that name was passed to the tag then an empty string is returned.

See Also:

`attributeExists, getAttributeList, getAttribute(String,String), getIntAttribute`

getIntAttribute

```
public int getIntAttribute(String name)
                    throws NumberFormatException
```

Retrieves the value of the passed attribute as an integer. Returns -1 if the attribute does not exist. Throws a NumberFormatException if the attribute is not a valid number. Use attributeExists to test whether an attribute was passed to the tag. Use getIntAttribute(String,int) to return a default value rather than throwing an exception or returning -1.

The following example retrieves an attribute named PORT and writes its value back to the user:

```
int nPort = request.getIntAttribute("PORT") ;
if ( nPort != -1 )
   response.write( "The port is: " + String.valueOf(nPort) ) ;
```

Parameters:

`name` — The attribute to retrieve (case insensitive)

Returns:

The value of the attribute passed to the tag. If no attribute of that name was passed to the tag then -1 is returned.

Throws:

`NumberFormatException` — If the attribute is not a valid number.

See Also:

`attributeExists, getAttributeList, getIntAttribute(String,int)`

getAttributeList

```
public String[] getAttributeList()
```

Retrieves a list of all attributes passed to the tag. To retrieve the value of an individual attribute you should use the getAttribute member function.

The following example retrieves the list of attributes and then iterates over the list, writing each attribute and its value back to the user:

```
String[] attribs = request.getAttributeList() ;
int nNumAttribs = attribs.length ;

for( int i=0; i<nNumAttribs; i++ )
{
   String strName = attribs[i] ;
   String strValue = request.getAttribute( strName ) ;
   response.write( strName + "=" + strValue + "<BR>" ) ;
}
```

Returns:

An array of strings containing the names of the attributes passed to the tag.

See Also:

`attributeExists, getAttribute`

getQuery

```
public Query getQuery()
```

Retrieves the query that was passed to this 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 (e.g. created using the CFQUERY tag). The QUERY attribute is optional and should only be used by tags which need to process an existing dataset.

The following example retrieves the query which was passed to the tag. If no query was passed then an exception is thrown:

```
Query query = request.getQuery() ;
if ( query == null )
{
   throw new Exception(
      "Missing QUERY parameter. " +
      "You must pass a QUERY parameter in "
```

```
            "order for this tag to work correctly." ) ;
    }
```

Returns:

The Query that was passed to the tag. If no query was passed to the tag then null is returned.

getSetting

```
public String getSetting(String name)
```

Retrieves the value of a global custom tag setting. Custom tag settings are stored within the CustomTags section of the ColdFusion Registry key.

Note All custom tags implemented in Java share a single registry key for storing settings. This means that to avoid name conflicts you should preface the names of your settings with the name of your CustomTag class.

For example, the code below retrieves the value of a setting named 'VerifyAddress' for a CustomTag class named MyCustomTag:

```
  String strVerify = request.getSetting("MyCustomTag.VerifyAddress")
;
   if ( Boolean.valueOf(strVerify) )
   {
      // Do address verification...
   }
```

Parameters:

`name` — The name of the setting to retrieve (case insensitive)

Returns:

The value of the custom tag setting. If no setting of that name exists then an empty string is returned.

debug

```
public boolean debug()
```

Checks whether the tag contains the DEBUG attribute. You should use this method to determine whether or not you need to write debug information for this request (see Response.writeDebug for details on writing debug information).

The following example checks to see whether the DEBUG attribute is present, and if it is then it writes a brief debug message:

```
   if ( request.debug() )
   {
      response.writeDebug( "debug info" ) ;
   }
```

Returns:

Returns true if the tag contains the DEBUG attribute otherwise returns false.

See Also:

`Response.writeDebug`

Interface Response

`public abstract interface Response`

Interface to response generated from a CustomTag. This interface includes methods for writing output, generating queries, and setting variables within the calling page.

Method Summary		
`Query`	`addQuery(String name, String[] columns)`	Adds a query to the calling template.
`void`	`setVariable(String name, String value)`	Sets a variable in the calling template.
`void`	`write(String output)`	Outputs text back to the user.
`void`	`writeDebug(String output)`	Writes text output into the debug stream.

Method Detail

write

`public void write(String output)`

Outputs text back to the user.

The following example outputs the value of the DESTINATION attribute:

```
   response.write( "DESTINATION = " +
request.getAttribute("DESTINATION") ) ;
```

Parameters:

`output` — Text to output

setVariable

```
public void setVariable(String name,
                        String value)
                 throws IllegalArgumentException
```

Sets a variable in the calling template. If the variable name specified already exists in the template then its value is replaced. If it does not already exist then a new variable is created.

