# Developing ColdFusion Applications

MacroMedia ColdFusion® 5

Macromedia® Incorporated

# Copyright Notice

Part number: ZCF50MDEV

# Contents

# About This Book

*Developing ColdFusion Applications* describes the process of developing Web applications using ColdFusion. In the first eight chapters, you can follow the instructions presented to learn how to create basic ColdFusion applications. Then, chapters nine through 19 cover various topics of interest in enhancing your applications. Finally, chapters 20 through 23 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 Reference* for details about various tags and functions.

## Contents

## Intended Audience

*Developing ColdFusion Applications* is intended for Web application programmers who are learning ColdFusion orwish to extended their ColdFusion programming knowledge. It provides a solid grouding in the tools that ColdFusion provides to develop Web applications. The initial chapters provide e instructions for creating a basic ColdFusion application and are intended for those who are new to ColdFusion. Later chapters cover more specific features in greater detail and are intended for both new ColdFusion programmers and for those who are looking to extend existing skill.

## New Features

The following table lists the new features in ColdFusion 5:

| Benefit | Feature | Description |
|---|---|---|
| Breakthrough productivity | User-defined functions | Create reusable functions to accelerate development. |
| | Query of queries | Easily integrate data from heterogeneous sources by merging and querying data in memory using standard SQL. |
| | Server analysis and troublshooting | Quickly detect and diagnose server errors with built-in server reporting and the new Log File Analyzer. |
| Powerful business intelligence capabilities | Charting engine | Create professional-quality charts and graphs from queried data without leaving the ColdFusion environment. |
| | Enhanced Verity K2 full-text search | Index and search up to 250,000 documents and enjoy greater performance. |
| | Reporting interface for Crystal Reports 8.0 | Create professional-quality tabular reports from queried data and applications. |

| **Benefit** | **Feature** | **Description** |
|---|---|---|
| Enhanced performance | Core engine tuning | Take advantage of dramatically improved server performance and reduced memory usage to deliver faster, more scalable applications. |
| | Incremental page delivery | Improve response time by delivering page output to users as it is built. |
| | Wire protocol database drivers | Deliver high-performance ODBC connectivity using new drivers. |
| Easy managment | Application deployment services | Effortlessly and reliably deploy, archive, or restore entire applications using ColdFusion archive files. |
| | Enhanced application monitoring | Keep track of server performance and availability with customizable alerts and recovery. |
| | SNMP support | Monitor ColdFusion applications from enterprise management systems. |
| Expanded integration | Expanded Linux support | Deploy on additional Linux distributions, including SuSE and Cobalt. |
| | Enhanced hardware load balancer integration | Apply optimized, agent-based support for hardware load balancers, including new support for the Cisco CSS 11000. |
| | Enhanced COM support | Experience easier integration with COM components. |

## Developer Resources

Macromedia, Inc. is committed to setting the standard for customer support in developer education, technical support, and professional services. The Web site is designed to give you quick access to the entire range of online resources, as the following table describes.

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

## About ColdFusion Documentation

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

## Printed and online documentation set

The ColdFusion documentation set consists of the following titles.

| Book | Description |
|---|---|
| *Installing and Configuring ColdFusion Server* | Describes system installation and basic configuration for Windows NT, Windows 2000, Solaris, Linux, and HP-UX |
| *Advanced ColdFusion Server Administration* | Describes how to connect your data sources to the ColdFusion Server, configure security for your applications, and how to use ClusterCATS to manage scalability, clustering, and load-balancing for your site |
| *Developing ColdFusion Applications* | Describes on how to develop your dynamic Web applications, including retrieving and updating your data, using structures, and forms. |
| *CFML Reference* | The online-only *ColdFusion Reference* provides descriptions, syntax, usage, and code examples for all ColdFusion tags, functions, and variables. |
| *CFML Quick Reference* | A brief guide that shows the syntax of ColdFusion tags, functions, and variables |
| *Using ColdFusion Studio* | Describes how to use ColdFusion Studio to build, test, and deploy Web content, including using the built-in editor for a variety of scripting and markup languages |

## Viewing online documentation

All ColdFusion documentation is available online in HTML and Adobe Acrobat PDF formats. To view the HTML documentation, open the following URL on the Web server running ColdFusion: http://localhost/coldfusion/docs/dochome.htm.

To view and print ColdFusion documentation in Acrobat format, open the following URL on the Web server running ColdFusion: http://localhost/coldfusion/docs/AcrobatDocs/index.htm.

## Printing ColdFusion documentation

To read printed documentation, locate the Adobe Acrobat PDF files installed with the product. The PDF files offer excellent print output. You can print an entire book or individual sections.

Locate the ColdFusion PDF files by opening the following URL on the host system: http://localhost/coldfusion/docs/AcrobatDocs/index.htm

## Getting Answers

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

## Contacting Macromedia

### Corporate headquarters

Macromedia, Inc.
600 Townsend street
San Francisco, CA 4103

Tel: 415.252.2000
Fax: 415.626.0554

Web: www.macromedia.com

### Technical support

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

You can make postings to the ColdFusion Support Forum (http://forums.coldfusion.com/DevConf/index.cfm) at any time.

### Sales

Toll Free: 888.939.2545

Tel: 617.219.2100
Fax: 617.219.2101

E-mail: sales@macromedia.com

Web: http://commerce.coldfusion.com/purchase/index.cfm

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 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 e-commerce to managing internal business processes. For example, a static HTML page allows 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 allows customers to order books online, write reviews of books they 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 conduct the day-to-day business of your organization.

# Before You Begin

Before you begin using ColdFusion to create your Web applications, you should be familiar with HTML, relational database design and management, and Structured Query Language (SQL).

## HTML

You will 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 most likely have a separate table that lists each department number and name just once.

## SQL

Familiarity with some SQL is 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 two editions of ColdFusion: Enterprise and Professional. Using ColdFusion Enterprise or Professional Edition 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 failover, and visual cluster administration.

# ColdFusion Features and Components

ColdFusion provides a comprehensive set of features and components for developing and managing your Web applications. Using the ColdFusion components, you can enhance the speed and ease of development, dynamically deploy your applications, integrate new and legacy technologies, and build secure applications.

## About the features

The following table describes the ColdfFusion features that let you manage your Web site:

| **Benefits** | **Features** |
|---|---|
| Rapid development | • A tag-based server scripting language that is powerful and intuitive<br>• Two-way visual programming and database tools<br>• Remote interactive debugging for quickly identifying and fixing problems<br>• Web application wizards to automate common development tasks<br>• Source control integration to enable team development<br>• Secure file and database access using HTTP for remote development<br>• A tag-based component architecture for flexible code reuse |
| Scalable deployment | • A multithreaded service architecture that scales across processors<br>• Database connection pooling to optimize database performance<br>• Just-in-time page compilation and caching to accelerate page request processing<br>• Dynamic load balancing for scalable performance in a cluster environment (Enterprise Edition only)<br>• Automatic server recovery and failover for high availability (Enterprise Edition only) |

| Benefits | Features |
|---|---|
| Open integration | • Database connectivity using native database drivers (Enterprise Edition only), ODBC, or OLE DB<br>• Embedded support for full-text indexing and searching<br>• Standards-based integration with directory, mail, HTTP, FTP, and file servers<br>• Connectivity to distributed object technologies, including CORBA (Enterprise Edition only), COM (Windows Enterprise Edition only), Java objects and EJBs<br>• Open extensibility with C/C++ and Java |
| Security | • Integration with existing authentication systems, including Windows NT domain and LDAP directory servers, and proprietary user and group databases<br>• Advanced access control so that server administrators can control developers' access to files and data sources<br>• Support for existing database security<br>• Server sandbox security for protecting multiple applications on a single server (Enterprise Edition only)<br>• Support for existing Web server authentication, security, and encryption |

For detailed information about security, see *Advanced ColdFusion Administration*.

For the latest publications from Macromedia on security, visit the Security Zone at http://www.coldfusion.com/developer/securityzone/.

For a complete feature list and more detailed information, see the ColdFusion product pages at http://www.coldfusion.com/coldfusion.

## About the components

ColdFusion applications rely on several core components:

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

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 application pages

ColdFusion application pages (often called templates) look somewhat like HTML pages, but are much more dynamic and powerful. They 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 (CFML) tags, HTML tags, CFScript, JavaScript, and anything else that you can normally embed in an ordinary HTML page. You can easily access data sources, such as relational databases, from your application pages. 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 e-mail. 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, see *Installing and Configuring 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

For details on using the Administrator, see *Advanced ColdFusion Administration*.

## Data sources

ColdFusion applications can 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). You can also retrieve data from mail servers that support the Post Office Protocol (POP), and index the data 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 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, and routes the data through the Web server, back to your browser.

The following steps describe in more detail what happens when you open a ColdFusion page:

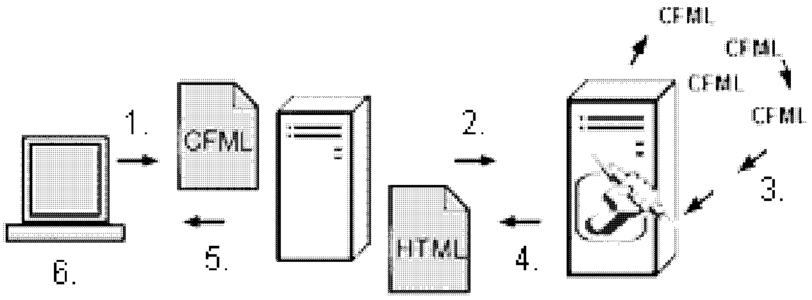
1 The client requests a page that contains CFML tags.

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:

1 Write some code.

2 Save the code to a document or page.

3 View the page in a browser.

4 Modify the page.

5 Save the page again.

6 View it in a browser.

## Working with ColdFusion Application Pages

While you can code your application pages using NotePad or any HTML editor, this book uses ColdFusion Studio because it provides many features that make ColdFusion development easier. You should install ColdFusion Studio if you have not done so already.

### About applicaton pages

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 is 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.

### Creating application pages

The following procedure creates a simple ColdFusion Application page, which you use for other examples in this chapter.

#### 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 follows:

```
<html>
<head>
<title>Call Department</title>
</head>
<body>
<strong>Call Department</strong><br>
```

```
<!--- Set all variables --->
<cfset department="Sales">
<!--- Display results --->
<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 can 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 might be:

(on Windows NT) c:\inetpub\wwwroot\myapps

(on UNIX) <mywebserverdocroot>/myapps

## 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 is returned to the browser.

| Original ColdFusion page | HTML file returned by Web server |
|---|---|
| `<html>` | `<html>` |
| `<head>` | `<head>` |
| `<title>Call Department</title>` | `<title>Call Department</title>` |
| `</head>` | `</head>` |
| `<body>` | `<body>` |
| `<strong>Call Department</strong><br>` | `<strong>Call Department</strong><br>` |
| `<!--- Set all variables --->` | |
| `<cfset department="Sales">` | |
| `<!--- Display results --->` | |
| `<cfoutput>` | |
| `I’d like to talk to someone in` | `I’d like to talk to someone in Sales.` |
| `#Department#.` | |
| `</cfoutput>` | `</body>` |
| `</body>` | `</html>` |
| `</html>` | |

## Reviewing the code

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 “Sales.” You then used the CFML tag `cfoutput` to display text and the value of the variable. The following table describes the code and its function:

| 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 example, the text “I'd like to talk to someone in” is followed by the value of the variable Department, which is “Sales.” |

# Working with 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 you can use in applications. As with other programming languages, you set variables in ColdFusion to store data that you want to access later. You reference a range of variables to perform different types of application processing.

## About variables

ColdFusion variable names are case-insensitive. The variable names CITY and city refer to the same data.

The kind of information that variables contain varies. Two characteristics distinguish the information in a variable:

- Data type
- Scope type

### Data types

A variable's **data type** specifies the kind of value a variable can represent, such as a text string or number. ColdFusion does not require you to specify a variable's data type. Whether a variable represents a string, a number, a Boolean value (Yes/No), a date and time, or a more complex object such as an array or structure, ColdFusion automatically uses the appropriate internal data representation when you assign its value. However, ColdFusion does provide methods to examine and change the type of data that a variable represents. For a complete list of data types see the *CFML Reference*.

For example, use the following syntax to create a string variable:

```
<cfset mystring="Hello world">
```

The following example uses scientific notation to create a floating-point numeric variable:

```
<cfset myfloat=1.296e-3>
```

### Scope types

Variables differ in the source the data came from, the places in your code where they are meaningful, and how long their values persist. These considerations are generally referred to as a variable's **scope**.

ColdFusion has many different scope types, which are identified by prefixes to a variable name. For example, the variable `Department` in calldept.cfm is a local variable (a variable that has meaning on the current page). Local variables have the optional prefix Variables. Instead of writing:

```
I'd like to talk to someone in #Department#.
```

you can write:

```
I’d like to talk to someone in #Variables.Department#.
```

Some variable scopes, such as the local scope, do not require the scope identifier prefix, while others do. However, it is good programming practice to use prefixes for most or all scopes. This helps to better identify each variable’s use and can prevent multiple uses of the same name. This book uses the scope prefix for all variables except for local variables.

The following table lists some of the more common types of variable scopes and the prefixes that you use to identify the variables. Other chapters in this book discuss additional scope types. The *CFML Reference* has a complete list of scope types, their identifiers, and how they are used.

| Scope type | Prefix | Description |
|---|---|---|
| Local (or Variables) | Variables | Variables created using cfset or cfparam, with or without specifying the scope prefix. You must define the variable on the current page or a page you include using `cfinclude`. |
| Form | Form | Data entered in tags in an HTML form or ColdFusion `cfform` tag block and processed on an action page. |
| URL | URL | Variables passed to a page as URL query string parameters. |

## Using the pound sign (#)

You surround a ColdFusion variable or function with pound signs (#) to tell the ColdFusion Server that it is not plain text. You only need to use pound signs in limited circumstances, particularly in the `cfoutput` and `cfquery` tag blocks. You do not need to use pound signs when you create a variable, assign it a value, or use it in a ColdFusion expression or as a parameter in a ColdFusion function.

**Note**
Remember that ColdFusion cannot interpret anything, including variables, that is not inside a ColdFusion tag or tag block.

The following table illustrates the basic use of pound signs. For a detailed description of the use of pound signs, see *CFML Reference*.

| CFML code | Results |
|---|---|
| `cfset Department="Sales">` | The variable named Department is created and the value is set to Sales. |
| `<cfoutput>`<br>`I’d like to talk to someone in Department.`<br>`</cfoutput>` | ColdFusion does not treat Department as a variable because it is not surrounded by pound signs. The HTML page displays:<br>I’d like to talk to someone in Department. |

| CFML code | Results |
|---|---|
| `<cfoutput>`<br>`I’d like to talk to someone in`<br>`#Department#.`<br>`</cfoutput>` | ColdFusion replaces the variable Department with its value. The HTML page displays:<br>I’d like to talk to someone in Sales. |
| `<cfoutput>`<br>`The department name spelled`<br>`backward is Reverse(Department).`<br>`</cfoutput>` | ColdFusion sees Reverse(Department) as text and displays it unchanged. The HTML page displays:<br>The department name spelled backward is Reverse(Department). |
| `<cfoutput>`<br>`The department name spelled`<br>`backward is #Reverse(Department)#.`<br>`</cfoutput>` | ColdFusion uses the `Reverse` function to reverse the text in the Department variable and displays the result. The pound signs tell `cfoutput` to interpret `Reverse` as a ColdFusion function. The `Reverse` function uses the Department variable name. The HTML page displays:<br>The department name spelled backward is selaS. |

## 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 Open the file calldept.cfm in ColdFusion Studio,.

2 Modify the code so that it appears as follows:

```
<html>
<head>
<title>Call Department</title><br>
</head>
<body>
<strong>Call Department</strong><br>
<!--- 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 --->`

  ColdFusion removes CFML comments from the HTML that it sends to the browser, so users do not see them if they view the HTML source. ColdFusion does send HTML comments to the browser.
- Filenames should be all one word, begin with a letter, and can contain only letters, numbers, and the underscore.
- Filenames should not contain special characters.
- Some operating systems are case-sensitive, so you should be consistent with your use of capital letters in filenames.

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 drives your applications, and for this discussion a **data source** means the entry point from ColdFusion to a database.

In this chapter, you build a query to retrieve data from the `CompanyInfo` data source, which accesses a Microosft Access database (company.mdb), on Windows systems or a DBase database on UNIX systems. In subsequent chapters in this book, you will insert and update data in this database.

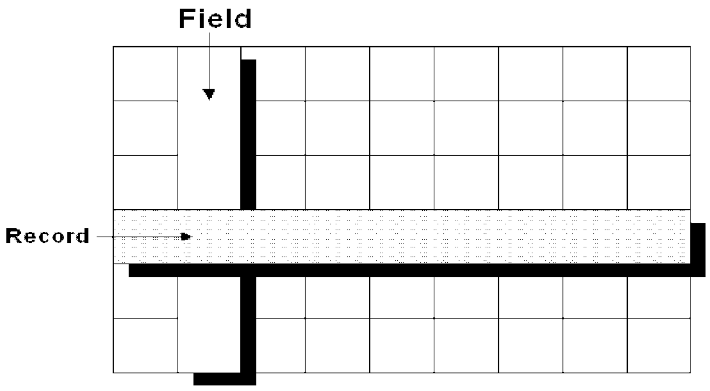
To query a database, you need to use:

- ColdFusion data sources
- The `cfquery` tag
- SQL commands

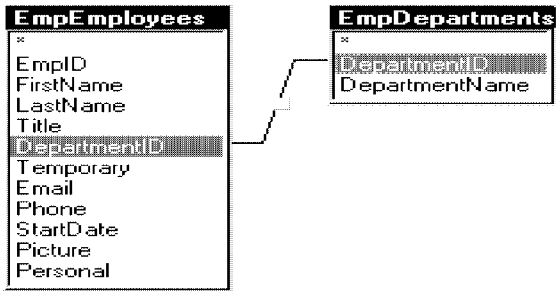
## Understanding Database Basics

You do not need a thorough knowledge of databases to develop a data-driven ColdFusion application, but you need to know 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**. In this case, each row is unique because it applies to one individual. Rows are also referred to as records. Columns are also referred to as **fields**.



You can organize data 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.



### Database design guidelines

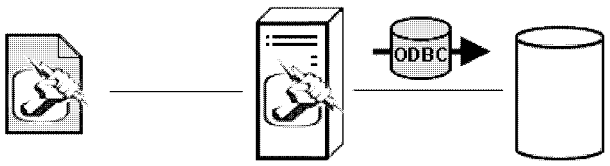
From this basic description, a few database design rules emerge:

- Each record should contain a unique identifier, known as the **primary key**. This can be an employee ID, a part number, or a customer number. The primary key is typically the column used to maintain each record's unique identity among the tables in a relational database.
- After you define a column to contain a specific type of information, you must enter data in that column in a consistent way.
- To enter data consistently, you define 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 database management system (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 defines the properties which ColdFusion uses to connect to a specific database. You add data sources to your ColdFusion Server so that you can connect to the databases from your ColdFusion applications.



## About Open Database Connectivity

Open Database Connectivity (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
- Borland Paradox Databases
- Informix databases
- Progress databases
- Oracle 8 databases
- Centura SQLBase databases
- Sybase ASE databases
- Delimited text files

You can also use any additional ODBC drivers that are installed on your system.

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

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

## Accessing Data Sources

There are two ways to access data sources:

- Add data sources in the ColdFusion Administrator.

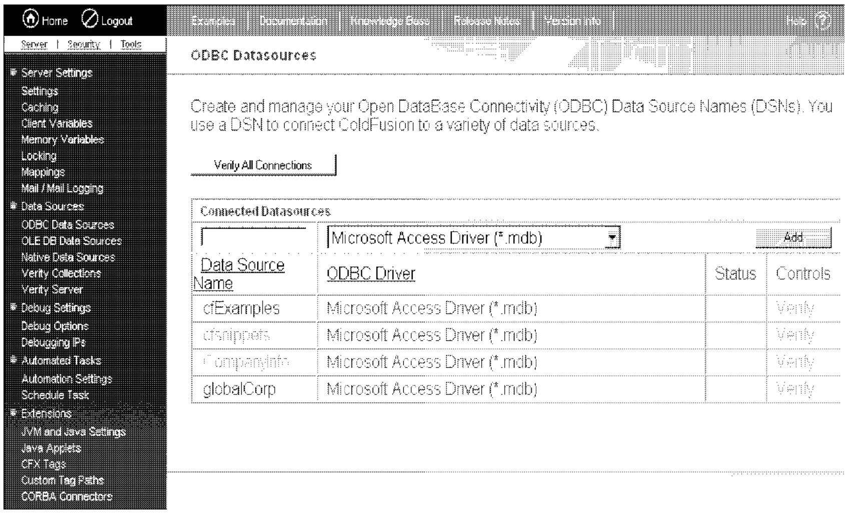
  You assign a data source name and set all the information required to establish an ODBC connection. You then use the data source name in any CFML tags that establish database connections. This technique puts all the information about a ColdFusion Server's database connections in a single, easy-to-manage location.