For example, this code sets the value of a variable named 'MessageSent' based on the success of an operation performed by the custom tag:

```
boolean bMessageSent ;

...attempt to send the message...

if ( bMessageSent == true )
{
   response.setVariable( "MessageSent", "Yes" ) ;
}
else
{
   response.setVariable( "MessageSent", "No" ) ;
}
```

Parameters:

`name` — The name of the variable to set

`value` — The value to set variable to

Throws:

`IllegalArgumentException` — If the `name` parameter is not a valid CFML variable name

addQuery

```
public Query addQuery(String name,
                      String[] columns)
               throws IllegalArgumentException
```

Adds a query to the calling template. This query can then be accessed by CFML tags 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 Query member functions addRow and setData.

The following example adds a Query named 'People' to the calling template. The query has two columns ('FirstName' and 'LastName') and 2 rows:

```
   // Create string array with column names (also track columns
indexes)
   String[] columns = { "FirstName", "LastName" } ;
   int iFirstName = 1, iLastName = 2 ;

   // Create a query which contains these columns
   Query query = response.addQuery( "People", columns ) ;

   // Add data to the query
   int iRow = query.addRow() ;
```

```
query.setData( iRow, iFirstName, "John" ) ;
query.setData( iRow, iLastName, "Smith" ) ;
iRow = query.addRow() ;
query.setData( iRow, iFirstName, "Jane" ) ;
query.setData( iRow, iLastName, "Doe" ) ;
```

Parameters:

`name` — The name of the query to add to the template

`columns` — The column names to be used in the query

Returns:

The Query that was added to the template.

Throws:

IllegalArgumentException - if the name parameter is not a valid CFML variable name

See Also:

`Query.addRow`, `Query.setData`

writeDebug

```
public void writeDebug(String output)
```

Writes text output into the debug stream. This text is only displayed to the end-user if the tag contains the DEBUG attribute (you can check for this attribute using the Request.debug member function).

The following example checks to see whether the DEBUG attribute is present, and if it is then it writes a brief debug message:

```
if ( request.debug() )
{
   response.writeDebug( "debug info" ) ;
}
```

Parameters:

`output` — The text to output

See Also:

`Request.debug`

CHAPTER 19

Using CFOBJECT to Invoke Component Objects

The CFOBJECT tag is used to invoke objects created by component technologies. This includes COM/DCOM, CORBA, Java, and EJB objects.

Contents

Component Object Overview

This section gives you some basic information on objects supported in ColdFusion and provides resources for further inquiry.

About COM

COM (Component Object Model) is a specification and a set of services defined by Microsoft to enable component portability, reusability, and versioning. DCOM (Distributed Component Object Model) is an implementation of COM for distributed services, allowing access to components residing on a network.

COM objects can reside locally or on any network node. Currently, COM is supported on Windows NT 3.51/4.0 and Windows 95/98.

Resources

To find out more about COM/DCOM, go to Microsoft's COM site

About CORBA

CORBA (Common Object Request Broker Architecture) is a specification for a distributed component object system defined by the Object Management Group (OMG). In this model, an object is an encapsulated entity whose services are accessed only through well-defined interfaces. The location and implementation of each object is hidden from the client requesting the services. ColdFusion supports CORBA 2.0 on both Windows and Unix.

Resources

The OMG site is the main Web repository for CORBA information.

The following are CORBA vendors:

- Inprise VisiBroker
- Iona Orbix
- ORBacus

About Java Objects

Java objects include any Java class available in the Class Path specified in the ColdFusion Administrator.

Invoking Component Objects

The CFOBJECT tag is used to create an instance of the object and other ColdFusion tags, such as CFSET and CFOUTPUT, are used to invoke properties (attributes), and methods (operations) on the object. An object created by CFOBJECT or returned by other objects is implicitly released at the end of the template execution.

Coding guidelines

The following coding practice is required (or recommended) when accessing the object. Assume that the NAME attribute in the CFOBJECT tag specified the value "obj", and that the object has a property called "Property", and methods called "Method1", "Method2", and "Method3".

To set a property:

```
<CFSET obj.property = "somevalue">
```

To get a property:

```
<CFSET value = obj.property>
```

Note that parentheses are not used on the right side of the equation for property-gets.

Calling methods

Object methods usually take zero or more arguments. Arguments can be sent by value ([in] arguments) or by reference ([out] and [in,out]). Arguments sent by reference usually have their value changed by the object. Some methods return values while others may not.

Methods with no arguments:

```
<CFSET retVal = obj.Method1()>
```

Note that parentheses are required for methods with no arguments.