- Specify the database information directly in your CFML tag.

  This way you accesses the data source dynamically. It eliminates the need for you to add a data source for each database on your server. It also allows a ColdFusion application to run on multiple servers without having to statically configure each server independently.

## Adding data sources

You use the ColdFusion Administrator to add data sources to the ColdFusion Administrator.



ODBC Datasources

Create and manage your Open DataBase Connectivity (ODBC) Data Source Names (DSNs). You use a DSN to connect ColdFusion to a variety of data sources.

Connected Datasources

|  | Microsoft Access Driver (*.mdb) |  |  |
|---|---|---|---|
| Data Source Name | ODBC Driver | Status | Controls |
| cfExamples | Microsoft Access Driver (*.mdb) |  | Verify |
| cfsnippets | Microsoft Access Driver (*.mdb) |  | Verify |
| CompanyInfo | Microsoft Access Driver (*.mdb) |  | Verify |
| globalCorp | Microsoft Access Driver (*.mdb) |  | Verify |

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.

Use the following procedure to add the CompanyInfo data source that you use in many examples in this book.

---

**Note**
By default, the ColdFusion setup installs the CompanyInfo and cfsnippets databases used in examples in this book and adds them to the available ODBC data sources. Therefore, this procedure should not be necessary to work with examples in this book.

---

### 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 the installation.

2. Enter a password to gain access to the Administrator.
3. Select ODBC under the Data Sources heading on the left menu.

4 Name the data source `CompanyInfo`.

5 On Windows Select Microsoft Access Driver (*.mdb) from the drop-down box to describe the ODBC driver. On UNIX, select the Merant Dbase/FoxPro driver.

6 Click Add.

7 In the Database File field, enter the full path of the database. (You can also use the Browse button to locate the file.).

On Windows specify the path to the company.mdb file, typically C:\CFusion\database\Company.mdb. On UNIX, specify the path to the CompanyInfo directory, typically /opt/coldfusion/database/CompanyInfo.

8 Click 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 Select 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.
```

11 Click Go Back to return to the data sources list.

For more information about managing data sources, see *Advanced ColdFusion Administration*.

## Specifying a connection string

You can dynamically override ODBC connection information that you set in the ColdFusion Administrator. You can also specify connection attributes that are not set in the Administrator. To do so, use the `connectstring` attribute in any CFML tag that connects to a database: `cfquery`, `cfinsert`, `cfupdate`, `cfgridupdate`, and `cfstoredproc`.

For example, the following code creates a connection to a defined Microsoft SQLServer data source using a connect string to specify the Application and Work Station ID.

```
<cfupdate datasource = "mssql"
  connectstring = "APP=ColdFusion;WSID=fenway"
  tablename = "department">
```

**Note**
Connect string properties are specific to the database you are connecting to. See the documentation for your database for a list of connect string properties.

### Adding 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 (unless you specify the data source dynamically).

## Specifying data sources dynamically

To specify a data source dynamically, use the following attribute in the `cfquery` tag:

```
dbtype = "dynamic"
```

Specify all the required ODBC connection information, including the ODBC driver and the database location, in the `connectstring` attribute. For example, you could use the following code for a query that dynamically specifies the pubs database on a local Microsoft SQLServer:

```
<cfquery name = "datelist"
  dbtype = "dynamic"
  blockfactor = 100
  connectstring = "DRIVER={SQLSERVER};SERVER=(local);UID=sa;PWD=;
        DATABASE=pubs">
  SELECT * FROM authors
</cfquery>
```

The following example uses a Microsoft Access database:

```
<cfquery name="titles"
  dbtype = "dynamic"
  ConnectString="DRIVER=Microsoft Access Driver (*.mdb);
        DBQ=C:\CFusion\Database\cfsnippets.mdb;DriverId=281;
        FIL=MS Access;MaxBufferSize=2048;PageTimeout=5">
  SELECT * FROM Courses
</cfquery>
```

# Retrieving Data

You can query databases to retrieve data at runtime. The retrieved data, called the **result set**, is stored on that page as a query object. When retrieving data from a database, perform the following tasks:

- Use the `cfquery` tag on a page to tell ColdFusion how to connect to a database.
- Write SQL commands inside the `cfquery` block to specify the data that you want to retrieve from the database.
- Later on the page, reference the query object and use its data values in any tag that presents data, such as `cfoutput`, `cfgrid`, `cftable`, `cfgraph`, or `cftree`.

## The cfquery tag

The `cfquery` tag is one of the most frequently used CFML tags. You use it 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 result set values to the page in a special kind of variable called a **query object**. You specify the query object's name in the `cfquery` tag's `name` attribute. Often, we refer to the query object simply as "the query".

## The 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 object EmpList.

Follow these rules when creating a `cfquery` tag:

- 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 object 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. Alternatively, use the `dbtype = "dynamic"` and `connectString` attributes to dynamically specify a 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.

## 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 that you want to retrieve.

When the database processes the SQL, it creates a **data set** (a structure containing the requested data) 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 `name` attribute of the `cfquery` tag.

You can reference that data set by name using the `cfoutput` tag later on the page.

## Basic SQL syntax elements

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

### Statements

A SQL statement always begins with a SQL verb. The following keywords identify commonly used SQL verbs:

| Keyword | Description |
|---|---|
| SELECT | Retrieves the specified records |
| INSERT | Adds a new row |
| UPDATEw | Changes values in the specified rows |
| DELETE | Removes the specified rows |

### Statement clauses

Use the following keywords to refine SQL statements:

| Keyword | Description |
|---|---|
| FROM | Names the data tables for the operation |
| WHERE | Sets one or more conditions for the operation |
| ORDER BY | Sorts the result set in the specified order |
| GROUP BY | Groups the result set by the specified select list items |

### Operators

The following basic operators specify conditions and perform logical and numeric functions:

| Operator | Description |
|---|---|
| AND | Both conditions must be met |
| OR | At least one condition must be met |
| NOT | Exclude the condition following |
| LIKE | Matches with a pattern |
| IN | Matches with a list of values |
| BETWEEN | Matches with a range of values |
| = | Equal to |
| <> | Not equal to |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| + | Addition |
| - | Subtraction |
| / | Division |
| * | Multiplication |

## SQL notes and considerations

When writing SQL in ColdFusion, keep the following guidelines in mind:

- There is a lot more to SQL than what is covered here. It is a good idea to purchase one or several SQL guides 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 nonstandard 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 nonstandard 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 in ColdFusion Studio.

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 Employee
</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 might 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 the source in the browser.

The ColdFusion Server creates the EmpList data set, but only HTML and text is sent back to the browser so you just see the heading “Employee List”. To display the data set on the page, you must code tags and variables to output the data.

### Reviewing the code

The query you just created retrieves data from the CompanyInfo database. The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfquery name="EmpList" datasource="CompanyInfo">` | Queries the database specified in the CompanyInfo data source |
| `SELECT FirstName, LastName, Salary, Contract FROM Employee` | Gets information from the FirstName, LastName, Salary, and Contract fields in the Employee table |
| `</cfquery>` | Ends the `cfquery` block |

## Query notes and considerations

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

- Enter the query `name` and `datasource` attributes in the begin `cfquery` tag.
- Surround attribute settings with double quotes(").
- Make sure that a data source exists in the ColdFusion Administrator before you reference it n a `cfquery` tag. Alternatively, use the `dbtype = "dynamic"` and `queryString` attributes to dynamically specify a database.
- 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.
- Reference the query data by naming the query in one of the presentation tags, such as `cfoutput`, `cfgrid`, `cftable`, `cfgraph`, or `cftree` later on the page.

## Outputting Query Data

After you define a query on a page, you can use the `cfoutput` tag with the `query` attribute to specify the query object that contains the data 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.
- You must reference specific column names within the `cfoutput` block to output the data to the page.
- You can place text, CFML tags, and HTML tags inside or surrounding the `cfoutput` block to format the data on the page.
- You do not have to specify the query object name when you refer to a query column. For example, if you specify the Emplist query in your `cfoutput` tag, you can refer to the Firstname column in the Emplist query as either Emplist.Firstname or just Firstname.

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

### To output query data on your page:

1 Return to `empList.cfm` in ColdFusion 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 Employee
</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 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.

### Reviewing the code

You now display the results of the query on the page. The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfoutput query="EmpList">` | Display information retrieved in the EmpList query. Display information for each record in the query, until you run out of records. |
| `#FirstName#, #LastName#, #Salary#, #Contract#` | Display the value of the `FirstName`, `LastName`, `Salary`, `Contract` fields of each record, separated by commas and spaces. |
| `<br>` | Insert a line break (go to the next line) after each record. |
| `</cfoutput>` | End the `cfoutput` block. |

## Query output notes and considerations

When outputting query results, keep the following guidelines in mind:

- A `cfquery` must precede the `cfoutput` that references its results. Both must be on the same page (unless you use the `cfinclude tag`).
- It is a good idea to run all queries before all output blocks.
- To output data from all the records of a query, specify the query name by using the `query` attribute in the `cfoutput` tag.
- Columns must exist and be retrieved to the application in order to output their values.
- Inside a `cfoutput` block that uses a `cfquery` attribute you can optionally prefix the query variables with the name of the query, for example `Emplist.FirstName`.
- As with other attributes, surround the `query` attribute value with double quotes (").
- As with any variables that you reference for output, surround column names with pound signs (#) to tell ColdFusion to output the column’s current values.
- Add a `<br>` tag to the end of the variable references so that ColdFusion starts 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:

| Property | Description |
|---|---|
| RecordCount | The total number of records returned by the query. |
| ColumnList | 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 ColdFusion 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 Employee
</cfquery>
<cfoutput query="EmpList">
  #FirstName#, #LastName#, #Salary#, #Contract#<br>
</cfoutput>
<br>
<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.

**Note**
The variable `cfquery.executionTime` contains the amount of time, in milliseconds, it took for the query to complete. Do not prefix the variable name with the query name.

### Reviewing the code

You now display the number of records retrieved in the query. The following table describes the code and its function:

| Code | Description |
|---|---|
| `<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

When using query properties, keep the following guidelines in mind:

- Reference the query property within a `cfoutput` block so that ColdFusion outputs the query property 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.
- Do not use the `cfoutput` tag `query` attribute when you output the `RecordCount` or `ColumnList` property. If you do, you will get one copy of the output for each row. Instead, prefix the property with the name of the query.

## Using Query Results in Queries

ColdFusion allows you to use the results of a previous query in any `cfquery` tag that returns row data to ColdFusion. You can query a database once and use the results in several dependent queries. Queries generated from other queries are often referred to as **query of queries**.

### Query of query benefits

Performing queries on query results has many benefits:

- If you need to access the same tables multiple times, you greatly reduce access time for tables with up to 10,000 rows because the data is already in memory.
- You can join and perform unions on results from different data sources.

  For example, you can do a union on queries from different databases to eliminate duplicates for a mailing list.
- You can efficiently manipulate cached query results in different ways. You can query a database once, and then use the results to generate several different summary tables.

  For example, if you need to summarize the total salary by department, by skill, and job, you can make one query to the database and use its results in three separate queries to generate the summaries.
- You can make drill-down, master-detail-like functionality where you do not go to the database for the details.

  For example, you can select information about departments and employees in a query and cache the results. You can then display the employee names. When users select an employee, the application displays the employee details by selecting information from the cached query without accessing the database.

### Creating queries of queries

You can create a query using a query object from any ColdFusion tag or function that generates query results, including `cfldap`, `cfdirectory`, `chttp`, `cfstoredproc`, `cfpop`, `cfindex`, and the `Query` functions.

You can use a limited subset of the SQL SELECT syntax, which includes:

| | |
|---|---|
| FROM | WHERE |
| GROUP BY | UNION |
| ORDER BY | HAVING |
| AS | DISTINCT |

Boolean predicates:

```
LIKE
NOT LIKE
IN
NOT IN
BETWEEN
NOT BETWEEN
AND
OR
```

Comparison operators:

```
<=
>=
=
<
>
<>
```

Aggregate functions:

```
Count([DISTINCT][*] expr)
Sum([DISTINCT] expr)
Avg([DISTINCT] expr)
Max(expr)
Min(expr)
```

You cannot nest aggregate functions.

You can also do the following tasks:

- Use the results of one or two queries in your SQL statement.
- Generate computed columns.

## Performing a query on a query

To generate a query using an existing query:

- Specify the `cfquery tag’s` `dbtype` attribute as `"query"`.
- Do not specify a `datasource` attribute.
- Specify the names of one or more existing queries as the table names in a SQL SELECT statement.
- If the database content does not change rapidly, it is a good idea to use the `cachedwithin` attribute to cache the query results of between page requests. This way, ColdFusion accesses the database on the first page request, and does not query the database again until the specified time expires. Note that you must use the `CreateTimeSpan` function to specify the `cachedwithin` attribute value.

**Note**
You cannot add a literal value as a column to the SELECT list in a query of queries.

Your query generates a new query results set, identified by the value of the `name` attribute. The following example illustrates the use of a master query and a single detail query that extracts information from the master. A more extended example would use multiple detail queries to get different information from the same master query.

### To use the results of a query in a query:

1 Create a new application page in ColdFusion Studio.

2 Edit the page so that it appears as follows:

```
<html>
<head>
<title>Using Query Results in a Query</title>
</head>
<body>
<h1>Employee List</h1>
<!--- LastNameSearch normally would be generated interactively --->
<cfset LastNameSearch = "Jones">
<!--- Normal query --->
<cfquery datasource = "CompanyInfo" name = "EmpList"
     cachedwithin=#CreateTimeSpan(0,1,0,0)#>
  SELECT *
  FROM Employee
</cfquery>

<!--- Query using query results --->
<cfquery dbtype = "query" name = "QueryFromQuery" >
  SELECT Emp_ID, FirstName, LastName
  FROM EmpList
  WHERE LastName = ’#LastNameSearch#’
</cfquery>

Output using a query of query<br>
<cfoutput query = QueryFromQuery>
  #Emp_ID#: #FirstName# #LastName#<br>
</cfoutput>
<br>

Columns in the Emplist database query<br>
<cfoutput>
  #Emplist.columnlist#<br>
</cfoutput>
<br>

Columns in the QueryFromQuery query<br>
<cfoutput>
  #QueryFromQuery.columnlist#<br>
</cfoutput>
</body>
</html>
```

3 Save the page as queryquery.cfm in myapps under the Web root directory.

4 Return to your browser and enter the following URL to view the results of the query:

http://127.0.0.1/myapps/queryquery.cfm

5 View the source in the browser.

## Reviewing the code

The page retrieves the entire Employee table from the CompanyInfo database. A second query selects only the three columns to display for employees with the specified last name. The following table describes the code and its function:

| Code | Description |
|---|---|
| `cfset ListNameSearch = "Jones"` | Set the last name to use in the second query. In a complete application, this information comes from user interaction. |
| `<cfquery datasource = "CompanyInfo"`<br>`name = "EmpList"`<br>`cachedwithin=#CreateTimeSpan(0,1,0,0`<br>`)#>`<br>`  SELECT *`<br>`  FROM Employee`<br>`</cfquery>` | Query the database specified in the CompanyInfo data source and select all data in the Employee table. Cache the query data between requests to this page, and do not query the database if the cached data is less than an hour old. |
| `<cfquery dbtype = "query" name =`<br>`"QueryFromQuery" >`<br>`  SELECT Emp_ID, FirstName,`<br>`    LastName`<br>`  FROM Emplist`<br>`  WHERE LastName='#LastNameSearch#'`<br>`</cfquery>` | Use the EmpList query as the source of the data in a new query. This query selects only entries that match the last name specified by the `LastNameSearch` variable. The query also selects only three columns of data: employee ID, first name, and last name. |
| `<cfoutput query = QueryFromQuery>`<br>`  #Emp_ID#: #FirstName#`<br>`  #LastName#<br>`<br>`</cfoutput>`<br>`<br>` | Use the QueryFromQuery query to display the list of employee IDs, first names, and last names. |
| `<cfoutput>`<br>`  #EmpList.columnlist#<br>`<br>`</cfoutput>` | List all the columns returned by the Emplist query. |
| `<cfoutput>`<br>`  #QueryFromQuery.columnlist#<br>`<br>`</cfoutput>` | List all the columns returned by the QueryFromQuery query. |

Chapter 4

# Retrieving and Formatting Data

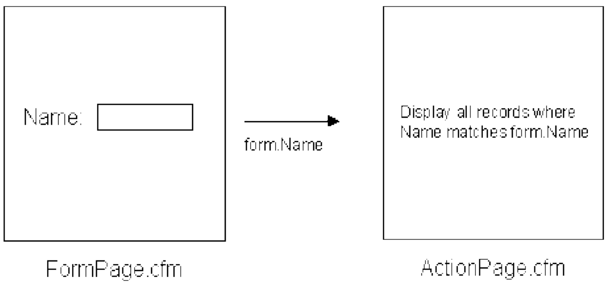
This chapter explains how to select the data to display in a dynamic Web page. It also describes how to populate an HTML table with query results and how to use ColdFusion functions to format and manipulate data.

## Contents

## Using Forms to Specify the Data to Retrieve

In the examples in previous chapters, you have retrieved all of the records from a table. However, there are many instances when you want to retrieve data based on certain criteria. For example, you might 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 standard HTML, the syntax and examples that follow provide you with just enough detail to begin using ColdFusion.

## form tag syntax

Use the following syntax for the create a form tag:

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

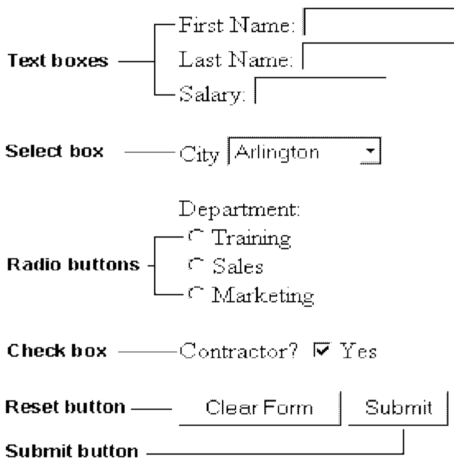
| Attribute | Description |
|---|---|
| `action` | Specifies an action page to which you pass form variables for processing. |
| `method` | Specifies how the variables are submitted from the browser to the action page on the server. All ColdFusion forms *must* be submitted with an attribute setting of method="post" |

You can override the server request timeout (set on the ColdFusion Administrator Server Settings page) by adding a RequestTimeout parameter to the action page URL. The following example specifies a request timeout of two minutes:

```
<form name="getReportCriteria"
    action="runReport.cfm?RequestTimeout=120" method="post">
```

## Form controls

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



The following table illustrates the format of form control tags:

| 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` |
| List 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` |

| Control | Code |
|---|---|
| Reset button | `<input type="Reset" name="ControlName" value="DisplayName">` |
| Submit button | `<input type="Submit" name="ControlName" value="DisplayName">` |

Use the following procedure to create a sample form.

### To create a form

1 Create a new application page, using ColdFusion 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>

<!-- list box -->
<p>
City
<select name="City">
  <option value="Arlington">Arlington
  <option value="Boston">Boston
  <option value="Cambridge">Cambridge
  <option value="Minneapolis">Minneapolis
  <option value="Seattle">Seattle
</select>
</p>

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

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

<!-- reset button -->
<input type="Reset" name="ResetForm" value="Clear Form">
<!-- submit button -->
<input type="Submit" name="SubmitForm" value="Submit">

</form>
</body>
</html>
```

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.

## Reviewing the code

The following table describes the highlighted code and its function:.

| 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. |

| Code | Description |
|---|---|
| `<select name="City">`<br>`    <option value="Arlington">`<br>`        Arlington`<br>`    <option value="Boston">Boston`<br>`    <option value="Cambridge">`<br>`        Cambridge`<br>`    <option value="Minneapolis">`<br>`        Minneapolis`<br>`    <option value="Seattle">Seattle`<br>`</select>` | Create a drop-down list box named City and populate it with the values “Arlington,” “Boston,” “Cambridge,” “Minneapolis,” and “Seattle.” |
| `<input type="checkbox" name=`<br>`    "Contractor" value="Yes\|No"`<br>`    checked>Yes` | Create a check box that allows users to specify whether they want to list employees who are contractors. Make the box selected by default. |
| `<input type="Reset"`<br>`name="ResetForm" value="Clear`<br>`Form">` | Create a reset button to allow users to clear the form. Put the text Clear Form on the button. |
| `<input type="Submit"`<br>`name="SubmitForm" value="Submit">` | Create a submit button to send the values that 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.
- For ease of use, limit radio buttons to between three and five mutually exclusive options. If you need more options, consider a drop-down select list.
- Use list boxes to allow the user to choose from many options or to chose multiple items from a list.
- All the data that you collect on a form is automatically passed as form variables to the associated action page.
- Check boxes and radio buttons do not pass to action pages unless they are selected on a form. If you try to reference these variables on the action page, you receive an error if they are not present.
- You can dynamically populate drop-down select lists using query data. See “Dynamically Populating List Boxes” on page 82 for details.