Methods with one or more arguments:

```
<CFSET x = 23>
<CFSET retVal = obj.Method1(x, "a string literal")>
```

This method accepts one integer argument, and one string argument.

Methods with reference arguments:

```
<CFSET x = 23>
<CFSET retVal = obj.Method2("x",  "a string literal")>
<CFOUTPUT> #x#</CFOUTPUT>
```

Note the use of double-quotes ("") to specify reference arguments. If the object changes the value of "x", it will now contain a value other than 23.

Calling nested objects

The current release of ColdFusion does not support nested (scoped) object calls. For example, if an object method returns another object and you would like to invoke a property/method on that object, the following is required:

```
<CFSET objX = myObj.X>
<CFSET prop = objX.Property>
```

(That is, the syntax `<CFSET prop = myObj.X.Property>` will fail.)

Getting Started with COM/DCOM

ColdFusion is an automation (late-binding) COM client. This implies that the COM object has to support the IDispacth interface, and that arguments for methods and properties be standard automation types. Since ColdFusion is a typeless language, it uses the object's type information to correctly set up the arguments on call invocations. Any ambiguity in the object's data-types could lead to unexpected behavior.

It is important to use server-side COM objects in ColdFusion, that is, they should not have a graphical user interface. If you invoke an object with a graphical interface in your ColdFusion application, a window for the component might appear on the Web server desktop, not on the user's desktop. This could tie up ColdFusion server threads and result in further Web server requests not being serviced.

ColdFusion can call Inproc, Local, or remote COM objects. The attributes specified in the CFOBJECT tag determine which type of object is called.

Requirements for COM

To make use of COM components in your ColdFusion application, you need at least the following items:

- The Microsoft OLE/COM Object Viewer, available from Microsoft. It is a handy tool for viewing registered COM objects.
- The COM objects you want to use in your ColdFusion application pages. These are typically DLL or EXE files. These components should allow late binding, that is, they implement the IDispatch interface. Object Viewer allows you to view the object's class information so that you can properly define the CLASS attribute for the CFOBJECT tag. It also displays the object's supported interfaces, which allows you to discover the properties and methods (for the IDispatch interface) of the object.

Registering the object

Once you've acquired the object you want to use, you may need to register it with Windows 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 manual registration.

Inproc object servers (*.dll, *.ocx) can be registered manually by using the "regsvr32.exe" utility using the following form:

```
regsvr32 c:\path\servername.dll
```

Local servers (*.exe) are typically registered either by simply starting them or specifying a command line parameters like:

```
C:\pathname\servername.exe -register
```

Finding 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 Object Viewer to view the component's interface.

Using the OLE/COM Object Viewer

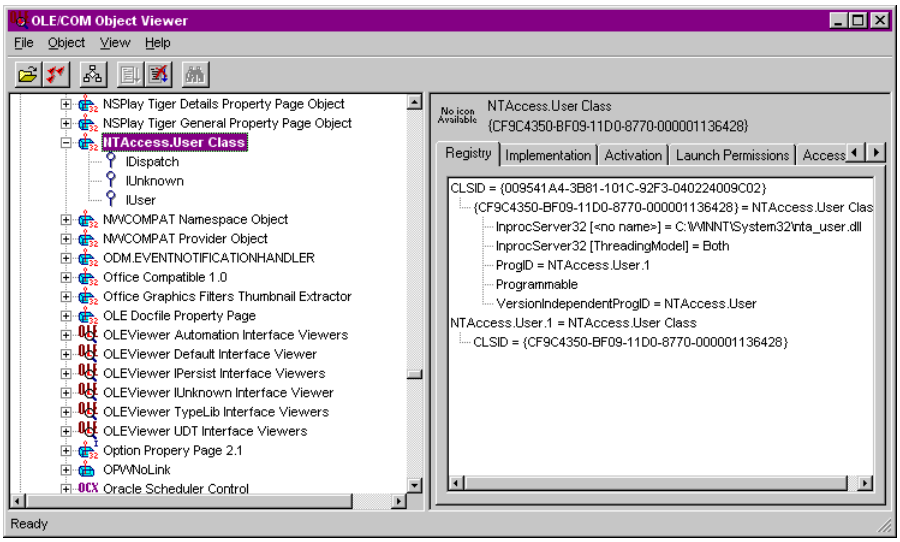
The OLE/COM Object Viewer installation installs the executable by default as `\mstools\bin\oleview.exe`. You use the Object Viewer to retrieve a COM object's Program ID as well as its methods and properties.

Once you've installed a COM object, make sure you register it using the `regsvr32.exe` utility. Otherwise you won't find the object in the Object Viewer. The Object Viewer retrieves all COM objects and controls from the Registry and presents the information in a simple format, sorted into groups for easy viewing.