# Working with Action Pages

A ColdFusion action page is just like any other application page except that you can use the form variables that are passed to it from an associated form. The following sections describe how to create effective action pages.

## Processing form variables on action pages

The action page gets a form variable for every form control that contains a value when the form is submitted.

**Note**
If multiple controls have the same name, one form variable is passed to the action page. 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 the form variable by name within tags, functions, and other expressions on an action page.

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

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

## Dynamically generating SQL statements

As you have 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   Employee
</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 Smith, you build a query that looks like this:

```
<cfquery name="GetEmployees" datasource="CompanyInfo">
  SELECT FirstName, LastName, Contract
  FROM    Employee
  WHERE  LastName = "Smith"
</cfquery>
```

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

```
<cfquery name="GetEmployees" datasource="CompanyInfo">
  SELECT FirstName, LastName, Salary
  FROM Employee
  WHERE LastName="#Form.LastName#"
</cfquery>
```

For more information on Dynamic SQL, see “Dynamic SQL” on page 96.

## Creating action pages

Use the following procedure to create an action page for cormpage.cfm.

### To create an action page for the form:

1 Create a new application page in ColdFusion Studio.

2 Enter the following code:

```
<html>
<head>
<title>Retrieving Employee Data Based on Criteria from Form</title>
</head>

<body>
<cfquery name="GetEmployees" datasource="CompanyInfo">
  SELECT FirstName, LastName, Salary
  FROM Employee
  WHERE LastName='#Form.LastName#'
</cfquery>
<h4>Employee Data Based on Criteria from Form</h4>
<cfoutput query="GetEmployees">
#FirstName#
#LastName#
#Salary#<br>
</cfoutput>
<br>
<cfoutput>Contractor: #Form.Contractor#</cfoutput>
</body>
</html>
```

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

4 View `formpage.cfm` in your browser.

5 Enter data, for example, Smith, in the Last Name box and submit the form.

The browser displays a line with the first and last name and salary for each entry in the database that match the name you typed, followed by a line with the text “Contractor: Yes”

6 Click Return in your browser to redisplay the form.

7 Remove the check mark from the check box and submit the form again.

This time an error occurs because the check box does not pass a variable to the action page.

### Reviewing the code

The following table describes the highlighted code and its function:

| Code | Description |
|---|---|
| `<cfquery name="GetEmployees"`<br>`datasource="CompanyInfo">` | Query the data source CompanyInfo and name the query GetEmployees. |
| `SELECT FirstName, LastName, Salary`<br>`FROM Employee`<br>`WHERE LastName='#Form.LastName#'` | Retrieve the FirstName, LastName, and Salary fields from the Employee table, but only if the value of the LastName field matches what the user entered in the LastName text box in the form on formpage.cfm. |
| `<cfoutput query="GetEmployees">` | Display results of the GetEmployees query. |
| `#FirstName#`<br>`#LastName#`<br>`#Salary#<BR>` | 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. |
| `<br>`<br>`<cfoutput>Contractor:`<br>`#Form.Contractor#</cfoutput>` | Display a blank line followed by the text Contractor: and the value of the form Contractor check box. A more complete example would test to ensure the existence of the variable and would use the variable in the query. |

## Testing for a variable's existence

Before relying on a variable's existence in an application page, you can test to see if it exists using the `IsDefined` function. A **function** is a named procedure that takes input and operates on it. For example, the `IsDefined` function determines whether a variable is defined. CFML provides a large number of functions, which are documented in the *CFML Reference*.

The following code prevents the error that you saw in the previous example by checking to see if the Contractor Form variable exists before using it:

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

The argument passed to the IsDefined function must always be enclosed in double quotes. For more information on the IsDefined function, see the *CFML Reference*.

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

## 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.
- Check boxes and radio button variables only get passed to the action page if you select an option. Text boxes pass an empty string if you do not enter text.
- An error occurs if the action page tries to use a variable that has not been passed.

# Working with Queries and Data

The ability to generate and display query data is one of the most important and flexible features of ColdFusion. The following sections further your understanding of using queries and displaying their results. Some of these tools are effective for presenting any data, not just query results.

## Using HTML tables to display query results

You displayed each row of data from the Employee 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 actionpage.cfm in ColdFusion 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 Employee
  WHERE LastName='#Form.LastName#'
</cfquery>
<h4>Employee Data Based on Criteria from Form</h4>
<table>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Salary</th>
</tr>
<cfoutput query="GetEmployees">
<tr>
<td>#FirstName#</td>
<td>#LastName#</td>
<td>#Salary#</td>
</tr>
</cfoutput>
</table>
<br>
<cfoutput>Contractor: #Form.Contractor#</cfoutput>
```

```
</body>
</html>
```

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

4 View formpage.cfm in your browser.

5 Enter Smith in the Last Name text box and submit the form.

6 The records that match the criteria specified in the form appear in a table.

### Reviewing the code

The following table describes the highlighted code and its function:

| Code | Description |
|---|---|
| `<table>` | Put data into a table. |
| `<tr>`<br>`    <th>First Name</th>`<br>`    <th>Last Name</th>`<br>`    <th>Salary</th>`<br>`</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>`<br>`    <td>#FirstName#</td>`<br>`    <td>#LastName#</td>`<br>`    <td>#Salary#</td>`<br>`</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 might 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 Open the file actionpage.cfm in ColdFusion 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 technique is known as **pattern matching** and is often used to query databases.

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

```
<cfquery name="GetEmployees" datasource="CompanyInfo">
  SELECT FirstName, LastName,
  StartDate, Salary, Contract
  FROM    Employee
  WHERE   LastName LIKE 'AL%'
</cfquery>
```

- The LIKE operator tells the database to use the string that follows for pattern matching.
- If you place a wildcard before and after AL, you retrieve any record in that column that contains AL.
- Surround strings in SQL statements with single quotes (').

To return information from the Departmt table on all departments except the sales department, you would build a query that looks like this:

```
<cfquery name="GetDepartments" datasource="CompanyInfo">
  SELECT *
  FROM    Departmt
  WHERE   Dept_Name NOT LIKE '[Ss]ales'
</cfquery>
```

- The first character in the match can be either upper case S or lower case s.

**Note**
Whether SQL identifiers and data comparison operations are case sensitive depends on the database.

## Filtering data based on multiple conditions

When you want to retrieve data based on the results of more than one comparison you can use AND and OR operators to combine conditions.

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

```
<cfquery name="GetEmployees" datasource="CompanyInfo">
  SELECT FirstName, LastName,
  StartDate, Salary, Contract
  FROM    Employee
  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 are working with:

- Department information is maintained in the Departmt table. This includes department ID numbers.
- Employee information is maintained in the Employee 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 who work for the HR department, you build a query that looks like this:

```
<cfquery name="GetEmployees" datasource="CompanyInfo">
  SELECT Departmt.Dept_Name,
  Employee.FirstName,
  Employee.LastName,
  Employee.StartDate,
  Employee.Salary
  FROM Departmt, Employee
  WHERE Departmt.Dept_ID = Employee.Dept_ID
  AND Departmt.Dept_Name = ’HR’
</cfquery>
```

In this example, the following criteria joins the two tables:

```
Departmt.Dept_ID = Employee.Dept_ID
```

It ensures that each row of the query results contains the department name from the Departmt table that corresponds to the Department ID in this employee’s row in the Employee table. Without this statement, the query returns a row for *every* employee in the Employee table, and all rows have the Dept_Name HR, even if the employee is not in the HR department.

When you do table joins, keep the following information in mind:

- Prefix each column in the SELECT statement to explicitly state which table the data should be retrieved from.
- In this example, the Dept_ID field is the primary key of the Departmt table and the foreign Key of the Employee table. A **foreign key** uniquely identifies another record (in this example, a record in the Departmt table) but does not uniquely identify the current record (in the Employee table).

## Building flexible search interfaces

If you want users to optionally enter multiple search criteria, you can 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.

The following action page allows users to search for employees by department, last name, or both.

### To build a more flexible search interface:

1 Open the page actionpage.cfm in ColdFusion 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 Departmt.Dept_Name,
  Employee.FirstName,
  Employee.LastName,
  Employee.StartDate,
  Employee.Salary
  FROM Departmt, Employee
  WHERE Departmt.Dept_ID = Employee.Dept_ID
  <cfif IsDefined("FORM.Department")>
         AND Departmt.Dept_Name = '#Form.Department#'
  </cfif>
  <cfif Form.LastName IS NOT "">
    AND Employee.LastName = '#Form.LastName#'
  </cfif>
</cfquery>

<h4>Employee Data Based on Criteria from Form</h4>
<table>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Salary</th>
</tr>
<cfoutput query="GetEmployees">
<tr>
<td>#FirstName#</td>
<td>#LastName#</td>
<td>#Salary#</td>
</tr>
</cfoutput>
</table>
</body>
</html>
```

3 Save the file.

4 View formpage.cfm in your browser.

5 Select a department, optionally enter a last name, and submit the form.

## Reviewing the code

The following table describes the highlighted code and its function:

| Code | Description |
|---|---|
| `SELECT  Departmt.Dept_Name, Employee.FirstName, Employee.LastName, Employee.StartDate, Employee.Salary FROM Departmt, Employee WHERE Departmt.Dept_ID = Employee.Dept_ID` | Retrieve the fields listed from the Departmt and Employee tables, joining the tables based on the Dept_ID field in each table. |
| `<cfif IsDefined("FORM.Department")> AND Departmt.Dept_Name = ’#Form.Department#’ </cfif>` | If the user specified a department on the form, only retrieve records where the department name is the same as the one the user specified. Note that you need the pound signs in the SQL AND statement to identify Form.Department as a ColdFusion variable, but not in the IsDefined function. |
| `<cfif Form.LastName IS NOT ""> AND Employee.LastName = ’#Form.LastName#’ </cfif>` | If the user specified a last name in the form, only retrieve the records in which the last name is the same as the one the user entered in the form. |

## Returning Results to the User

When you return your results to the user, you must make sure that your pages respond to the user's needs and are appropriate for the type and amount of information. In particular you must consider the following situations:

- When there are no query results
- When you want to return partial results

### If there are no query results

Your code must accommodate the cases where a query does not return any records. To determine whether a search has retrieved records, use the `RecordCount` query variable. You can use the variable in a conditional logic expression that determines how to display search results appropriately to users.

For example, to inform the user when no records were found by the GetEmployees query, insert the following code before displaying the data:

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

You mst do the following:

- Prefix `RecordCount` with the query name.
- Add a procedure after the `cfif` tag that displays a message to the user.
- Add a procedure after the `cfelse` tag to format the returned data.
- Follow the second procedure with a `</cfif>` tag end to indicate the end of the conditional code.

#### To return search results to users:

1 Open the page actionpage.cfm in ColdFusion Studio.

2 Change 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 Departmt.Dept_Name,
  Employee.FirstName,
  Employee.LastName,
  Employee.StartDate,
  Employee.Salary
  FROM Departmt, Employee
  WHERE Departmt.Dept_ID = Employee.Dept_ID
  <cfif isdefined("FORM.Department")>
    AND Departmt.Dept_Name = ’#Form.Department#’
  </cfif>
  <cfif form.lastname is not "">
    AND Employee.LastName = ’#Form.LastName#’
  </cfif>
</cfquery>

<cfif GetEmployees.recordcount is "0">
No records match your search criteria. <br>
Please go back to the form and try again.
<cfelse>
<h4>Employee Data Based on Criteria from Form</h4>
<table>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Salary</th>
</tr>
<cfoutput query="GetEmployees">
<tr>
<td>#FirstName#</td>
<td>#LastName#</td>
<td>#Salary#</td>
</tr>
</cfoutput>
</cfif>
</table>
</body>
</html>
```

3 Save the file.

4 Return to the form, enter search criteria and submit the form.

5 If no records match the criteria you specified, the message displays.

## Returning results incrementally

You can use the `cfflush` tag to incrementally output long-running requests to the browser before a ColdFusion page is fully processed. This allows you to give the user quick feedback when it takes a long time to complete processing a request. For example, you can use `cfflush` to display the message, “Processing your request -- please wait.” when a request takes time to return. You can also use it to incrementally display a long list as it gets retrieved.

The first time you use the `cfflush` tag on a page, it sends to the browser all of the HTML headers and any other available HTML. Subsequent `cfflush` tags on the page send only the output that ColdFusion generated since the previous flush.

You can specify an `interval` attribute to tell ColdFusion to flush the output each time that at least the specified number of bytes become available. (The count does not include HTML headers and any data that is already available when you make this call.) You can use the `cfflush` tag in a `cfloop` to incrementally flush data as it becomes available. This format is particularly useful when a query responds slowly with large amounts of data.

When you flush data, make sure that a sufficient amount of information is available, because some browsers might not respond if you flush only a very small amount. Similarly, if you use an `interval` attribute, set it for a reasonable size, such as a few hundred bytes or more but not many thousands of bytes.

**Caution**

After you use the `cfflush` tag on a page, any CFML function or tag on the page that modifies the HTML header causes an error. These include the `cfcontent`, `cfcookie`, `cfform`, `cfheader`, `cfhtmlhead`, and `cflocation` tags. You also get an error if you use the `cfset` tag to set a Cookie scope variable. All errors except Cookie errors can be caught with a `cfcatch type="template"` tag. Cookie errors can be caught with `cfcatch type="Any"`.

The following example uses the `cfloop` tag and the `rand()` random number generating function to artificially delay the generation of data for display. It simulates a situation in which it takes time to retrieve the first data and additional information becomes available slowly.

```
<html>
<head>
  <title>Your Magic numbers</title>
</head>

<body>
<H1>Your Magic numbers</H1>
<P>It will take us a little while to calculate your ten magic numbers.
It takes a lot of work to find numbers that truly fit your personality.
So relax for a minute or so while we do the hard work for you.</P>
<H2>We are sure you will agree it was worth the short wait!</H2>
<cfflush>

<cfflush interval=10>
<!--- Delay Loop to make is seem harder --->
<cfloop index="randomindex" from="1" to="200000" step="1">
  <cfset random=rand()>
</cfloop>

<!--- Now slowly output 10 random numbers --->
<cfloop index="Myindex" from="1" to="10" step="1">
  <cfloop index="randomindex" from="1" to="100000" step="1">
    <cfset random=rand()>
  </cfloop>
  <cfoutput>
    Magic number number #Myindex# is:&nbsp;&nbsp;#RandRange( 100000,
      999999)#<br><br>
  </cfoutput>
</cfloop>
</body>
</html>
```

## Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<H2>We are sure you will agree it was worth the short wait!</H2>` `<cfflush>` | Send the HTML header and all HTML output to the `cfflush` tag to the user. This displays the explanatory paragraph and H2 tag contents. |
| `<cfflush interval=10>` | Flush additional data to the user every time at least ten bytes are available. |
| `<cfloop index="randomindex" from="1" to="200000" step="1">` `<cfset random=Rand()>` `</cfloop>` | Insert an artificial delay by using the `Rand` function to calculate many random numbers. |
| `<cfloop index="Myindex" from="1" to="10" step="1">` `<cfloop index="randomindex" from="1" to="100000" step="1">` `<cfset random=rand()>` `</cfloop>` `<cfoutput>` `Magic number number #Myindex# is:&nbsp;&nbsp;#RandRange (100000,999999)#<br><br>` `</cfoutput>` `</cfloop>` | Generate and display ten random numbers. This code uses two loops. The outer loop is repeated ten times, once for each number to display. The inner loop uses the `rand` function to create another delay by generating more (unused) random numbers. It then calls the `RandRange` function to generate a six-digit random number for display. |

Chapter 5

# Graphing Data

This chapter explains how to use the `cfgraph` tag to display graphs. It illustrates ways that you can graph data and gives you the tools you need to create effective graphs.

The `cfgraph` tag requires the JRun server and the Macromedia Generator server, which install with ColdFusion.

## Contents

# Creating a Graph

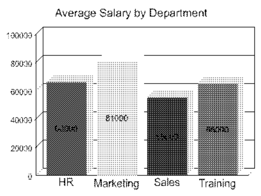
The `cfgraph` tag provides five graph types. A large number of attributes let you customize the graph appearance to meet your needs.

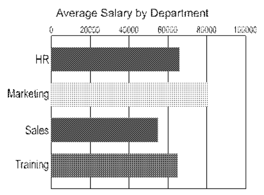
## Graph types

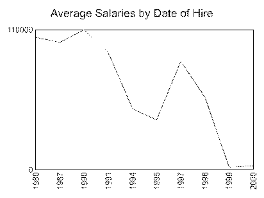
You can create the following types of graphs:

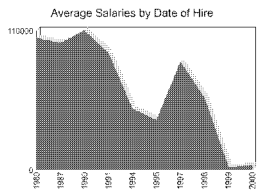
- Bar
- Horizontal bar
- Line
- Area (a line graph with the area below the line filled in)
- Pie

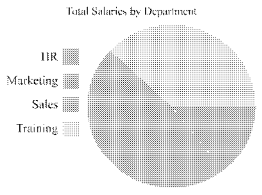
The following illustrations show one sample of each type of graph:











## Creating a basic graph

You use the following `cfgraph` attributes to create a basic graph:

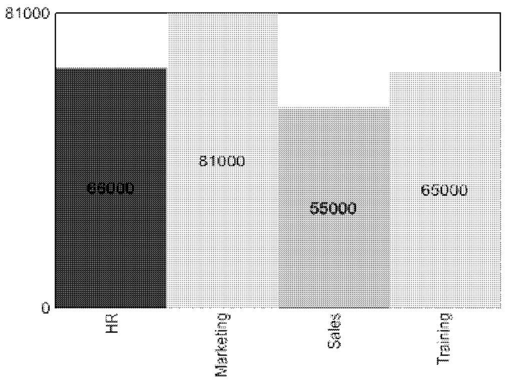
| Attribute | Description |
|---|---|
| type | Must be one of the following values: `bar`, `horizontalbar`, `pie`, or `line`. (An area graph is a kind of line graph.) |
| query | The query containing the data. |
| valueColumn | The query column containing the values to be graphed. |
| itemColumn | (Optional) The query column containing the description for this data point. The item normally appears on the horizontal axis of bar and line graphs, and in pie charts. |

You must end your `cfgraph` tag with a `</cfgraph>` end tag.

For example, if you have a query that contains average salary by department, the following tag displays a bar graph with the information:

```
<cfgraph type="bar"
        query="DataTable"
        valueColumn="AvgByDept"
        itemColumn="Dept_Name">
</cfgraph>
```

The resulting graph looks like this:



Later sections in this chapter provide information on how you can specify the data differently and how you can change and enhance graph appearance.

## Graphing Data

One of the most important considerations when you graph data is the way you supply the data to the `cfgraph` tag. You can supply data in several ways:

- Provide all the data in a single query.
- Specify individual data points using `cfgraphdata` tags.
- Combine data from a query with additional data points from `cfgraphdata` tags.

**Note**
The `cfgraph` tag graphs numeric data only. As a result, you must convert any dates, times, or preformatted currency values, such as $3,000.53, to integers or real numbers.

## Graphing a query

When you graph a query, you specify the query name in the `cfgraph` tag `query` attribute. In this format the `cfgraph` tag body is empty. However, you must still provide the `<\cfgraph>` end tag. For example, a simple `cfgraph` tag for a bar chart might look like this:

```
<cfgraph type="bar" title="Salaries by Department"
        query="DataTable"
        itemColumn="Dept_Name"
        valueColumn="AvgByDept">
</cfgraph>
```

This tag displays the values in the AvgByDept column of the DataTable query. It displays the Dept_Name column value as the item label by each bar. The title “Salaries by Department” appears above the chart.

The `cfgraph` tag can take the following information from a query:

| Attribute | Description |
|---|---|
| query | The query containing the data. |
| valueColumn | The query column containing the values to be graphed. |
| itemColumn | (Optional) The query column containing the description for this data point. The item normally appears on the horizontal axis of bar and line graphs, on the vertical axis of horizontal bar graphs, and in the legend in pie charts. |

| Attribute | Description |
|---|---|
| URL | (Optional) Works only with bar, horizontal bar, and pie charts in Flash file format.<br>A static prefix for all data point URLs. When the user clicks a bar or pie wedge, the page links to a URL created by appending the data from the data point's `URLColumn` value.<br>Use this attribute to specify a string that is part of all links on a chart, such as http://www.mycompany.com/myapp/salary_info/chart_details/. |
| URLColumn | (Optional) Works only with bar, horizontal bar, and pie charts in Flash file format.<br>The query column containing the data point–specific part of a URL to load when the user clicks the corresponding data point the graph. ColdFusion encodes the contents of the query field, in URL format (for example, replacing space characters with %20) and appends it to any static URL string in the `URL` attribute to create a full URL link.<br>Use this attribute to do data drill-down from your charts.<br>For more information on using URLs in graphs, see "Linking Dynamically from Graphs," on page 77. |

The ability to use queries of queries, as described in Chapter 3, "Using Query Results in Queries" on page 34 provides significant power in generating the data for the chart. For example, you can use aggregating clauses operators such as SUM, AVG, and GROUP BY to create a query of queries with statistical data based on a raw database query.

You can also take advantage of the ability to reference and modify query data dynamically. For example, you can loop through the entries in a query column and reformat the data to show whole dollar values

## Example: graphing a query of queries

The example in the following procedure analyzes the salary data in the CompanyInfo database and displays three graphs:

### To graph a query of queries:

1 Create a new application page in ColdFusion Studio.

2 Edit the page so that it appears as follows:

```
<!-- Get the raw data from the database. -->
<cfquery name="GetSalaries" datasource="CompanyInfo">
  SELECT Departmt.Dept_Name,
    Employee.Salary
  FROM Departmt, Employee
  WHERE Departmt.Dept_ID = Employee.Dept_ID
</cfquery>
```

```
<!-- Generate a query with statistical data for each department. -->
<cfquery dbtype = "query" name = "DeptSalaries">
  SELECT
    Dept_Name,
    AVG(Salary) AS AvgByDept
  FROM GetSalaries
  GROUP BY Dept_Name
</cfquery>

<!--- Reformat the generated numbers to show only thousands --->
<cfloop index="i" from="1" to="#DeptSalaries.RecordCount#">
  <cfset DeptSalaries.AvgByDept[i]=Round(DeptSalaries.AvgByDept[i]/
     1000)*1000>
</cfloop>

<html>
<head>
  <title>Employee Salary Analysis</title>
</head>

<body>
<h1>Employee Salary Analysis</h1>

<!--- Bar graph, from DeptSalaries Query of Queries --->
<cfgraph type="bar"
  query="DeptSalaries"
  valueColumn="AvgByDept"
  itemColumn="Dept_Name">
</cfgraph>
<br>

</body>
</html>
```

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

C:\Inetpub\wwwroot\myapps on Windows NT

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

http://127.0.0.1/myapps/graphdata.cfm

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfquery name="GetSalaries" datasource="CompanyInfo">`<br>`  SELECT Departmt.Dept_Name,`<br>`    Employee.Salary`<br>`  FROM Departmt, Employee`<br>`  WHERE Departmt.Dept_ID = Employee.Dept_ID`<br>`</cfquery>` | Query the CompanyInfo database to get the Dept_Name and Salary for each employee. Because the Dept_Name is in the Departmt table and the Salary is in the Employee table, you need a table join in the WHERE clause. The raw results of this query could be used elsewhere on the page. |
| `<cfquery dbtype = "query" name = "DeptSalaries">`<br>`  SELECT`<br>`    Dept_Name,`<br>`    AVG(Salary) AS AvgByDept`<br>`  FROM GetSalaries`<br>`  GROUP BY Dept_Name`<br>`</cfquery>` | Generate a new query from the GetSalaries query. Use the AVG aggregating function to get statistical data on the employees. Use the GROUP BY statement to ensure that there is only one row for each department. |
| `<cfloop index="i" from="1"`<br>`    to="#DeptSalaries.RecordCount#">`<br>`  <cfset DeptSalaries.AvgByDept[i]=`<br>`    Round(DeptSalaries.AvgByDept[i]`<br>`    /1000)*1000>`<br>`</cfloop>` | Loop through all the rows in DeptSalaries query and round the salary data to the nearest thousand. This loop uses the query variable RecordCount to get the number of rows and changes the contents of the query object directly. |
| `<cfgraph type="bar"`<br>`query="DeptSalaries"`<br>`valueColumn="AvgByDept"`<br>`itemColumn="Dept_Name"`<br>`>`<br>`</cfgraph>` | Create a bar graph using the data from the AvgByDept column of the DeptSalaries query. Label the bars with the Department names. |

## Graphing individual data points

When you graph individual data points you specify each data point by putting a cfgraphdata tag in the cfgraph tag body. For example, the following code creates a simple pie chart:

```
<cfgraph type="pie" title="Income by Department">
  <cfgraphdata item="New Vehicle Sales" value=500000>
  <cfgraphdata item="Used Vehicle Sales" value=250000>
  <cfgraphdata item="Leasing" value=300000>
  <cfgraphdata item="Service" value=400000>
</cfgraph>
```

This pie chart displays the income values of four departments. Each cfgraph tag specifies a department's income and the corresponding item description for the legend. The values are specified by individual ColdFusion variables. The title "Income by Department" appears above the chart.

The cfgraphdata tag lets you specify the following information about a data point:

| Attribute | Description |
|---|---|
| value | The data value to be graphed. |
| item | (Optional) The description for this data point. The item normally appears on the horizontal axis of bar and line graphs, on the vertical axis of horizontal bar graphs, and in the legend in pie charts. |
| color | (Optional) The color of the bar or pie slice. Ignored for line and area graphs. |
| URL | (Optional) Works only with bar, horizontal bar, and pie charts in Flash file format.<br>A URL to load when the user clicks this data point. Use this attribute to do data drill-down from your charts.<br>For more information on using URLs in graphs, see "Linking Dynamically from Graphs," on page 77. |

## Combining a query and data points

To graph data from both query and individual data value, you specify the query name and related attributes in the cfgraph tag and provide the additional data points and their appearance attributes in cfgraphdata tags.

Data specified by a cfgraphdata tag is graphed before (for example, to the left on a bar chart) the data from a query.

For example, if the database is missing data for one department, you can add the information manually:

```
<cfgraph type="bar"  title="Salaries by Department" query="DataTable"
        itemColumn ="Dept_Name" valueColumn="AvgByDept"
  <cfgraphdata item="Facilities" value="35000">
</cfgraph>
```

# Controlling Graph Appearance

The `cfgraph` tag allows you to customize the appearance of your graph in many ways.

## Common graph characteristics

You can optionally specify the following characteristics on all types of graphs:

| **Graph characteristic** | **Attributes used** | **Description** |
|---|---|---|
| Title | title<br>titleFont | The title to display on the graph and the font to use. |
| File Type | fileFormat | Whether to send the graph to the user as a jpeg or Flash (.swf) file. Flash is the default format. On pie and bar charts, Flash supports rollover display of data values and data drill-down by clicking on the data point (using the `URLColumn` attribute). |
| Dimensions | graphWidth<br>graphHeight | The width and height in pixels of the graph. This size defines the entire graph area, including the legend and background area around the graph.<br>You cannot use these attributes to change the ratio of the data area height to width. For example, you cannot set a large `graphWidth` value to stretch just the horizontal dimension. To change the overall graph size, specify both the `graphHeight` and `graphWidth`. |
| Background | backgroundColor | The background color to use for the entire graph area, including legends and margins. You can specify any of the standard 256 Web colors. You can use any valid HTML color format. If you use the numeric format, you must use double pound signs, for example, ##CCFFFF. |
| Border | borderWidth<br>borderColor | The border that surrounds the graph. You specify the width in pixels and the color using any valid HTML color format, as described for the `backgroundColor`. A value of 0 means no border. |
| 3D Appearance | depth | The depth of the shading that gives the graph three-dimensional appearance, in pixels. A value of 0 (the default) means no 3D appearance. |

## Setting bar and horizontal bar chart characteristics

You can specify the following additional characteristics for bar and horizontal bar charts:

| Graph characteristic | Attributes used | Description |
|---|---|---|
| Value labels | showValueLabel<br>valueLabelFont<br>valueLabelSize<br>valueLocation | Labels that display the numeric value being graphed.<br>By default, value labels are on. You can turn them off or have them display when the user points to the bar (Flash file format only). You can specify the font type (Arial, Courier, or Times), point size, and location (OnBar or OverBar). |
| Value axis | scaleFrom<br>scaleTo | The minimum and maximum points on the data axis (vertical axis for bar charts, horizontal axis for horizontal bar charts.<br>By default the minimum is 0 and the maximum is the largest data value. |
| Grid lines | gridLines | The number of grid lines between the top and bottom of the graph.<br>The value of each grid line appears along the value axis. The `cfgraph` tag displays horizontal grids only. A value of 0 (the default) means no grid lines. |
| Item labels | showItemLabel<br>itemLabelFont<br>itemLabelSize<br>itemLabelOrientation | Labels to show on the second axis of the chart.<br>Item labels are on by default if you specify an `itemColumn` (or for `cfgraphdata` tags, `item`) attribute. You can specify the label font type (Arial, Courier, or Times), point size, and orientation (horizontal or vertical). |

| Graph characteristic | Attributes used | Description |
|---|---|---|
| Data point colors | colorList | A comma-separated list of colors to use for each bar.<br>You can use any of the 256 standard Web colors and any valid Web color name notation (for example, blue or ##FF33CC). You must use double pound signs with hexadecimal color notation. These colors replace the standard system-defined colors. If you specify fewer colors than data points, the colors repeat. If you specify more colors than data points, the extra colors are not used. |
| Bar spacing | barSpacing | The space, in pixels, between bars.<br>Any 3D shadow specified by the `depth` attribute appears in this space, so if you want the background to appear between all bars, make the `barSpacing` value greater than the `depth` value. |

## Example: adding character to a bar graph

The example in the following procedure adds a title to the bar graph and changes its appearance from the default, flat look, to a 3D look. It adds gridlines, sets the maximum Y-axis value to 100000, separates the bars, and uses a custom set of colors.

### To enhance the bar graph:

1 Open graphdata.cfm in ColdFusion Studio.

2 Edit the `cfgraph` tag so that it appears as follows:

```
<!--- Bar graph, from Query of Queries --->
<cfgraph type="bar"
  query="DeptSalaries"
  valueColumn="AvgByDept"
  itemColumn="Dept_Name"
  title = "Average Salary by Department"
  depth = 10
  scaleTo = 100000
  itemLabelSize=16
  itemLabelOrientation="horizontal"
  colorList = "red,orange,green,teal,purple"
  gridLines = 4
  barSpacing = 15>
</cfgraph>
```

3 Save the page.

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

   http://127.0.0.1/myapps/graphdata.cfm

## Reviewing the code

The following table describes the highlighted code and its function:

| Code | Description |
|---|---|
| `title = "Average Salary by Department"` | Put a title above the graph. |
| `depth = 10` | Give the graph 10 pixels of 3D "depth" shadow. |
| `scaleTo = 100000` | Set the maximum value of the vertical axis to 100000. The minimum value is the default, 0. |
| `itemLabelSize=16` | Make the labels on the horizontal axis 16 points. |
| `itemLabelOrientation="horizontal"` | Make the labels horizontal on the horizontal axis. |
| `colorList = "red,orange, green,teal,purple"` | Get the bar colors from a custom list. In this example, the graph does not use purple because there are only four data points. |
| `gridLines = 4` | Display four grid lines between the top and bottom of the graph. |
| `barSpacing = 15` | Separate the bars by 15 pixels of background. |

## Setting pie chart characteristics

You can specify the following additional characteristics for pie charts:

| Graph characteristic | Attributes used | Description |
|---|---|---|
| Value labels | showValueLabel<br>valueLabelFont<br>valueLabelSize<br>valueLocation | Labels that display the numeric value being graphed.<br>Value labels are on by default. You can turn them off or have them display when the user points to the bar (Flash file format only). You can specify the font type (Arial, Courier, or Times), point size, and location (OnBar or OverBar). |
| Legend | showLegend<br>legendFont | A legend relating the pie slice colors to the data point Item descriptions from the `itemColumn` attribute or `cfgraphdata` tag `itemColumn` attribute.<br>By default the legend appears to the left of the chart. You can also specify above, below, right, and none. You can specify the font type as Arial (the default), Courier, or Times. |
| Data point colors | colorList | A comma separated list of colors to use for each bar.<br>You can use any of the 256 standard Web colors and any valid Web color name notation (for example, blue or ##FF33CC). You must use double pound signs with hexadecimal color notation. These colors replace the standard system-defined colors.<br>If you specify fewer colors than data points, the colors repeat. If you specify more colors than data points, the extra colors are not used. |

## Example: adding a pie chart

The example in the following procedure adds a pie chart to the page.

### To create a pie chart:

1 Open graphdata.cfm in ColdFusion Studio.

2 Edit the DeptSalaries query and the `cfloop` code so that they appear as follows:

```
<!--- A query to get statistical data for each department. --->
<cfquery dbtype = "query" name = "DeptSalaries">
  SELECT
    Dept_Name,
    SUM(Salary) AS SumByDept,
    AVG(Salary) AS AvgByDept
  FROM GetSalaries
  GROUP BY Dept_Name
</cfquery>

<!--- Reformat the generated numbers to show only thousands --->
<cfloop index="i" from="1" to="#DeptSalaries.RecordCount#">
  <cfset DeptSalaries.SumByDept[i]=Round(DeptSalaries.SumByDept[i]/
     1000)*1000>
  <cfset DeptSalaries.AvgByDept[i]=Round(DeptSalaries.AvgByDept[i]/
     1000)*1000>
</cfloop>
```

3 Add the following `cfgraph` tag before the end of the body:

```
<!--- Pie graph, from DeptSalaries Query of Queries --->
<cfgraph type="pie"
  query="DeptSalaries"
  valueColumn="SumByDept"
  itemColumn="Dept_Name"
  title="Total Salaries by Department"
  titleFont="Times"
  showValueLabel="rollover"
  valueLabelFont="Times"
  borderWidth = 0
  backgroundColor = "##CCFFFF"
  colorlist="##6666FF,##66FF66,##FF6666,##66CCCC"
  LegendFont="Times">
</cfgraph>
<br>
```

4 Save the page.

5 Return to your browser and enter the following URL to view graphdata.cfm:

http://127.0.0.1/myapps/graphdata.cfm

## Reviewing the code

The following table describes the highlighted code and its function:

| Code | Description |
|---|---|
| `SUM(Salary) AS SumByDept,` | In the DeptSalaries query, add a SUM aggregation function to get the sum of all salaries per department. |
| `<cfset DeptSalaries.SumByDept[i]= Round(DeptSalaries.SumByDept[i]/ 1000)*1000>` | In the `cfloop`, round the salary sums to the nearest thousand. |
| `<cfgraph type="pie"` `query="DeptSalaries"` `valueColumn="SumByDept"` | Create a pie graph using the SumByDept salary sum values from the DeptSalares query. |
| `itemColumn="Dept_Name"` | Use the contents of the Dept_Name column for the item labels displayed in the chart legend. |
| `title="Total Salaries by Department"` `titleFont="Times"` | Put a title above the graph. Format it in Times font. |
| `showvalue="rollover"` `valueLabelFont="Times"` | Display the data value, in Times font, only when the user points to a pie slice. |
| `borderWidth = 0` | Do not put a border around the chart |
| `backgroundColor = "##CCFFFF"` `colorList = "##6666FF,##66FF66, ##FF6666,##66AAAA"` | Set the background for the entire chart area to a light blue. Get the pie slice colors from a custom list, which uses hexadecimal color numbers. The double pound signs prevent ColdFusion from trying to interpret the color data as variable names. |
| `LegendFont="Times"` | Use Times font for the legend. |

## Setting line and area graph characteristics

You can specify the following additional characteristics for line-based graphs

| Graph characteristic | Attributes used | Description |
|---|---|---|
| Value axis | scaleFrom<br>scaleTo | The minimum and maximum points on the vertical axis.<br>By default the minimum is 0 and the maximum is the largest data value. |
| Item labels | showItemLabel<br>itemLabelFont<br>itemLabelSize<br>itemLabelOrientation | Labels to show on the horizontal axis of the chart.<br>By default, item labels are on if you specify an `itemColumn` (or for `cfgraphdata` tags, `item`) attribute. You can specify the label font type (Arial, Courier, or Times), point size, and orientation (horizontal or vertical). |
| Line characteristics | lineColor<br>lineWidth | These attributes specify the line format.<br>For the line color, you can use any of the 256 standard Web colors and any valid Web color name notation (for example, blue or ##FF33CC). You must use double pound signs with hexadecimal color notation. The default line color is blue.<br>You can also specify the line width in pixels. The default is 1 pixel. |
| Area fill | fill | Specifies whether to fill the area below the line with the line color to form an area graph By default there is no fill. |
| Grid lines | gridLines | The number of grid lines between the top and bottom of the graph. The value of each grid line appears along the value axis. The `cfgraph` tag displays horizontal grids only. A value of 0 (the default) means no grid lines. |

## Example: adding an area graph

The example in the following procedure adds an area graph showing the average salary by start date to the salaries analysis page. It shows the use of a second query of queries to generate a new analysis of the raw data from the GetSalaries query; in this example, the average salary by start date. It also shows the use of additional `cfgraph` attributes.

### To create an area graph:

1 Open graphdata.cfm in ColdFusion Studio.

2 Edit the GetSalaries query so that it appears as follows:

```
<!-- Get the raw data from the database. -->
<cfquery name="GetSalaries" datasource="CompanyInfo">
  SELECT Departmt.Dept_Name,
    Employee.StartDate,
    Employee.Salary
  FROM Departmt, Employee
  WHERE Departmt.Dept_ID = Employee.Dept_ID
</cfquery>
```

3 Add the following code before the `html` tag:

```
<!--- Convert start date to start year. --->
<!--- You must explicitly convert the date to a number for the query
     to work --->
<cfloop index="i" from="1" to="#GetSalaries.RecordCount#">
<cfset GetSalaries.StartDate[i]=NumberFormat(DatePart("yyyy",
     GetSalaries.StartDate[i]) ,9999)>
</cfloop>

<!--- Query of Queries for average salary by start year --->
<cfquery dbtype = "query" name = "HireSalaries">
  SELECT
    StartDate,
    AVG(Salary) AS AvgByStart
  FROM GetSalaries
  GROUP BY StartDate
</cfquery>

<!--- Round average salaries to thousands --->
<cfloop index="i" from="1" to="#HireSalaries.RecordCount#">
  <cfset
     HireSalaries.AvgByStart[i]=Round(HireSalaries.AvgByStart[i]/
     1000)*1000>
</cfloop>
```

4 Add the following `cfgraph` tag before the end of the body tag block.

```
<!--- Area-style Line graph, from HireSalaries Query of Queries --->
<cfgraph type="line"
  query="HireSalaries"
  valueColumn="AvgByStart"
  itemColumn="StartDate"
  title="Average Salaries by Date of Hire"
  fileFormat="Flash"
  GraphWidth=400
  BackgroundColor="##FFFF00"
  Depth=5
  lineColor="teal"
  fill="yes" >
</cfgraph>
<br>
```

5 Save the page.

6 Return to your browser and enter the following URL to view graphdata.cfm:

http://127.0.0.1/myapps/graphdata.cfm

## Reviewing the code

The following table describes the highlighted code and its function:

| Code | Description |
|---|---|
| `Employee.StartDate,` | Add the employee start date to the data in the GetSalaries query. |
| `<cfloop index="i" from="1"`<br>`    to="#GetSalaries.RecordCount#">`<br>`  <cfset GetSalaries.StartDate[i]=`<br>`  NumberFormat(DatePart("yyyy",`<br>`  GetSalaries.StartDate[i]) ,9999)>`<br>`</cfloop>` | Use a `cfloop` to extract the year of hire from each employee's hire data and convert the result to a four-digit number. |
| `<cfquery dbtype = "query" name =`<br>`"HireSalaries">`<br>`  SELECT`<br>`    StartDate,`<br>`    AVG(Salary) AS AvgByStart`<br>`  FROM GetSalaries`<br>`  GROUP BY StartDate`<br>`</cfquery>` | Create a second query from the GetSalaries query. This query contains the average salary for each start year. |
| `<cfloop index="i" from="1"`<br>`   to="#HireSalaries.RecordCount#">`<br>`  <cfset HireSalaries.AvgByStart[i]`<br>`  =Round(HireSalaries.AvgByStart[i]`<br>`  /1000)*1000>`<br>`</cfloop>` | Round the salaries to the nearest thousand. |
| `<cfgraph type="line"`<br>`query="HireSalaries"`<br>`valueColumn="AvgByStart"`<br>`itemColumn="StartDate"` | Create a line graph using the HireSalaries query. Graph the average salaries against the start date. |
| `title="Average Salaries by`<br>`Date of Hire"` | Title the graph. |
| `fileFormat="Flash"` | Send the graph to the user as a Flash file. |
| `GraphWidth=400` | Limit the graph width to 400 pixels. Generator automatically resizes the graph's height to maintain the aspect ratio. |
| `BackgroundColor="##FFFF00"`<br>`Depth=5`<br>`lineColor="teal"` | Display a 3D graph in teal against a yellow background. |
| `fill="yes"` | Fill the region below the graph to create an area graph. |

# Linking Dynamically from Graphs

You can make Flash-format bar and pie charts interactive so that ColdFusion displays a new data point–specific Web page when the user clicks a bar or pie wedge. ColdFusion provides two methods for specifying the destination page:

- **For data points from queries,** ColdFusion takes the value of the `cfgraph` `URL` attribute, appends the value of the query column specified by the `URLColumn` attribute, and sends the resulting Web request.
- **For data points from `cfgraphdata` tags**, ColdFusion uses the value of the tag's `URL` attribute as the page to link to.