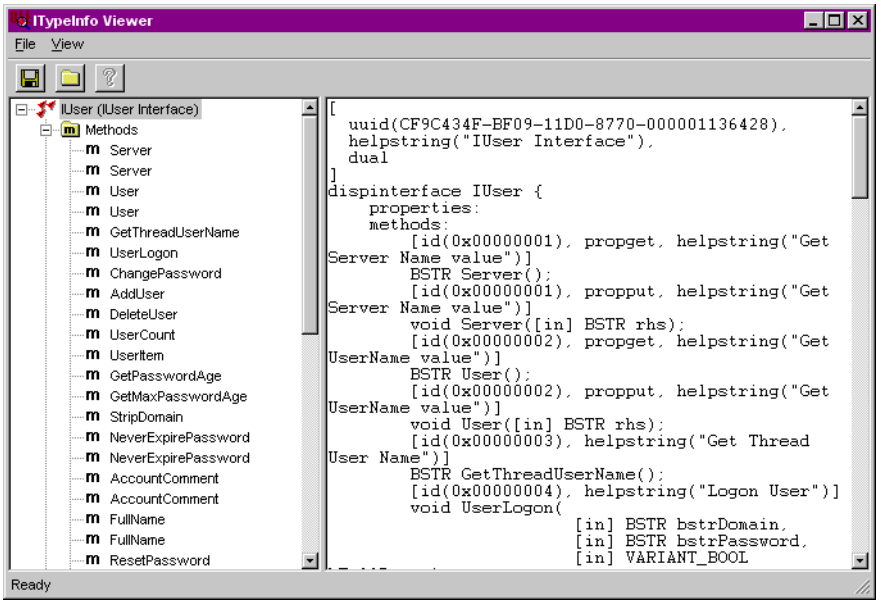
By selecting the category and then the component, you can see the Program ID of the COM object you want to use. The Object Viewer also gives you access to options for the operation of the object.

To view an object's properties:

1. Open the Object Viewer and scroll to the object you want to examine.



2. Select and expand the object in the Object Viewer.
3. Right-click the object to view 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.



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 sample SMTP component includes a number of methods and properties to perform a wide range of mail handling tasks.. 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 SMTP mail component includes properties such as:

```
Screen
User
FullName
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
```

Connecting to COM objects

There are essentially two ways, specified with the ACTION attribute of CFOBJECT, to connect to COM objects using CFOBJECT:

- The Create method (CFOBJECT ACTION="Create"), which takes a COM object, typically a DLL, and instantiates it prior to invoking methods and assigning properties.
- The Connect method (CFOBJECT ACTION="Connect"), which links to an object that is already running on the server, typically an executable.

In addition to specifying which way to connect to a COM object, you also have to specify the following with the CONTEXT attribute:

- INPROC — This means an In-Process server object (typically a DLL) that is running in the same process space as the calling process, such as ColdFusion.
- LOCAL — This is an Out-of-Process 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 Out-of-Process server object (also typically an EXE) that is running remotely on the network. Using REMOTE implies using the SERVER attribute to identify where the object resides.

Setting properties and invoking methods

The following example, using the sample SMTPMailer COM object, shows how to assign properties to the mail message you want to send and how to execute component methods 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()>
```

Getting Started with CORBA

ColdFusion supports CORBA through the Dynamic Invocation Interface (DII). As with COM, the object's type information has to be available to ColdFusion. This implies that an IIOP compliant Interface Repository (IR) should be running on the network, and that the object's IDL is registered in the IR.

ColdFusion Enterprise version 4.0 is bundled with deployment software from Inprise 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 Enterprise. This directory is called `vbroker\log` and its location is determined as follows:

- If VisiBroker is already installed on the server, the log directory is the directory pointed to by the `VBROKER_ADM` environment variable.
- 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` (UNIX), then the log directory will be `c:\vbroker\log` or `/vbroker` (UNIX).
- If the creation of the log directory on the root fails, then the directory is created in the ColdFusion installation directory.

Calling 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 accessible location.
- If the CONTEXT attribute is set to a NameService, the CLASS attribute must include a period-delimited name such as Allaire.Department.Dev. Currently ColdFusion can only resolve objects registered in VisiBrokers's Naming Service. This will change when ORB vendors implement the CORBA 3.0 specification. Make sure that the NamingService (NS) is brought-up with a default NamingContext. The server implementing the object should bind to the default context, and register the appropriate name. ColdFusion also binds to the default context to resolve the name.
- Set the NAME attribute to the name your application uses to call the object's operations and attributes.

See the *CFML Language Reference* for the complete CFOBJECT syntax.

Declaring structures and sequences

Once the object is created, attributes and operations on the object can be invoked using the syntax outlined in the above sections. In addition, ColdFusion also supports the use of complex types such as structures and sequences. For structures, use ColdFusion structures. For sequences, use ColdFusion arrays.

Example

The IDL for an object