Using ColdFusion you can combine a static URL component with a query column component. This lets you link dynamically based on query column data without having to format the column contents as a URL. For example, you can use the values of the Dept_Name field in the CompanyInfo database to determine the data to display. To do this, follow these guidelines:

- In the `cfgraph` tag, specify a single Web page in the `URL` attribute.
- In the `URL` attribute, include the name of a parameter, but not its value, in the form *ParameterName*=
- In the `URLColumn` attribute, specify a query column that contains the value of the parameter being passed.
- In the target page, determine the data to be displayed based on the parameter that gets passed.

The example code in the following procedure illustrates this technique.

## Example: dynamically linking from a pie chart

In the following example, when you click a pie wedge, ColdFusion displays a table containing the detailed salary information for the departments represented by the wedge. The example is divided into two parts: creating the detail page and making the graph dynamic.

### Part 1: Creating the detail page

1 Create a new application page in ColdFusion Studio.

This page displays the drill-down information on the selected department based on the department name passed as the URL parameter.

2 Edit the page so that it appears as follows:

```
<cfquery name="GetSalaryDetails" datasource="CompanyInfo">
  SELECT Departmt.Dept_Name,
    Employee.FirstName,
    Employee.LastName,
    Employee.StartDate,
    Employee.Salary,
    Employee.Contract
  FROM Departmt, Employee
  WHERE Departmt.Dept_Name = '#URL.Dept_Name#'
  AND Departmt.Dept_ID = Employee.Dept_ID
  ORDER BY Employee.LastName, Employee.Firstname
</cfquery>

<html>
<head>
  <title>Employee Salary Details</title>
</head>

<body>

<h1><cfoutput>#GetSalaryDetails.Dept_Name[1]# Department
     Salary Details</cfoutput></h1>
<table border cellspacing=0 cellpadding=5>
<tr>
  <th>Employee Name</td>
  <th>StartDate</td>
  <th>Salary</td>
  <th>Contract?</td>
</tr>
<cfoutput query="GetSalaryDetails" >
<tr>
  <td>#FirstName# #LastName#</td>
  <td>#dateFormat(StartDate, "mm/dd/yyyy")#</td>
  <td>#numberFormat(Salary, "$999,999")#</td>
  <td>#Contract#</td>
</tr>
</cfoutput>
</table>
</body>
</html>
```

3 Save the page as Salary_details.cfm in myapps under the Web root directory.

## Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfquery name="GetSalaryDetails"`<br>`    datasource="CompanyInfo">`<br>`  SELECT`<br>`    Departmt.Dept_Name,`<br>`    Employee.FirstName,`<br>`    Employee.LastName,`<br>`    Employee.StartDate,`<br>`    Employee.Salary,`<br>`    Employee.Contract`<br>`  FROM Departmt, Employee`<br>`  WHERE`<br>`    Departmt.Dept_Name =`<br>`      ’#URL.Dept_Name#’`<br>`  AND Departmt.Dept_ID =`<br>`    Employee.Dept_ID`<br>`  ORDER BY Employee.LastName,`<br>`    Employee.Firstname`<br>`</cfquery>` | Get the salary data for the department whose name was passed in the URL parameter string. Sort the data by the employee’s last and first names. |
| `<table border cellspacing=0 cellpadding=5>`<br>`<tr>`<br>`  <th>Employee Name</td>`<br>`  <th>StartDate</td>`<br>`  <th>Salary</td>`<br>`  <th>Contract?</td>`<br>`</tr>`<br>`<cfoutput query="GetSalaryDetails" >`<br>`<tr>`<br>`  <td>#FirstName# #LastName#</td>`<br>`  <td>#dateFormat(StartDate,`<br>`    "mm/dd/yyyy")#</td>`<br>`  <td>#numberFormat(Salary, "$999,999")#</td>`<br>`  <td>#Contract#</td>`<br>`</tr>`<br>`</cfoutput>`<br>`</table>` | Display the data retrieved by the query as a table. Format the start date into standard month/date/year format, and format the salary with a leading dollar sign comma separator, and no decimal places. |

## Part 2: Making the graph dynamic

1 Open graphdata.cfm in ColdFusion Studio.

2 Edit the `cfgraph` tag for the pie chart so it appears as follows:

```
<cfgraph type="pie"
  query="DeptSalaries"
  valueColumn="SumByDept"
  itemColumn="Dept_Name"
  URL="Salary_Details.cfm?Dept_Name="
  URLColumn="Dept_Name"
```

```
    title="Total Salaries by Department"
    titleFont="Times"
    showValueLabel="rollover"
    valueLabelFont="Times"
    backgroundColor = "##CCFFFF"
    borderWidth = 0
    colorlist="##6666FF,##66FF66,##FF6666,##66CCCC"
    LegendFont="Times">
</cfgraph>
```

3 Save the page.

4 Return to your browser and enter the following URL to view graphdata.cfm: http://127.0.0.1/myapps/graphdata.cfm. Click the slices of the pie chart.

## Reviewing the code

The following table describes the highlighted code and its function:

| Code | Description |
|---|---|
| `URL="Salary_Details.cfm? Dept_Name="` | When the user clicks a data point, call the Salary_Details.cfm page in the current directory, and pass it the parameter named Dept_Name. The parameter value must come from the `URLColumn` attribute. |
| `URLColumn="Dept_Name"` | Complete the URL string with the value from the query Dept_Name field. So, if the Dept_Name is HR, ColdFusion calls the following URL: Salary_Details.cfm?Dept_Name=HR |

Chapter 6

# Making Variables Dynamic

This chapter explains how to use CFML to dynamically populate forms and dynamically generate SQL. It also discusses ways to make sure that variables exist and have valid data because this information is important to effectively use dynamic data.

## Contents

## Dynamically Populating List Boxes

In Chapter 4, you hard-coded a form's list box options. Instead of manually entering the information on a form, you can dynamically populate a list box with database fields. When you code this way, the form page automatically reflects the changes that you make to the database.

You use two tags to dynamically populate a list box:

- Use the `cfquery` tag to retrieve the column data from a database table.
- Use the `cfoutput` tag with the `query` attribute within the `select` tag to dynamically populate the `options` of this form control.

### To dynamically populate a list box:

1 Open the file formpage.cfm in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<html>
<head>
<title>Input form</title>
</head>
<body>
<cfquery name="GetDepartments" datasource="CompanyInfo">
SELECT DISTINCT Location
FROM Departmt
</cfquery>

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

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

<!-- text box -->
<p>
First Name: <input type="Text" name="FirstName" size="20"
          maxlength="35"><br>
Last Name: <input type="Text" name="LastName" size="20"
          maxlength="35"><br>
Salary: <input type="Text" name="Salary" size="10" maxlength="10">
</p>

<!-- list box -->
City

<select name="City">
<cfoutput query="GetDepartments">
<option value="#GetDepartments.Location#">
#GetDepartments.Location#
</option>
</cfoutput>
</select>
```

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

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

<!-- reset button -->
<input type="reset" name="ResetForm" value="Clear Form">

<!-- submit button -->
<input type="submit" name="SubmitForm" value="Submit">
</form>
</body>
</html>
```

3 Save the page as formpage.cfm.

4 View formpage.cfm in a browser.

The changes that you just made appear in the form.

Remember that you need an action page to submit values.

## Reviewing the code

The following table describes the highlighted code and its function:

| Code | Description |
|---|---|
| `<cfquery name="GetDepartments" datasource="CompanyInfo">`<br>`SELECT DISTINCT Location`<br>`FROM Departmt`<br>`</cfquery>` | Get the locations of all departments in the Departmt table. The DISTINCT clause eliminates duplicate location names from the returned query results. |
| **`<select name="City">`**<br>**`<cfoutput query="GetDepartments">`**<br>**`<option value="#GetDepartments.Location#">`**<br>**`#GetDepartments.Location#`**<br>**`</option>`**<br>**`</cfoutput>`**<br>**`</select>`** | Populate the City selection list from the Location column of the GetDepartments query. The control has one option for each row returned by the query. |

## Creating Dynamic Check Boxes and Multiple-Selection List Boxes

When an HTML form contains either a list of check boxes with the same name or a multiple-selection list box (that is, where users can select multiple items from the list), the user's entries are made available as a comma-delimited list with the selected values. These lists can be very useful for a wide range of inputs.

**Note**
If the user does not select a check box or make a selection from a list box, no variable is created. The `cfinsert` and `cfupdate` tags do not work correctly if there are no values. To correct this problem, make the form fields required, use Dynamic SQL, or use `cfparam` to establish a default value for the form field. For more information, see the "Ensuring that Variables Exist" and "Dynamic SQL" sections later in this chapter.

### Check boxes

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

#### Searching numeric values

Suppose you want a user to select one or more departments using check boxes. You then query the database to retrieve detailed information on the selected department(s). The code for a simple set of check boxes that lets the user select departments looks like this:

```
<input type="checkbox"
  name="SelectedDepts"
  value="1">
  Training<br>

<input type="checkbox"
  name="SelectedDepts"
  value="2">
  Marketing<br>

<input type="checkbox"
  name="SelectedDepts"
  value="3">
  HR<br>

<input type="checkbox"
  name="SelectedDepts"
  value="4">
  Sales<br>
</html>
```

The user sees the name of the department, but the `value` attribute of each check box is a number that corresponds to the underlying database primary key for the department's record.

If the user checks the Marketing and Sales items, the value of the SelectedDept form field is "2,4" and you use the SelectedDepts in the following SQL statement:

```
SELECT *
  FROM Departmt
  WHERE Dept_ID IN ( #Form.SelectedDepts# )
```

The ColdFusion Server sends the following statement to the database:

```
SELECT *
  FROM Departmt
  WHERE Dept_ID IN ( 2,4 )
```

## Searching string values

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

The first example searched for department information based on a numeric primary key field called Dept_ID. Suppose, instead, that the primary key is a database field called Dept_Name that contains string values. In that case, your code for check boxes should look like this:

```
<input type="checkbox"
  name="SelectedDepts"
  value="Training">
  Training<br>

<input type="checkbox"
  name="SelectedDepts"
  value="Marketing">
  Marketing<br>

<input type="checkbox"
  name="SelectedDepts"
  value="HR">
  HR<br>

<input type="checkbox"
  name="SelectedDepts"
  value="Sales">
  Sales<br>
```

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

```
SELECT *
  FROM Departmt
  WHERE Dept_Name IN
  (#ListQualify(Form.SelectedDepts,"'")#)
```

**Note**
In SQL, all strings must be surrounded in single quotes. The `ListQualify` function returns a list with the specified qualifying character (here, a single quote) around each item in the list.

If you select the second and fourth check boxes in the form, the following statement gets sent to the database:

```
SELECT *
  FROM Departmt
  WHERE Dept_Name IN (’Marketing’,’Sales’)
```

## Multiple selection lists

ColdFusion treats the result when a user selects multiple choices from a list box (HTML input type `select` with attribute `multiple`) just like results of selecting multiple check boxes. The data made available to your page from any multiple selection list box is a comma-delimited list of the entries selected by the user; for example, a list box could contain the four entries: Training, Marketing, HR, and Sales. If the user selects Marketing and Sales, the form field variable value is Marketing,Sales.

You use multiple selection lists to search a database in the same way that you use check boxes.

### Searching numeric values

Suppose you want the user to select departments from a multiple-selection list box. The query retrieves detailed information on the selected department(s):

```
Select one or more companies to get more information on:
<select name="SelectDepts" multiple>
  <option value="1">Training
  <option value="2">Marketing
  <option value="3">HR
  <option value="4">Sales
</select>
```

If the user selects the Marketing and Sales items, the value of the SelectDepts form field is 2,4.

If this parameter is used in the following SQL statement:

```
SELECT *
  FROM Departmt
  WHERE Dept_ID IN (#form.SelectDepts#)
```

the following statement is sent to the database:

```
SELECT *
  FROM Departmt
  WHERE Dept_ID IN (2,4)
```

## Searching string values

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

```
<select name="SelectDepts" multiple>
  <option value="Training">Training
  <option value="Marketing">Marketing
  <option value="HR">HR
  <option value="Sales">Sales
</select>
```

If the user selects the Marketing and Sales items, the SelectDepts form field value is Marketing,Sales.

Just as you did when using check boxes to search database fields containing string values, use the ColdFusion `ListQualify`function with multiple-selection list boxes:

```
SELECT *
  FROM Departmt
  WHERE Dept_Name IN (#ListQualify(Form.SelectDepts,"’")#)
```

The following statement gets sent to the database:

```
SELECT *
  FROM Departmt
  WHERE Dept_Name IN (’Marketing’,’Sales’)
```

## Ensuring that Variables Exist

The sample code in the previous sections is incomplete. Either the form or the action page should make sure that the SelectDepts variable has a value before it is used in the SQL Select statement. Otherwise, users who do not select any department get an error message. There are several ways to ensure that a variable exists before you use it:

- You can use the `IsDefined` function, as described in the section “Testing for a variable's existence,” in Chapter 4.
- You can use the `cfparam` tag to test for a variable and set it to a default value if it does not exist.
- You can use a form `input` tag with a `hidden` attribute to tell ColdFusion to display a helpful message to any user who does not enter data in a required field.

### Using cfparam to test for variables and set default values

One way to ensure a variable exists is to use the `cfparam` tag, which tests for the variable’s existence and optionally supplies a default value if the variable does not exist. The following code shows the syntax of the `cfparam` tag:

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

**Note**
For information on using the `type` attribute to validate the parameter data type, see the “Using cfparam to validate the data type” section of this chapter.

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

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

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

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

### Example: Testing for variables

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

For example, the following 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. By default, ColdFusion displays an error message; you can also handle the error as described in Chapter 11, “Preventing and Handling Errors” on page 191.

### Example: setting default values

This example uses `cfparam` 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 use `cfparam` to set default values for URL and Form variables, instead of using conditional logic. For example, the action page could have the following code to ensure that a `SelectDepts` variable exists:

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

## Requiring users to enter values in form fields

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

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

```
<input type="hidden" name="FirstName_required">
```

If the user leaves the FirstName field empty, ColdFusion rejects the form submittal and returns a message informing the user that the field is required. You can customize the contents of this error message using the `value` attribute of the hidden field. For example, if you want the error message to read “You must enter your first name,” use the following syntax:

```
<input type="hidden"
  name="FirstName_required"
  value="You must enter your first name.">
```

## Validating Data Types

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

- Using the `cfparam` tag with the `type` attribute to validate any variable.
- Using a form `input` tag with a `hidden` attribute to validate the contents of a form input field.
- Using `cfform` controls that have validation attributes. (For information on using `cfform` tags, see Chapter 9, "Building Dynamic Forms" on page 135.)
- Using the `cfqueryparam` tag in a SQL WHERE clause to validate query parameters.

**Note**
The data validation discussed in this chapter is done by the ColdFusion Server. Validation using `cfform` tags is done using JavaScript in the user's browser, before any data is sent to the server.

### Using cfparam to validate the data type

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

| Type value | Meaning |
|---|---|
| any | any value |
| array | any array value |
| binary | any binary value |
| boolean | true, false, yes, or no |
| date | any value in a valid date, time, or date-time format |
| numeric | any number |
| query | a query object |
| string | a text string or single character |
| struct | a structure |
| UUID | a Universally Unique Identifier (UUID) formatted as XXXXXXXX-XXXX-XXXX-XXXXXXXXXXXXXXXX where X stands for a hexadecimal digit (0-9 or A-F). |
| variableName | a valid variable name |

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

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

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

## Validating form field data types

One limitation of standard HTML forms is that you cannot validate that users input the type or range of data you expect. ColdFusion enables you to do several types of data validation by adding hidden fields to forms.

The following table describes the hidden field suffixes that you can use to do validation:

| Field Suffix | Value Attribute | Description |
|---|---|---|
| `_integer` | Custom error message | Verifies that the user entered a number. If the user enters a floating point value, it is rounded to an integer. |
| `_float` | Custom error message | Verifies that the user entered a number. Does not do any rounding of floating point values. |
| `_range` | MIN=MinValue<br>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 the user entered a date and converts the date into the proper ODBC date format. Will accept most common date forms; for example, 9/1/98; Sept. 9, 1998. |
| `_time` | Custom error message | Verifies that the user correctly entered a time and converts the time to the proper ODBC time format. |
| `_eurodate` | Custom error message | Verifies that the user entered a date in a standard European date format and converts into the proper ODBC date format. |

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

The following procedure creates a simple form for entering a start date and a salary. It uses hidden fields to ensure that you enter data and that the data is in the right format.

This example illustrates another concept that might seem surprising. You can use the same CFML page as both a form page and its action page. Because the only action is to display the values of the two variables that you enter, the action is on the same page as the form.

Using a single page for both the form and action provides the opportunity to illustrate the use of the `IsDefined` function to check that data exists. This way, the form does not show any results until you submit the input.

## To validate the data that users enter in the insert form:

1 Create a new page in ColdFusion Studio.

2 Enter the following text:

```
<html>
<head>
  <title>Simple Data Form</title>
</head>
<body>
<h2>Simple Data Form</h2>

<!--- Form part --->
<form action="datatest.cfm" method="Post">
  <input type="hidden"
    name="StartDate_required"
    value="You must enter a start date.">
  <input type="hidden"
    name="StartDate_date"
    value="Enter a valid date as the start date.">
  <input type="hidden"
    name="Salary_required"
    value="You must enter a salary.">
  <input type="hidden"
    name="Salary_float"
    value="The salary must be a number.">
  Start Date:
  <input type="text"
    name="StartDate" size="16"
    maxlength="16"><br>
  Salary:
  <input type="text"
    name="Salary"
    size="10"
    maxlength="10"><br>
  <input type="reset"
    name="ResetForm"
    value="Clear Form">
  <input type="submit"
    name="SubmitForm"
    value="Insert Data">
</form>
<br>

<!--- Action part --->
<cfif isdefined("Form.StartDate")>
  <cfoutput>
    Start Date is: #DateFormat(Form.StartDate)#<br>
    Salary is: #DollarFormat(Form.Salary)#
    </cfoutput>
  </cfif>
  </html>
```

3 Save the file as datatest.cfm.

4 View the file in your browser, omit a field or enter invalid data, and click the Submit button.

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

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

## Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<form action="actionpage.cfm"`<br>`  method="post">` | Gather the information from this form using the Post method, and do something with it on the page dataform.cfm, which is this page. |
| `<input type="hidden"`<br>`  name="StartDate_required"`<br>`  value="You must enter a start date.">`<br>`<input type="hidden"`<br>`  name="StartDate_date"`<br>`  value="Enter a valid date as the`<br>`  start date.">` | Require input into the StartDate input field. If there is no input, display the error information "You must enter a start date." Require the input to be in a valid date format. If the input is not valid, display the error information "Enter a valid date as the start date." |
| `<input type="hidden"`<br>`  name="Salary_required"`<br>`  value="You must enter a salary.">`<br>`<input type="hidden"`<br>`  name="Salary_float"`<br>`  value="The salary must be a number.">` | Require input into the Salary input field. If there is no input, display the error information "You must enter a salary." Require the input to be in a valid number. If it is not valid, display the error information "The salary must be a number." |
| `Start Date:`<br>`<input type="text"`<br>`  name="StartDate" size="16"`<br>`  maxlength="16"><br>` | Create a text box called StartDate in which users can enter their starting date. Make it exactly 16 characters wide. |

| Code | Description |
|---|---|
| `Salary:`<br>`<input type="text"`<br>`  name="Salary"`<br>`  size="10"`<br>`  maxlength="10"><br>` | Create a text box called Salary in which users can enter their salary. Make it exactly ten characters wide. |
| `<cfif isdefined("Form.StartDate")>`<br>`  <cfoutput>`<br>`    Start Date is:`<br>`      #DateFormat(Form.StartDate)#<br>`<br>`    Salary is:`<br>`      #DollarFormat(Form.Salary)#`<br>`  </cfoutput>`<br>`</cfif>` | Output the values of the StartDate and Salary form fields only if they are defined. They are not defined until you submit the form, so they do not appear on the initial form. Use the `DateFormat` function to display the start date in the default date format. Use the `DollarFormat` function to display the salary with a dollar sign and commas. |

## Checking query parameters with cfqueryparam

You can use the `cfqueryparam` tag to validate SQL query parameters. This tag can validate the value of the SQL query parameter against a SQL data type such as REAL, TIME, or DATE. The `cfqueryparam` tag validates the data as follows:

- If the value does not match the data type, the tag returns an error message.
- If the value matches the data type and the database driver supports data bind parameters, the tag generates a SQL BIND PARAMETER statement to bind the parameter.
- If the database driver does not support bind parameters, the tag just uses the parameter value in the query string.

The `cfqueryparam` tag can also validate parameter value length and its number of decimal places.

**Note**

The `cfqueryparam` tag allows you to specify SQL parameters in queries. It improves performance, maintenance, and security of data queries by improving server-side caching for Oracle databases, supporting updating of long text fields from a SQL statement, and preventing a malicious user from attaching multiple SQL statements to a SQL statement substitution variable. For more information on `cfqueryparam` and its use, see the *CFML Reference*.

The `cfqueryparam` tag can have any of several additional advantages, depending on the database system and Web server software that you are using:

- Some Web servers have security issues in which SQL appended to URL strings can evade system security. `cfqueryparam` can help prevent this problem.
- Some database management systems, including some Oracle releases, limit the size of query text fields to 4K bytes. By using `cfqueryparam` you can overcome this limitation.
- Using `cfqueryparam` can speed database processing by using bind parameters.

The following example shows the use of `cfqueryparam` when valid input is given in the Course_ID variable used as a query parameter. To see what happens when you use invalid data, substitute a text string such as “test” for the integer 12 in the `cfset` statement.

Note that this example uses the cfsnippets database that is provided with ColdFusion, not the CompanyInfo database used in most of this book.

```
<html>
<head>
<title>cfqueryparam Example</title>
</head>

<body>
<h3>cfqueryparam Example</h3>
<cfset course_id=12>
<cfquery name="getFirst" datasource="cfsnippets">
    SELECT *
    FROM courses
    WHERE Course_ID=<cfqueryparam value="#Course_ID#"
    cfsqltype="CF_SQL_INTEGER">
</cfquery>
<cfoutput query="getFirst">
<p>
Course Number: #number#<br>
 Description: #descript#
</p>
</cfoutput>
</body>
</html>
```

## Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfset Course_ID=12>>` | Set the course_ID variable to 12. |
| `<cfquery name="getFirst" DataSource="cfsnippets">` | Query the cfsnippets data source and return the results in the getFirst query object. |
| `SELECT *` `FROM courses` | Select all columns from the courses table. |

| Code | Description |
|---|---|
| `WHERE Course_ID=<cfqueryparam`<br>`    value="#Course_ID#"`<br>`    cfsqltype="CF_SQL_INTEGER">`<br>`</cfquery>` | Only select rows where the Course_ID column equals the value of Course_ID CFML local variable. Validate that the variable value is an integer and, if appropriate for the database driver, use a bind parameter to associate the value with the SQL statement. |
| `<cfoutput query="getFirst">`<br>`<p>`<br>`  Department Number: #number#<br>`<br>`  Description: #descript#`<br>`</p>`<br>`</cfoutput>` | For each row that matches the query, output the contents of the number and descript columns. |

## 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 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 can simply omit the part of the WHERE clause that refers to that field. Or, if a user does not specify a sort order, you can omit the entire ORDER BY clause.

### Implementing dynamic SQL

You implement dynamic SQL 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(s)

<cfif value operator value >
...additional SQL statement(s)
</cfif>

</cfquery>
```

The following code creates an application that lets a user search the CompanyInfo database for employees by first name, last name, minimum salary, contract status, or any combination of these criteria.

## Creating the input form

First, you need to create an input form, which asks for information about several fields in the Employee table. 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 ColdFusion Studio.

2 Enter the following code:

```
<html>
<head>
<title>Input form</title>
</head>
<body>

<!--- Query the Employee table to be able to populate the form --->
<cfquery name="AskEmployees" datasource="CompanyInfo">
SELECT
  FirstName,
  LastName,
  Salary,
  Contract
FROM Employee
</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`.

## Creating the action page

After you create the input form, you can create the action page to process the user's request. This action page determines where the user entered search criteria and searches based only on those criteria.

### To create the action page:

1 Create a new application page in ColdFusion Studio.

2 Enter the following code:

```
<html>
<head>
  <title>Get Employee Data</title>
</head>

<body>
<cfquery name="GetEmployees" datasource="CompanyInfo">
  SELECT *
  FROM Employee
  WHERE

<cfif #form.firstname# is not "">
  Employee.FirstName LIKE '#form.FirstName#%' AND
</cfif>

<cfif #form.lastname# is not "">
  Employee.LastName LIKE '#form.LastName#%' AND
</cfif>

<cfif #form.salary# is not "">
  Employee.Salary >= #form.Salary# AND
</cfif>

<cfif isdefined("Form.Contract")>
  Employee.Contract = 'Yes' AND
<cfelse>
  Employee.Contract = 'No' AND
</cfif>
  0=0

</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 Verify that the results meet the criteria you specify.

## Reviewing the code

The action page `getemp.cfm` builds a SQL statement dynamically based on what the user enters in the form page AskEmp.cfm. The following table describes the highlighted code and its function:

| CFML Code | Description |
|---|---|
| `SELECT *`<br>`    FROM Employee`<br>`    WHERE` | Get the records from the Employee table according to the following conditions. |
| `<cfif #Form.FirstName# is not "">`<br>`    Employee.FirstName LIKE`<br>`        ’#form.FirstName#%’ AND`<br>`</cfif>` | If the user entered anything in the FirstName text box in the form, add "AND Employee.FirstName LIKE ‘[*what the user entered in the FirstName text box*]%'" to the SQL statement. You can use the FirstName variable without ensuring its existence because text boxes pass an empty string if you do not enter text. |
| `<cfif #Form.LastName# is not "">`<br>`    Employee.LastName LIKE`<br>`        ’#form.LastName#%’ AND`<br>`</cfif>` | If the user entered anything in the LastName text box in the form, add "AND Employee.LastName LIKE ‘[*what the user entered in the LastName text box*]%'" to the SQL statement. |
| `<cfif #Form.Salary# is not "">`<br>`    Employee.Salary >=`<br>`        #form.Salary# AND`<br>`</cfif>` | If the user entered anything in the Salary text box in the form, add "AND Employee.Salary >= *[what the user entered in the Salary text box]"* to the SQL statement. |

| **CFML Code** | **Description** |
|---|---|
| `<cfif isDefined("Form.Contract")>`<br>    `Employee.Contract = ‘Yes' AND`<br>`<cfelse>`<br>    `Employee.Contract = 'No' AND`<br>`</cfif>` | If the user selected the Contractor check box, get data for the employees who are contractors; otherwise, get data for employees who are not contractors. The `isdefined` function test for the existence of the Form.Contract variable is needed because the variable only exists if they select the Contractor box. |
|     `0=0` | If none of the conditions are true, the 0=0 statement ensures that the WHERE clause does not result in a SQL syntax error. Instead, the SELECT statement returns the entire table. Putting this statement at the end of the WHERE clause improves security by making it harder to attach extra SQL statements in a dynamic variable, and may affect the database’s optimization of the SQL statement. |

Chapter 7

# Updating Your Database

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

## Contents

## Inserting Data

You usually use two application pages to insert data into a database:

- An insert form
- An insert action page

You can create an insert form with standard HTML form tags or with `cfform` tags (see "Creating Forms with the cfform Tag" on page 136). When the user submits the form, form variables are passed to a ColdFusion action page that performs an insert operation (and whatever else is called for) on the specified data source. The insert action page can contain either a `cfinsert` tag or a `cfquery` tag with a SQL INSERT statement. The insert action page should also contain a message for the end user.

## Creating an HTML insert form

The following procedure creates a form using standard HTML tags.

#### To create an insert form:

1 Create a new application page in ColdFusion 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="insertaction.cfm" method="post">
  Employee ID:
  <input type="text" name="Emp_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="35" maxlength="50"><br>
  Department Number:
  <input type="Text" name="Dept_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" value="Clear Form">
  <!-- Submit button -->
  <input type="Submit" value="Submit">
</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

If you use the `cfinsert` tag in the action page to insert the data into the database, you should follow these rules for creating the form page:

- You only need to create HTML form fields for the database fields into which you want to insert data.
- By default, `cfinsert` inserts all of the form's fields into the database table fields with the same names. For example, it puts the Form.Emp_ID value in the database Emp_ID field. The tag ignores any form fields with no corresponding database column name. You can also use the `formfields` attribute to specify the fields you want to insert.

## Creating an action page to insert data

You can use the `cfinsert` tag or `cfquery` tag to create an action page that inserts data into a database.

### Creating an insert action page with cfinsert

The `cfinsert` tag is the easiest way to handle simple inserts from either a `cfform` or an HTML form. This tag inserts data from all the form fields with names that match database field names.

#### To create an insert action page with cfinsert:

1 Create a new application page in ColdFusion Studio.

2 Enter the following code:

```
<!--- Make the contract variable be No if it is not set (check box is
       empty) --->
<cfif not isdefined("Form.Contract")>
   <cfset Form.contract = "No">
</cfif>

<!--- Insert the new record --->
<cfinsert datasource="CompanyInfo" tablename="Employee">

<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 insertaction.cfm.

4 View insertform.cfm in a browser, enter values, and click the Submit button.

5 The data is inserted into the Employee table and the message displays.

## Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfif not isdefined("Form.Contract")>`<br>`   <cfset Form.contract = "No">`<br>`</cfif>` | If the user clears the Contractor check box, no value gets passed to the action page. The database field must have a value, so check the Form.contract variable and set it to No if it is not defined. |
| `<cfinsert datasource="CompanyInfo"`<br>`  tablename="Employee">` | Create a new row in the Employee table of the CompanyInfo database. Insert data from the form into database fields with the same names as the form fields. |
| `<cfoutput>You have added`<br>`  #Form.FirstName# #Form.Lastname#`<br>`  to the employees database.`<br>`</cfoutput>` | Inform the user that the data was inserted into the database. |

**Note**

If you use form variables in `cfinsert` or `cfupdate` tags, ColdFusion automatically validates any form data it sends to numeric, date, or time data database columns. You can use the hidden field validation functions for these fields to display a custom error message.

## Creating an insert action page with cfquery

For more complex inserts from a form submittal you can use a SQL INSERT statement in a `cfquery` tag instead of 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 page with cfquery:

1 Rename (or delete) the insertaction.cfm page that you created in the previous section.

2 Create a new application page in ColdFusion Studio.

3 Enter the following code:

```
<!--- Make the contract variable be No if it is not set
     (check box is empty) --->
<cfif not isdefined("Form.Contract")>
   <cfset form.contract = "No">
</cfif>

<!--- Insert the new record --->
<cfquery name="AddEmployee" datasource="CompanyInfo">
  INSERT INTO Employee
  VALUES ('#Form.Emp_ID#', '#Form.FirstName#',
         '#Form.LastName#', '#Form.Dept_ID#',
         '#Form.StartDate#', '#Form.Salary#', '#Form.Contract#'
</cfquery>

<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>
```

4 Save the page as insertaction.cfm.

5 View insertform.cfm in a browser, enter values, and click Submit.

6 The data is inserted into the Employee table and the message displays.

### Reviewing the code

The following table describes the highlighted code and its function:

| Code | Description |
|---|---|
| `<cfquery name="AddEmployee"`<br>`  datasource="CompanyInfo">`<br>`  INSERT INTO Employee`<br>`  VALUES ('#Form.Emp_ID#',`<br>`    '#Form.FirstName#',`<br>`    '#Form.LastName#',`<br>`    '#Form.Dept_ID#',`<br>`    '#Form.StartDate#',`<br>`    '#Form.Salary#',`<br>`    '#Form.Contract#')`<br>`</cfquery>` | Use a `cfquery` tag to insert a new row into the Employee table of the CompanyInfo Database. Specify each form field to be added. Because the form and database field names are identical, you do not have to specify the database field names in the query. |

## Updating Data

You usually use two application pages to update data in a database:

- An update form
- An update action page

You can create an update form with `cfform` tags or HTML form tags. The update form calls an update action page, which can contain either a `cfupdate` tag or a `cfquery` tag with a SQL UPDATE statement. The update action page should also contain a 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 Emp_ID would be unique to each record.
- An update form is usually populated with existing record data because the form's purpose is to update 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 ColdFusion Studio.
2. Edit the page so that it appears as follows:

```
<cfquery name="GetRecordtoUpdate"
  datasource="CompanyInfo">
  SELECT *
    FROM Employee
    WHERE Emp_ID = #URL.Emp_ID#
</cfquery>

<html>
<head>
<title>Update Form</title>
</head>

<body>

<cfoutput query="GetRecordtoUpdate">
<form action="updateaction.cfm" method="Post">
  <input type="Hidden" name="Emp_ID"
    value="#Emp_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="Dept_ID" value="#Dept_ID#"><br>
  Start Date:
  <input type="text" name="StartDate" value="#StartDate#"><br>
  Salary:
  <input type="text" name="Salary" value="#Salary#"><br>
  Contractor:
  <cfif #Contract# IS "Yes">
    <input type="checkbox" name="Contract" checked>Yes<br>
  <cfelse>
    <input type="checkbox" name="Contract">Yes<br>
  </cfif>
<br>
    <input type="Submit" value="Update Information">
</form>
</cfoutput>
</body>
</html>
```

3. Save the page as updateform.cfm.
4. View updateform.cfm in a browser by specifying the page URL and an Employee ID, for example, http://localhost/myapps/updateform.cfm?Emp_ID=3.

## Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfquery name="GetRecordtoUpdate"`<br>`  datasource="CompanyInfo">`<br>`  SELECT *`<br>`  FROM Employee`<br>`  WHERE Emp_ID = #URL.Emp_ID#`<br>`</cfquery>` | Query the CompanyInfo data source and return the records in which the employee ID matches what was entered in the URL that called this page. |
| `<cfoutput query="GetRecordtoUpdate">` | Make the results of the GetRecordtoUpdate query available as variables in the form created on the next line. |
| `<form action="updateaction.cfm"`<br>`  method="Post">` | Create a form whose variables will be processed on the updateaction.cfm action page. |
| `<input type="Hidden" name="Emp_ID"`<br>`  value="#Emp_ID#"><br>` | Use a hidden input field to pass the employee ID to the action page. |
| `First Name:`<br>`<input type="text" name="FirstName"`<br>`  value="#FirstName#"><br>`<br>`Last Name:`<br>`<input type="text" name="LastName"`<br>`  value="#LastName#"><br>`<br>`Department Number:`<br>`<input type="text" name="Dept_ID"`<br>`    value="#Dept_ID#"><br>`<br>`Start Date:`<br>`<input type="text" name="StartDate"`<br>`  value="#StartDate#"><br>`<br>`Salary:`<br>`<input type="text" name="Salary"`<br>`  value="#Salary#"><br>` | Populate the fields of the update form. This example does not use any ColdFusion formatting functions to “clean up” the form. As a result, the start dates look like 1985-03-12 00:00:00 and the salaries do not have dollar signs or commas. The user can replace the information in any field using any valid input format for the data. |
| `Contractor:`<br>`<cfif #Contract# IS "Yes">`<br>`  <input type="checkbox" name="Contract"`<br>`    checked>Yes<br>`<br>`<cfelse>`<br>`  <input type="checkbox" name="Contract">`<br>`    Yes <br>`<br>`</cfif>`<br>`<br>`<br>`<input type="Submit" value="Update`<br>`  Information">`<br>`</form>`<br>`</cfoutput>` | The Contractor field needs special treatment because a check box displays and sets its value. Use the `cfif` structure to put a check mark in the check box if the Contract field value is Yes, and leave the box empty otherwise. |

## Creating an action page to update data

You can create an action page to update data with either the `cfupdate` tag or `cfquery` with the UPDATE statement.

### Creating an update action page with cfupdate

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

To use `cfupdate`, you must include the field or fields that make up the primary key in your form submittal. The `cfupdate` tag automatically detects the primary key fields in the table that you are updating and looks for them in the submitted form fields. ColdFusion uses the primary key fields to select the record to update. (Therefore, you cannot update the primary key value itself.) It then uses the remaining form fields that are submitted to update the corresponding fields in the record. Your form only needs to have fields for the database fields that you want to change.

#### To create an update page with cfupdate:

1 Create a new application page in ColdFusion Studio.

2 Enter the following code:

```
<cfif not isdefined("Form.Contract")>
  <cfset form.contract = "No">
<cfelse>
  <cfset form.contract = "Yes">
</cfif>

<cfupdate datasource="CompanyInfo"
  tablename="Employee">

<html>
<head>
  <title>Update Employee</title>
</head>
<body>

<h1>Employee Updated</h1>
<cfoutput>
You have updated the information for #Form.FirstName#
          #Form.LastName# in the Employees database.
</cfoutput>

</body>
</html>
```

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

4 View `updateform.cfm` in a browser by specifying the page URL and an Employee ID, for example, `http://localhost/myapps/updateform.cfm?Emp_ID=3`. Enter new values in any of the fields, and click the Submit button.

5 The data is updated in the Employee table and the message appears.

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfif not isdefined("Form.Contract")> <cfset Form.contract = "No"> <cfelse> <cfset form.contract = "Yes"> </cfif>` | If the user clears the Contractor check box, no value gets passed to the action page. Also, the database field must have a value of Yes or No. Test the Form.contract variable and set it to No if it is not defined and Yes if it is defined. |
| `<cfupdate datasource="CompanyInfo" tablename="Employee">` | Update the record in the database that matches the primary key on the form (the Emp_ID). Update all fields in the record with names that match the names of controls on the form. |
| `<cfoutput> You have updated the information for #Form.FirstName# #Form.LastName# in the Employees database. </cfoutput>` | Inform the user that the change was made successfully. |

## Creating an update action page with cfquery

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

### To create an update page with cfquery:

1 Open updatepage.cfm.

2 Replace the `cfupdate` tag with the highlighted `cfquery` code.:

```
<cfif not isdefined("Form.Contract")>
  <cfset form.contract = "No">
<cfelse>
  <cfset form.contract = "Yes">
</cfif>

<cfquery name="UpdateEmployee" datasource="CompanyInfo">
  UPDATE Employee
  SET FirstName = ’#Form.Firstname#’,
    LastName = ’#Form.LastName#’,
    Dept_ID = ’#Form.Dept_ID#’,
    StartDate = ’#Form.StartDate#’,
    Salary = ’#Form.Salary#’
  WHERE Emp_ID = #Form.Emp_ID#
</cfquery>
```

```
<h1>Employee Updated</h1>
<cfoutput>
You have updated the information for #Form.FirstName#
          #Form.LastName# in the Employees database.
</cfoutput>
```

3 Save the page.

4 View updateform.cfm in a browser by specifying the page URL and an Employee ID, for example, http://localhost/myapps/updateform.cfm?Emp_ID=3. Enter new values in any of the fields, and click Submit.

5 The data is updated into the Employee table and the message displays.

## Reviewing the code

The following table describes the highlighted code and its function:

| Code | Description |
|---|---|
| `<cfquery name="UpdateEmployee" datasource="CompanyInfo">`<br>`UPDATE Employee`<br>`SET FirstName = '#Form.Firstname#',`<br>`LastName = '#Form.LastName#',`<br>`Dept_ID = '#Form.Dept_ID#',`<br>`StartDate = '#Form.StartDate#',`<br>`Salary = '#Form.Salary#'`<br>`WHERE Emp_ID = #Form.Emp_ID#`<br>`</cfquery>` | Update the record in the database that matches the primary key on the form, (Emp_ID). Update all fields in the record with names that match the names of controls on the form. Because #From.Emp_ID# is numeric, you do not enclose it in quotes. |

## Deleting Data

You use a `cfquery` tag with a SQL DELETE statement to delete data from a database.

## Deleting a single record

To delete a single record, use the table's primary key in the WHERE condition of a SQL DELETE statement. In the example, the Emp_ID field is the primary key, so the SQL Delete statement is as follows:

```
DELETE FROM Employee WHERE Emp_ID = #Form.Emp_ID#
```

You often want to see the data before you delete it. The following example displays the data to be deleted by reusing the form page used to insert and update data. Any data that you enter in the form before submitting it is not used, so you can use a table to display the record to be deleted instead.

### To delete one record from a database:

1 Open the file updateform.cfm in ColdFusion Studio.

2 Change the title to "Delete Form" and the text on the submit button to "Delete Record".

3 Change the `form` tag so that it appears as follows:

```
<form action="deleteaction.cfm" method="Post">
```

4 Save the modified file as deleteform.cfm.

5 Create a new application page in ColdFusion Studio.

6 Enter the following code:

```
<cfquery name="DeleteEmployee"
  datasource="CompanyInfo">
  DELETE FROM Employee
  WHERE Emp_ID = #Form.Emp_ID#
</cfquery>

<html>
<head>
  <title>Delete Employee Record</title>
</head>
<body>
<h3>The employee record has been deleted.</h3>
<P><cfoutput>
You have deleted #Form.FirstName# #Form.LastName# from the Employees
          database.
</cfoutput></P>
</body>
</html>
```

7 Save the page. as deleteaction.cfm.

8 View deleteform.cfm a browser by specifying the page URL and an Employee ID, for example, http://localhost/myapps/updateform.cfm?Emp_ID=3. and click the Submit button.

The employee is deleted from the Employee table and the message displays.