```
struct SimpleStruct
```

```
{
  short s;
  long l;
  float d;
};

struct NestedStruct
{
  SimpleStruct f;
  char c;
  string s;
};

typedef sequence<long, 5> BLongSequence;

interface SomeObject {
  short SomeMethod( in NestedStruct inStruct,  in BlongSequence inSeq);

};
```

The applicable ColdFusion code

```
<!—Declare a couple of structures -‡
<CFSET x = StructNew()>
<CFIF IsStruct(x)>
    <CFSET temp=StructInsert(x,"s",3)>
    <CFSET temp=StructInsert(x,"l", 256)>
    <CFSET temp=StructInsert(x,"d", 93.45)>
</CFIF>

<CFSET NestedStruct = StructNew()>
<CFIF IsStruct(xx)>
    <CFSET temp=StructInsert(NestedStruct,"f",x)>
    <CFSET temp=StructInsert(NestedStruct,"c", 'b')>
    <CFSET temp=StructInsert(NestedStruct,"s", " bu-bu")>
</CFIF>

<!—Declare a sequence -‡
<CFSET FixedSeq = ArrayNew(1)>

<CFLOOP INDEX="LoopCount" FROM="1" TO="5">
    <CFSET FixedSeq [LoopCount] = #LoopCount#>
</CFLOOP>

<CFSET retA=obj.SomeMethod(NestedStruct, FixedSeq)>
```

Exception handling

Exceptions thrown by the CORBA object methods can be caught with the CFTRY and CFCATCH tags. However, no information can be extracted from the exception object in this release.

Calling Java Objects

The CFOBJECT tag can call any Java class that's available on the Class Path specified in the ColdFusion Administrator. For example:

```
<CFOBJECT Type="Java" Class="MyClass" Name = "myObj">
```

Although this loads the class, it doesn't create an instance object. Static methods and fields are accessible after the call to CFOBJECT.

To call the constructors explicitly, use the `init` method with the appropriate arguments. For example:

```
<CFSET ret=myObj.init(arg1, arg2)>
```

If you call a public method on the object without first calling the `init` method, the result will be an implicit call to the default constructor. Arguments and return values can be any valid Java type, for example simple arrays and objects. ColdFusion does the appropriate conversions when strings are passed as arguments, but not when they are received as return values.

Calling EJBs

To call an EJB, you use CFOBJECT to create and call all the appropriate objects. In the following example, it is assumed that the Weblogic JNDI is used to register and find EJBHome instances:

```
<CFOBJECT ACTION="Create"
        TYPE="Java"
        CLASS="weblogic/jndi/Environment"
        NAME="wlEnv">

<CFSET ctx=wlEnv.getInitialContext()>
<CFSET ejbHome=ctx.lookup("statelessSession.TraderHome")>
<CFSET trader=ejbHome.Create()>
<CFSET value=trader.shareValue(20,55.45)>
<CFOUTPUT>
    Share value=#value#
</CFOUTPUT>

<CFSET value=trader.remove()>
```

In this example, the CFOBJECT tag creates the Weblogic Environment object, which is then used to get the InitialContext. The context object is used to look up the EJBHome interface. The call to create() results in getting an instance of stateless session EJB.

Exception handling

Exceptions thrown by Java object methods can be caught by the CFTRY and CFCATCH tags. ColdFusion checks to see if the exceptin thrown is the method exception and stores the class name of the eception in the `message` field of the CFCATCH variable.

CHAPTER 20

Extending ColdFusion Pages with CFML Scripting

ColdFusion 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

The ColdFusion server-side scripting language, CFScript, 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 the *CFML Language Reference* for more information about 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 the *CFML Language Reference* .

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 *CFML Language Reference* 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 21

Accessing the Windows NT Registry

The CFREGISTRY tag gives you programmatic access to the Windows Registry.

Contents

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 uses to maintain hierarchical information about users, hardware, and software. Under UNIX, ColdFusion includes functionality that emulates the registry. 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*.

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 restrict the CFREGISTRY tag (this is especially true for ISPs, whose server may host a large and diverse set of developers).

Getting Registry Values

You can use CFREGISTRY with the GETaction to retrieve one entry, or the GETALL action 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, which 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 create 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.

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

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">
```

Index