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfquery name="DeleteEmployee"`<br>`  datasource="CompanyInfo">`<br>`  DELETE FROM Employee`<br>`  WHERE Emp_ID = #Form.Emp_ID#`<br>`</cfquery>` | Delete the record in the database whose Emp_ID column matches the Emp_ID (hidden) field on the form. Since the Emp_ID is the table's primary key, only one record gets deleted. |
| `<cfoutput>`<br>`You have deleted #Form.FirstName#`<br>`  #Form.LastName# from the`<br>`  Employees database.`<br>`</cfoutput>` | Inform the user that the record was deleted. |

## Deleting multiple records

You can use a SQL condition to delete several records. The following example deletes the records for everyone in the Sales department (which has Dept_ID number 4) from the Employee table:

```
DELETE FROM Employee
WHERE Dept_ID = 4
```

To delete all the records from the Employee table, you use the following code:

```
DELETE FROM Employee
```

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

Chapter 8

# 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.

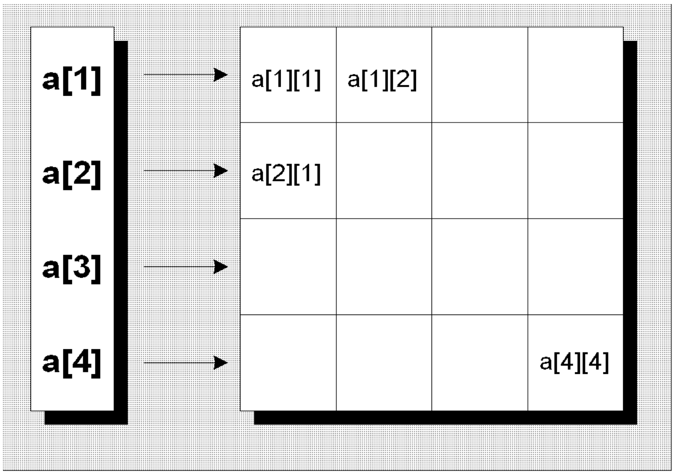
## Contents

## 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 two-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, you can store their shopping cart contents in an array. This allows you to make changes easily without committing the information, which the user can change before completing the transaction, to a database.

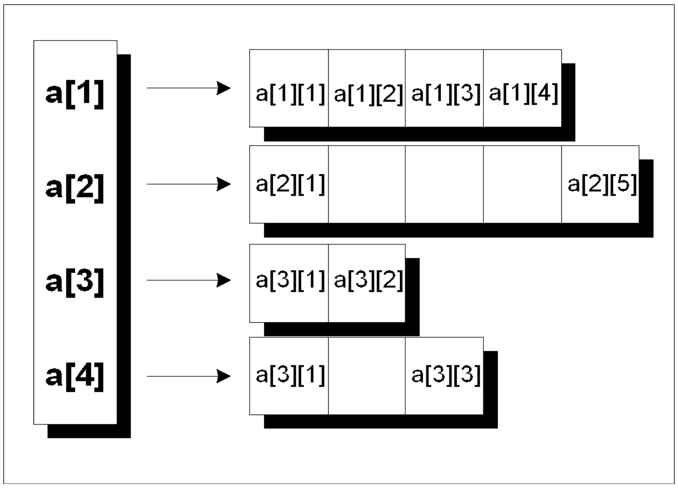
### Conventional fixed-size 2D array

A 2D array is like a cube made up of individual cells, as the following figure shows:



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 rows of differing lengths based on the data that has been added or removed. The following figure represents a ColdFusion 2D array:

## ColdFusion dynamic 2D array



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

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.

## Basic Array Techniques

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

As an example, suppose you declare a 1D 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">
```

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

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

If you remove data from an array, the array resizes dynamically. Use the `ArrayDeleteAt` function to delete data from the array at a particular index, rather than set the data value to 0 or the empty string:

```
<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
```

## 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 that you want to create.

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

```
<cfset mynewarray[4]=Form.emailaddress>
```

## Creating 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]=3>
<cfset biggerarray[2][1][1]=myotherarray>
<cfset biggerarray[2][1][1][4][2]="reality">

<cfset biggestarray=ArrayNew(3)>
<cfset biggestarray[3][1][1]=biggerarray>
<cfset biggestarray[3][1][1][2][3][1]="This is complex">
```

## Adding elements to an array

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

```
<cfset myarray[5]=form.variable>
```

But you can also use a number of array functions to add data to an array. You can use the following functions:

| Function | Description |
|---|---|
| `ArrayAppend` | Creates a new array index at the end of the array |
| `ArrayPrepend` | Creates a new array index at the beginning of the array |
| `ArrayInsertAt` | Inserts an array index and data |

When you insert an array index with `ArrayInsertAt`, 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 Reference*.

**Note**
Because ColdFusion arrays are dynamic, if you add or delete an element from the middle of an array, higher-numbered index values 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]`.

Now you can 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 --->

<!--- Declare two one-dimensional arrays for the first and last names
--->
<cfset firstname=arraynew(1)>
<cfset lastname=arraynew(1)>

<!--- Assign first names directly to the firstname array --->
<cfset firstname[1]="Coleman">
<cfset firstname[2]="Charlie">
<cfset firstname[3]="Dexter">

<!--- Declare the full name array --->
<cfset fullname=arraynew(1)>

<!--- Add the firstname array to index 1 and the lastname array to
       index 2 of the fullname array --->
<cfset fullname[1]=firstname>
<cfset fullname[2]=Lastname>

<!--- Add the last names using reference to the fullname array --->

<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>
  #fullname[1][1]# #fullname[2][3]#<br>
</cfoutput>
```

Note that because this is a full 2D array, you can easily output names that do not make sense in the real world.

## Additional referencing methods

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

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

## Populating Arrays with Data

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

In particular you should master the following basic techniques:

- Populating an array with `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, or one dimension of a multidimensional 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 do not need to add data to it right away. One reason to do this is so that you can refer to all the array indexes. If you refer to an array index that does not contain some value, such as an empty string, you get an error.

The `ArraySet` function has the following form:

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

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`.

The following example uses a `cfloop` tag and the `MonthAsString` function to populate a simple 1D array with the names of the months. A second `cfloop` outputs data in the array to the browser.

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

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

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

## Using nested loops for 2D and 3D arrays

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

### Nesting cfloops for a 2D array

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

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

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

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

### Nesting CFLOOPs for a 3D array

For 3D arrays, you simply nest an additional `cfloop`. (This example does not set the array values first. You can try doing it as an exercise.)

```
<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 of the myquery query.
- Inside a `cfloop query=` loop, you do not have to specify the query name to reference the query's variables.

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

`<cfset arrayname[`*`x`*`]=`*`column`*`[`*`row`*`]>`

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

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

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

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

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

## About Structures

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

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

A structure's **key** must be a string. The **values** associated with the key can be anything; for example, a string, an integer, an array, or another structure. Because structures can contain both structures and arrays, they provide a very powerful and flexible mechanism for representing complex data.

You can use structures as **associative arrays**. When used as associative arrays, structures index repetitive data by string keys rather than by integers. For example, 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"]`.

## Structure notation

ColdFusion supports three types of notation for structures. Which notation you use depends on your needs:

| Notation | Description |
|---|---|
| Object.property | Use to refer to values in a structure. You can refer to a property, *prop*, of an object, *obj*, as *obj.prop*. This notation is useful for simple assignments, as in this example:<br>`depts.John="Sales"`<br>Use this notation only when you know the property names (keys) in advance and they are strings, with no special characters, numbers, or spaces. You cannot use the dot notation when the property, or key, is dynamic. |
| Associative arrays | If you do not know the key name is in advance, or it contains spaces, numbers or special characters, you can use associative array notation. This notation uses structures as arrays with string indexes, for example, depts["John"] or depts["John Doe"]="Sales.". |
| Structure functions | Use structure functions when you cannot use the simpler syntax styles.<br>You must use structure functions to create and remove structures and their elements, including renaming keys. You do not need structure functions to access value data in structures or to change the values in the key value pairs. |

# Creating and Using Structures

This section explains how to use the structure functions to create and use structures in ColdFusion. The sample code in this section uses a structure called *employee,* which is used to add new employees to a corporate information system.

## Creating structures

You 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 elements to structures

After you create 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. You can use it to specify whether an existing key should be overwritten. 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")>
```

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#")>
```

## Updating values in structures

You can update structure element values in a `cfset` tag or a `SructUpdate` function.

### Updating a structure with cfset

You can use the `cfset` tag to update structure values (but not keys). For example, the following code uses `cfset` and Object.property notation to change John’s department from Sales to Marketing. It then uses associative array notation to change his department to Facilities. Each time the department changes, it outputs the results:

```
<cfset departments=structnew()>
<cfset value=StructInsert(departments, "John", "Sales")>
<cfoutput>
  Before the first change, John was in the #departments.John#
    Department<br>
</cfoutput>
<cfset Departments.John = "Marketing">
<cfoutput>
  After the first change, John is in the #departments.John#
    Department<br>
</cfoutput>
<cfset Departments.John = "Facilities">
<cfoutput>
  After the second change, John is in the #departments.John#
    Department<br>
</cfoutput>
```

### Updating a structure with StructUpdate

You can also use the StructUpdate function to change the value associated with a specific key. Because `StructUpdate` is a ColdFusion function, you must use it inside a ColdFusion tag. In some cases, you can use the `cfoutput` or `cfset` tag. You can also use the `cfscript` tag to tell ColdFusion to run a function. The following example uses a `StructUpdate` function in a `cfscript` tag to change a structure value. Note that you must follow a statement in a `cfscript` tag with a semicolon.

```
<cfset departments=structnew()>
<cfset value=StructInsert(departments, "John", "Sales")>
<cfoutput>
  Before the change, John was in the #departments.John# Department<br>
</cfoutput>
<cfscript>StructUpdate(Departments, "John", "Marketing"); </cfscript>
<cfoutput>
  After the change, John is in the #departments.John# Department<br>
</cfoutput>
```

For more information on using `cfscript`, see Chapter 13, “Extending ColdFusion Pages with CFML Scripting” on page 243

## 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 and its value

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)>
```

You can also use the `StructFind` function to find a key and return its value, as in this example:

```
<cfset keyvalue=StructFind(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 that 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* does not 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, structures are used to collect information from a form, structinsert.cfm, and to submit that information to a custom tag at addemployee.cfm.

This examples show how you can use a structure to pass information to a custom tag, named `cf_addemployee`. For information on creating and using custom tags, see "Reusing Code" on page 171.

### 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>

<!--- Action page code for the form at the bottom of this page --->

<!--- Establish parameters for first time through --->
<cfparam name="Form.firstname" default="">
<cfparam name="Form.lastname" default="">
<cfparam name="Form.email" default="">
<cfparam name="Form.phone" default="">
<cfparam name="Form.department" default="">

<!--- If at least the firstaname form field is passed, create
   a structure named employee and add values --->
<cfif #form.firstname# eq "">
 <p>Please fill out the form.
<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>

<!--- Display results of creating the structure --->
  <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>

<!--- The form for adding the new employee information --->
<hr>
<form action="structinsert.cfm" method="Post">
<p>First Name:&nbsp;
<input name="firstname" type="text" hspace="30" maxlength="30">
<p>Last Name:&nbsp;
<input name="lastname" type="text" hspace="30" maxlength="30">
<p>EMail:&nbsp;
<input name="email" type="text" hspace="30" maxlength="30">
<p>Phone:&nbsp;
<input name="phone" type="text" hspace="20" maxlength="20">
<p>Department:&nbsp;
<input name="department" type="text" hspace="30" maxlength="30">

<p>
```

```
<input type="Submit" value="OK">
</form>

</body>
</html>
```

### Example file addemployee.cfm

```
<!--- This file is an example of a custom tag used to add employees.
Employee information is passed through the employee structure (the
empinfo attribute). For databases that do not support automatic key
generation, you must also add the Emp_ID. --->


<cfif structisempty(attributes.empinfo)>
  <cfoutput>Error. No employee data was passed.</cfoutput>
  <cfexit method="ExitTag">
<cfelse>
  <!--- Add the employee --->
  <!--- If auto key generation is not supported,
    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>
```

## Looping through structures

You can loop through a structure to output its contents as illustrated in the following example. Note that when you enumerate key-value pairs using a loop, the keys appear in uppercase.

```
<!--- Create a structure and set its contents --->
<cfset departments=structnew()>

<cfset val=StructInsert(departments, "John", "Sales")>
<cfset val=StructInsert(departments, "Tom", "Finance")>
<cfset val=StructInsert(departments, "Mike", "Education")>

<!--- Build a table to display the contents --->
<cfoutput>
```

```
<table cellpadding="2" cellspacing="2">
  <tr>
  <td><b>Employee</b></td>
  <td><b>Department</b></td>
  </tr>

<!--- 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

Use the following functions to create and manage structures in ColdFusion applications:

| 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. |
| `StructNew` | Returns a new structure. |
| `StructUpdate` | Updates the specified key with the specified value. |

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

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

Chapter 9

# 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. You can use these controls without writing a line of Java code.

## Contents

## Creating Forms with the cfform Tag

You already learned how to use HTML forms to gather user input (see "Using Forms to Specify the Data to Retrieve" on page 40). 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
- Predefined ColdFusion Java applet based controls, including trees, sliders, and grids
- Custom Java applets that act as form elements

With `cfform`, you can access these Java applet-based controls without knowing the Java language, and you do not 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 do not have to write separate CFML code specifically for input validation, as you do in HTML forms.

## Using HTML and cfform

ColdFusion generates HTML forms dynamically from `cfform` tags and passes through to the browser any HTML code it finds in the form. As a result, you can also do the following:

- You can use the HTML `form` tag and form control tags in combination with the `cfform` tag.
- You can use the `passthrough` attribute of the `cfform`, `cfinput`, and `cfselect` tags to enter any HTML attributes that are not explicitly allowed in these tags. The attribute values are passed through to the HTML generated by these form tags.
- You can replace your existing HTML `form` tags with `cfform` and your forms will work fine.

## cfform controls

The following table describes the controls ColdFusion controls you use in forms created using `cfform`:

| **Control** | **Description** |
|---|---|
| `cfgrid` | A Java applet-based control that creates a data grid you can populate from a query or by defining the contents of individual cells. You can also use grids to insert, update, and delete records from a data source. |
| `cfslider` | A Java applet-based control that defines a slider. |
| `cfinput` | Places radio buttons, check boxes, text input boxes, and password entry boxes in your form (not a Java applet). |
| `cftree and cftreeitem` | Java applet-based controls that define a tree control and individual tree control items. |
| `cftextinput` | A Java applet-based control that defines a text input box. |
| `cfselect` | Defines a drop-down list box not a Java applet). |
| `cfapplet` | Embed your own Java applets in the form. |

## Preserving input data with preservedata

The `cfform` attribute `preservedata` tells ColdFusion to continue displaying the data that a user entered in the form after the user submits the form. Data is preserved in the `cftext`, `cfslider`, `cftextinput`, and `cftree` controls and `in cfselect` controls populated by queries. You can retain data on the form in the following circumstances:

- The form and action are on a single cfml page.
- The action page has a form that contains controls with the same names as the corresponding controls on the form page.

For example, if you save this form as preseve.cfm, it continues to display any text that you enter after you submit it:

```
<cfform action="preserve.cfm" method="POST" preservedata="Yes">
<p>Please enter your name:
<cfinput type="Text" name="UserName" required="Yes">
<p><input type="Submit" name=""> <input type="RESET">
</cfform>
```

## Usage notes

- In `cftree`, the `preservedata` attribute causes the tree to expand the tree to the previously selected element. For this to work correctly, you must also set the `completePath` attribute to True.
- The `preservedata` attribute has no effect on `cfgrid`. If you populate the control from a query, you should update the data source with the new data (typically by using `cfgridupdate`) before redisplaying the grid. The grid then displays the updated database information.

## Browser considerations

The applet-based controls for `cfform`—`cfgrid`, `cfslider`, `cftextinput`, and `cftree`—use JavaScript and Java to display their content. To allow them to display consistently across a variety of browsers, these applets use the Java plug-in, and not the browser's native Java runtime. As a result, they are independent of the level of Java support provided by the browser.

ColdFusion downloads and installs the browser plug-in if necessary. Some browsers display a single permission dialog box asking you to confirm the plug-in install. Other browsers, particularly older versions of Netscape, require you to navigate some simple option screens.

Because the controls use JavaScript to return data to ColdFusion, if you disable JavaScript in your browser it cannot properly run forms that contain these controls. In that case, the controls still display, but data return and validation does not work and you may receive a JavaScript error.

Because Java is handled by the plug-in and not directly by the browser, disabling Java execution in the browser should not affect the operation of the controls. If for some other reason, however, the browser is unable to render the controls as requested, a "notsupported" message displays in place of the control.

You can use the `cfform` tag's `notsupported` attribute to specify an alternate error message.

## Input Validation with cfform Controls

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

| Data type | Description |
|---|---|
| Date | Verifies US date entry in the form mm/dd/yyyy (where the year can have one through four digits). |
| Eurodate | Verifies valid European date entry in the form dd/mm/yyyy (where the year can have one through four digits). |
| Time | Verifies a time entry in the form hh:mm:ss. |
| Float | Verifies a floating point entry. |
| Integer | Verifies an integer entry. |
| Telephone | Verifies a telephone entry. You must enter telephone data as ###-###-####. You can replace the hyphen separator (-) with a blank. The area code and exchange must begin with a digit between 1 and 9. |
| Zipcode | (U.S. formats only) Number can be a 5-digit or 9-digit zip in the form #####-####. You can replace the hyphen separator (-) with a blank. |
| Creditcard | Blanks and dashes are stripped and the number is verified using the mod10 algorithm. |
| Social_security_number | You must enter the number as ###-##-####. You can replace the hyphen separator (-) with a blank. |
| Regular_expression | Matches the input against a JavaScript regular expression pattern. You must use the `pattern` attribute to specify the regular expression. Any entry containing characters that matches the pattern is valid. |

When you specify an input type in the `validate` attribute, ColdFusion tests for the specified input type when you submit the form, and submits form data only on a successful match. A successful form submission returns the value True and returns the value False if validation fails.

## Validating with regular expressions

You can use **regular expressions** to match and validate the text that users enter in `cfinput` and `cftextinput` tags. Ordinary characters are combined with special characters to define the match pattern. The validation succeeds only if the user input matches the pattern.

Regular expressions allow you to check input text for a wide variety of conditions. For example, if a date field must only contain dates between 1950 and 2050, you can create a regular expression that matches only numbers in that range. You can concatenate simple regular expressions into complex search criteria to validate against complex patterns, such as any of several words with different endings.

You can use ColdFusion variables and functions in regular expressions. The ColdFusion Server evaluates the variables and functions before the regular expression is evaluated. For example, you can validate against a value that you generate dynamically from other input data or database values.

**Note**
The rules listed in this section are for JavaScript regular expressions, and apply to the regular expressions used in `cfinput` and `cftextinput` tags only. These rules differ from those used by the ColdFusion functions `REfind`, `REreplace`, `REfindnocase`, and `RERplacenocase` and in ColdFusion Studio. For information on regular expressions used in ColdFusion functions, see “Using Regular Expressions in Functions” on page 259.

### Special characters

Because special characters are the operators in regular expressions, in order to represent a special character as an ordinary one, you need to precede it with a backslash. For example, use a double backslash (\\) to represent a backslash.

### Single-character regular expressions

The following rules govern regular expressions that match a single character:

- Special characters are: `+ * ? . [ ^ $ ( ) { | \`
- Any character that is not a special character or escaped by being preceded by the backslash (\) matches itself.
- A backslash (\) followed by any special character matches the literal character itself, that is, the backslash escapes the special character.
- A period (.) matches any character except newline.
- A set of characters enclosed in brackets ([]) is a one-character regular expression that matches any of the characters in that set. For example, "[akm]" matches an "a", "k", or "m". Note that if you want to include ] (closing square bracket) in square brackets it must be the first character. Otherwise, it does not work even if you use \].
- A dash can indicate a range of characters. For example, "[a-z]" matches any lowercase letter.

- If the first character of a set of characters in bracket is the caret (^), the expression matches any character except those in the set. It does not match the empty string. For example: [^akm] matches any character except "a", "k", or "m". The caret loses its special meaning if it is not the first character of the set.
- You can make regular expressions case insensitive by substituting individual characters with character sets, for example, [Nn][Ii][Cc][Kk].
- You can use the following escape sequences to match specific characters or character classes:

| Escape Seq | Matches | Escape Seq | Meaning |
|---|---|---|---|
| [\b] | Backspace | \s | Any of the following white space characters: space, tab, form feed, and line feed. |
| \b | A word boundary such as a space | \S | Any character except the white space characters matched by \s |
| \B | A non-word boundary | \t | Tab |
| \c*X* | The control character Ctrl-*x.* For example, \cv matches Ctrl-v, the usual control character for pasting text. | \v | Vertical tab |
| \d | A digit character [0-9] | \w | An alphanumeric character or underscore. The equivalent of [A-Za-z0-9_] |
| \D | Any character except a digit | \W | Any character not matched by \w. The equivalent of [^A-Za-z0-9_] |
| \f | Form feed | *n* | a backreference to the nth expression in parentheses. See “Backreferences” |
| \n | Line feed | \o*octal* | The character represented in the ASII character table by the specified octal number |
| \r | Carriage return | \x*hex* | The character represented in the ASCII character table by the specified hexadecimal number |

## Multicharacter regular expressions

Use the following rules to build a multicharacter regular expression:

- Parentheses group parts of regular expressions together into a subexpression that can be treated as a single unit. For example, (ha)+ matches one or more instances of "ha".
- A one-character regular expression or grouped subexpression followed by an asterisk (*) matches zero or more occurrences of the regular expression. For example, [a-z]* matches zero or more lowercase characters.

- A one-character regular expression or grouped subexpression followed by a plus (+) matches one or more occurrences of the regular expression. For example, [a-z]+ matches one or more lowercase characters.
- A one-character regular expression or grouped subexpression followed by a question mark (?) matches zero or one occurrences of the regular expression. For example, xy?z matches either "xyz" or "xz".
- The carat (^) at the beginning of a regular expression matches the beginning of the field.
- The dollar sign ($) at the end of a regular expression matches the end of the field.
- The concatenation of regular expressions creates a regular expression that matches the corresponding concatenation of strings. For example, [A-Z][a-z]* matches any capitalized word.
- The OR character (|) allows a choice between two regular expressions. For example, jell(y|ies) matches either "jelly" or "jellies".
- Braces ({}) are used to indicate a range of occurrences of a regular expression, in the form {m, n} where m is a positive integer equal to or greater than zero indicating the start of the range and n is equal to or greater than m, indicating the end of the range. For example, (ba){0,3} matches up to three pairs of the expression "ba". The form {m,} requires at least m occurrences of the preceding regular expression. The form {m} requires exactly m occurrences of the preceding regular expression.The syntax {,n} is not allowed.

## Backreferences

Backreferencing allows you to match text in previously matched sets of parentheses. A slash followed by a digit n (\n) refers to the n^th^ parenthesized subexpression.

One example of how you can use backreferencing is searching for doubled words; for example, to find instances of 'the the' or 'is is' in text. The following example shows the syntax you use for backreferencing in regular expressions:

```
(\b[A-Za-z]+)[ ]+\1
```

This code matches text that contains a word (specified by the \b word boundary special character and the [A-Za-z]+) followed by one or more spaces [ ]+, followed by the first matched subexpression in parentheses. For example, it would match "is is, or "This is is", but not "This is".

## Exact and partial matches

Entered data is normally valid if any of it matches the regular expression pattern. Often you might want to make sure that the entire entry matches the pattern. If so, you must "anchor" it to the beginning and end of the field as follows:

- If a caret (^) is at the beginning of a pattern, the field must begin with a string that matches the pattern.
- If a dollar sign ($) is at the end of pattern, the field must end with a string that matches the pattern.
- If the expression starts with a caret and ends with a dollar sign, the field must exactly match the pattern.

## Expression examples

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

| Expression | Description |
|---|---|
| `[\?&]value=` | Any string containing a URL parameter value. |
| `^[A-Z]:(\\[A-Z0-9_]+)+$` | An uppercase DOS/Windows directory path that is not the root of a drive and has only letters, numbers, and underscores in its text. |
| `^(\+\|-)?[1-9][0-9]*$` | An integer that does not begin with a zero and has an optional sign. |
| `^(\+\|-)?[1-9][0-9]*(\.[0-9]*)?$` | A real number. |
| `^(\+\|-)?[1-9]\.[0-9]*E(\+\|-)?[0-9]+$` | A real number in engineering notation. |
| `a{2,4}` | A string containing two to four occurrences of ’a’: aa, aaa, aaaa; for example aardvark, but not automatic. |
| `(ba){2,}` | A string containing least two ’ba’ pairs; for example Ali baba, but not Ali Baba. |

### Resources

An excellent reference on regular expressions is *Mastering Regular Expressions* by Jeffrey E.F. Friedl, published by O’Reilly & Associates, Inc.

## Input Validation with JavaScript

In addition to native ColdFusion input validation using the `validate` attribute of the `cfinput` and `cftextinput` tags, the following tags support the `onvalidate` attribute, which allows you to specify a JavaScript function to handle your `cfform` input validation:

- `cfgrid`
- `cfinput`
- `cfslider`
- `cftextinput`
- `cftree`

ColdFusion passes the following JavaScript objects to the JavaScript function you specify in the `onvalidate` attribute:

- form_object (the form)
- input_object (the tag whose value is being validated)
- object_value (the value to validate)

### Handling failed validation

The `onerror` attribute allows you to specify a JavaScript function to execute if a validation fails. For example, if you use the `onvalidate` attribute to specify a JavaScript function to handle input validation, you can also use the `onerror` attribute to specify a JavaScript function to handle a failed validation (that is, when `onvalidate` returns a false value). If you are using the `validate` attribute you can also use the `onerror` attribute to specify a JavaScript function handle validation errors. The following `cfform` tags support the `onerror` attribute:

- `cfgrid`
- `cfinput`
- `cfselect`
- `cfslider`
- `cftextinput`
- `cftree`

ColdFusion passes the following JavaScript objects to the function in the `onerror` attribute:

- form_object
- input_object
- object_value
- error message text

### Example: validating an e-mail address

The following example validates an e-mail entry. If the string is invalid it displays a message box. If the address is valid it redisplays the page.

### To use JavaScript to validate form data:

1 Create a new file in ColdFusion 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 ('@', 1) == -1 ||
     Ctrl.value.indexOf ('.', 3) == -1)
  {
    return (false);
  }
  else
  {
    return (true);
  }
}
//-->
</script>

</head>

<body>
<h2>JavaScript validation test</h2>

<p>Please enter your email address:</p>
<cfform name="UpdateForm" preservedata="Yes"
  action="validjs.cfm" >

  <cfinput type="text"
    name="inputbox1"
    required="YES"
    onvalidate="testbox"
    message="Sorry, your entry is not a valid email address."
    size="15"
    maxlength="30">

<input type="Submit" value=" Update... ">
</cfform>

</body>
</html>
```

3 Save the page as `validjs.cfm`.

4 View `validjs.cfm` in your browser.

## Reviewing the code

The following table describes the highlight code and its function:

| Code | Description |
|---|---|
| `<script>`<br>`<!--`<br>`function testbox(form) {`<br>`Ctrl = Form.inputbox1;`<br>`  if (Ctrl.value == "" \|\|`<br>`    Ctrl.value.indexOf ('@', 1) == -1 \|\|`<br>`    Ctrl.value.indexOf ('.', 3) == -1)`<br>`  {`<br>`    return (false);`<br>`  }`<br>`  else`<br>`  {`<br>`    return (true);`<br>`  }`<br>`}`<br>`//-->`<br>`</script>` | JavaScript code to test for valid entry in the text box. The if statement checks to making sure that the field is not empty and contains an at sign (@) that at least the second character and a period (.) that is at least the fourth character. |
| `onvalidate="testbox"` | Calls the JavaScript testbox function to validate entries in this control. |
| `message="Sorry, your entry is not a valid email address."` | Message to display if the validation function returns a false value. |

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, or supply a URL to your own gif image.

### To create and populate a tree control from a query:

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

```
<cfquery name="engquery" datasource="CompanyInfo">
  SELECT FirstName + ' ' + LastName AS FullName
  FROM Employee
</cfquery>
<cfform name="form1" action="submit.cfm"
  method="Post">
<cftree name="tree1"
  required="Yes"
  hscroll="No">
  <cftreeitem value=fullname
    query="engquery"
    queryasroot="Yes"
    img="folder,document">
</cftree>
</cfform>
```

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

### Reviewing the code

The following table describes the highlight code and its function:

| 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. |
| `<cftreeitem value=FullName query="engquery"` | Create an item in the tree and put the results of the query named engquery in it. Because this tag uses a query, it puts one item on the tree per query entry. |
| `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 query that you display using a `cftree` control, you might want to organize your employees by the department. In this case, you separate column names with commas in the `cftreeitem` `value` attribute.

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

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

```
<!--- CFQUERY with an ORDER BY clause --->
<cfquery name="deptquery" datasource="CompanyInfo">
  SELECT Dept_ID, FirstName + ’ ’ + LastName
  AS FullName
  FROM Employee
  ORDER BY Dept_ID
</cfquery>

<!--- Build the tree control --->
<cfform name="form1" action="submit.cfm"
  method="Post">

<cftree name="tree1"
  hscroll="No"
  border="Yes"
  height="350"
  required="Yes">

<cftreeitem value="Dept_ID, FullName"
  query="deptquery"
  queryasroot="Dept_ID"
  img="cd,folder">

</cftree>
<br>
<br><input type="Submit" value="Submit">
</cfform>
```

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

### Reviewing the code

The following table describes the highlight code and its function:

| Code | Description |
|---|---|
| `ORDER BY Dept_ID` | Order the query results by department. |
| `<cftreeitem value="Dept_ID, FullName"` | Populate the tree with the Department ID, and under each department, the Full Name for each employee in the department. |
| `queryasroot="Dept_ID"` | Label the root "Dept_ID". |
| `img="cd,folder">` | Use the ColdFusion-supplied CD image for the root level and Folder image for the department IDs. The names are preceded by a bullet. |

Note that the `cftreeitem` comma-separated `img` and the `value` attributes both correspond to the tree level structure. If you leave out the `img` attribute, ColdFusion uses the folder image for all levels in the tree except the individual items, which have bullets.

## 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*

To return the root part of the path, set the `completepath` attribute of `cftree` to Yes; otherwise, the path value starts with the first node. If you specify a root name in `queryastroot`, that value gets returned as the root.

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

```
Form.tree1.node = John Allen
        Form.tree1.path = Dept_ID\3\John Allen
```

You can specify the character used to delimit each element of the path form variable in the `cftree` `delimiter` attribute. The default is a backslash.

## 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 complex of `cftree` constructs.

### Example: one-level tree control

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

```
<cfquery name="deptquery" datasource="CompanyInfo">
        SELECT Dept_ID, FirstName + ’ ’ + LastName
        AS FullName
        FROM Employee
        ORDER BY Dept_ID
        </cfquery>

<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: multilevel tree control

When populating a `cftree`, you manipulate the structure of the tree by specifying a `cftreeitem` parent. In this example, every `cftreeitem`, except the top level, specifies a parent. The `parent` attribute allows your `cftree` to show the relationships between elements in the tree control. (This example populates the tree directly, not with a query.)

```
<cfform name="form2" 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 `img` attribute, ColdFusion displays the specified image beside the tree items. You can specify a built-in ColdFusion image name or the URL of an image of your choice, such as http://localhost/Myapp/Images/Level3.gif. As a general rule, your custom images should be less than 20 pixels high.

The built-in image names are:

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

**Note**
You can also control the tree appearance by using the `lookAndFeel` attribute to specify a Windows, Motif, or Metal look.

## Embedding URLs in a cftree

The `href` attribute in the `cftreeitem` tag 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`. The URL for the link can be a relative URL or an absolute URL as in the following examples.

### To embed links in a cftree:

1 Open a new file named `tree3.cfm` in ColdFusion 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">

  <cftreeitem value="Important Links">
  <cftreeitem value="Macromedia Home"
    parent="Important Links"
    img="document"
    href="http://www.macromedia.com">
  <cftreeitem value="ColdFusion Home"
    parent="Important Links"
    img="document"
    href="http://www.coldfusion.com">
</cftree>
</cfform>
```

3 Save the page and view it in your browser.

### Reviewing the code

The following table describes the highlight code and its function:

| Code | Description |
|---|---|
| `href="http:// www.macromedia.com">` | Make the node of the tree a link. |
| `href="http:// www.cofldusion.com">` | Make the node of the tree a link.<br>Note that, although this example does not show it, `href` can refer to the name of a column in a query if the tree item is populated from that query. |

## Specifying the tree item in the URL

When a user clicks on a tree item to link to a URL the `cftreeItemKey` variable, which identifies the selected value, is appended to the URL in the form:

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

Automatically passing the name of the selected tree item as part of the URL makes it easy to implement a basic "drill down" application that displays additional information based on the selection. For example, if the specified URL is another CFML page, it can access the selected value as the variable `URL.cfteeitemkey`.

You can disable this behavior by setting the `appendkey` attribute in the `cftree` tag to No.

# Creating Data Grids with cfgrid

The `cfgrid` tag to creates a `cfform` grid control. 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
- Display images in the grid

Users can sort the grid entries in ascending order by double-clicking any column header. Double-clicking again sorts the grid in descending order. You can also add sort buttons to the grid control.

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

Just as the `cftree` tag uses `cftreeitem`, `cfgrid` uses the `cfgridcolumn` tag. You can define a wide range of row and column formatting options, as well as a column name, data type, selection options, and so on. You use the `cfgridcolumn` tag to define individual columns in the grid or associate a query column with a grid column.

The `cfgrid` tag provides many attributes that control grid behavior and appearance. This document can only cover the most important of these. For detailed information on these attributes, see the *`CFML Reference`*.

## Populating a grid from a query

### To populate a grid from a query:

1 Open a new file named `grid1.cfm` in ColdFusion Studio.

2 Edit the file so that it appears as follows:

```
<cfquery name="empdata" datasource="CompanyInfo">
  SELECT * FROM Employee
</cfquery>

<cfform name="Form1" action="submit.cfm" method="Post">

  <cfgrid name="employee_grid" query="empdata"
      selectmode="single">
    <cfgridcolumn name="Emp_ID">
    <cfgridcolumn name="LastName">
    <cfgridcolumn name="Dept_ID">
  </cfgrid>

<br><input type="Submit" value="Submit">
</cfform>
```

**Note**
Use the `cfgridcolumn` `display`="No" attribute to hide columns you want to include in the grid but not expose to an end user. You would typically use this attribute to include columns such as the table's primary key column in the results returned by `cfgrid` without exposing this data to the user.

3 Save the file and view it in your browser.

### Reviewing the code

The following table describes the highlight code and its function:

| Code | Description |
|---|---|
| `<cfgrid name="employee_grid" query="empdata"` | Create a grid named "employee_grid" and populate it with the results of the query "empdata". |
| `selectmode="single">` | Allow the user to select only one cell. Other modes are row, column and edit. |
| `<cfgridcolumn NAME="Emp_ID">` | Put the contents of the Emp_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="Dept_ID">` | Put the contents of the Dept_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 grid 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 use an updateable grid in either of two ways to make changes to your ColdFusion data sources:

- Create a page to which you pass the `cfgrid` form variables. In that page perform `cfquery` operations to update data source records base on the form values returned by `cfgrid`.
- Pass grid edits to a page that includes the `cfgridupdate` tag, which automatically extracts the form variable values and passes that data directly to the data source.

Using `cfquery` gives you complete control over interactions with your data source. The `cfgridupdate` tag provides a much simpler interface for operations that do not require the same level of control.

### Navigating and entering data in a grid

Navigating and using the `cfgrid` control is fairly straightforward, but here are a few tips:

- To sort grid rows so that a column is in ascending order by double-clicking the column header. Double-clicking again sorts the rows in descending order.
- To rearrange the columns, click any column heading and drag the column to a new position.
- When you click a cell (or row or column) that you cannot edit, its background color changes. The default color is a salmon pink.
- When you click a cell that you can edit, it is surrounded by a yellow box.
- To edit a cell, Double-click it. You must press Return when you finish entering the data.
- To delete a row, click any cell in the row and click the Delete button.
- To insert a row, click the Insert button. An empty row appears at the bottom of the grid. To enter a value in each cell, double-click the cell, enter the value, and click Return.

## Controlling cell contents

The `cfgridcolumn` `type`, `value`, `valuesDisplay,` and `valuesDelimiter` attributes let you control the data that a user can enter into a `cfgrid` cell in the following ways:

- By default, a cell is an editable text field.
- Use the `type` attribute to require numeric or string data, to make the fields check boxes, or to display an image.
- Use the `values` attribute to specify a drop-down list of values from which the user can chose. You can use the `valuesDisplay` attribute to provide a list of items to display that differs from the actual values that you enter in the database. You can use the `valuesDelimiter` attribute to specify the separator between values in the `values` `valuesDisplay` lists.
- While `cfgrid` does not have a `validate` attribute, it does have an `onvalidate` attribute that lets you specify a JavaScript function to perform validation.

For more information on controlling the cell contents, see the attribute descriptions in the *CFML Reference*.

## How user edits are returned

ColdFusion creates the following arrays as Form variables to return edits to grid rows and cells:

| Array reference | Description |
|---|---|
| `*gridname.colname*[*change_index*]` | Stores the new value of an edited cell. |
| `*gridname.*Original.*colname*[*change_index*]` | Stores the original value of the edited grid cell. |
| `*gridname.*RowStatus.Action[*change_index*]` | Stores the edit type made to the edited grid row: D for delete, I for insert, or U for update. |

When a user selects and changes data in a row, ColdFusion creates arrays to store the following information for rows that are updated, inserted, or deleted:

- The original values for all columns
- The new column values
- The type of change

For example, the following arrays are created if you an update a `cfgrid` called "mygrid" consisting of two displayable columns, (col1, col2) and one hidden column (col3).

```
Form.mygrid.col1[ change_index ]
Form.mygrid.col2[ change_index ]
Form.mygrid.col3[ change_index ]
Form.mygrid.original.col1[ change_index ]
Form.mygrid.original.col2[ change_index ]
Form.mygrid.original.col3[ change_index ]
Form.mygrid.RowStatus.Action[ change_index ]
```

The value of *change_index* increments for each row that changes, and does not indicate the specific row number. When the user updates data or inserts or deletes rows, the action page gets one array for each changed column, plus the RowStatsus.Action array. The action page does not get arrays for unchanged columns.

If the user makes a change to a single cell in col2, you can access the edit operation, the original cell value, and the edited cell value in the following arrays:

```
Form.mygrid.RowStatus.Action[1]
        Form.mygrid.col2[1]>
        Form.mygrid.original.col2[1]>
```

If the user changes the values of the cells in col1 and col3 in one row and the cell in col2 in another row, the information about the original and changed values is in the following array entries:

```
Form.mygrid.RowStatus.Action[1]><BR>
        Form.mygrid.col1[1]><BR>
        Form.mygrid.original.col1[1]>
        Form.mygrid.col3[1]><BR>
        Form.mygrid.original.col3[1]>
```

```
Form.mygrid.RowStatus.Action[2]><BR>
        Form.mygrid.col2[2]><BR>
        Form.mygrid.original.col2[2]>
```

## Editing data in cfgrid

To enable grid editing, specify the `selectmode="edit"` attribute. When enabled, a user can edit cell data and insert or delete grid rows. When the user submits a cfform containing a `cfgrid`, data about changes to grid cells gets returned in the one-dimensional arrays described in the preceding section. You can reference these arrays as you would any other ColdFusion array.

**Note**
For the sake of code brevity, the following example handles only three of the fields in the Employee table. A more realistic example would, at a minimum, include all seven of the table's fields. You might also consider hiding the contents of the Emp_ID column and automatically generating its value for new records, and displaying the Department name, from the Departmt table, in place of the Department ID.

### To make the grid editable:

1 Open the file `grid1.cfm` in ColdFusion Studio.

2 Edit the file so that it appears as follows:

```
<cfquery name="empdata" datasource="CompanyInfo">
  SELECT * FROM Employee
</cfquery>

<cfform name="GridForm"
  action="handle_grid.cfm">

  <cfgrid name="employee_grid"
    height=300
    width=250
    vspace=10
    selectmode="edit"
    query="empdata"
    insert="Yes"
    delete="Yes">

    <cfgridcolumn name="Emp_ID"
      header="Emp ID"
      width=50
      headeralign="center"
      headerbold="Yes"
      select="No">

    <cfgridcolumn name="LastName"
      header="Last Name"
      width=100
      headeralign="center"
      headerbold="Yes">

    <cfgridcolumn name="Dept_ID"
      header="Dept"
      width=35
      headeralign="center"
      headerbold="Yes">

  </cfgrid>
  <br>
  <input type="Submit" value="Submit">
</cfform>
```

3 Save the file as `grid2.cfm`.

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfgrid name="employee_grid"`<br>`  height=300`<br>`  width=250`<br>`  vspace=10`<br>`  selectmode="edit"`<br>`  query="empdata"`<br>`  insert="Yes"`<br>`  delete="Yes">` | Populate a `cfgrid` control with data from the empdata query. Selecting a grid cell enables you to edit it. Rows can be inserted and deleted. The grid is 300 X 250 pixels and has 10 pixels of space above and below it. |
| `<cfgridcolumn name="Emp_ID"`<br>`  header="Emp ID"`<br>`  width=50`<br>`  headeralign="center"`<br>`  headerbold="Yes"`<br>`  select="No">` | Create a 50-pixel wide column for the data in the Emp_ID column of the data source. Center a header named Emp ID and make it bold.<br>Do not allow users to select fields in this column for editing. Since this field is the table's primary key, users should not be able to change it for existing records and the DBMS should generate this field as an automincrement value. |
| `<cfgridcolumn name="LastName"`<br>`  header="Last Name"`<br>`  width=100`<br>`  headeralign="center"`<br>`  headerbold="Yes">` | Create a 100-pixel wide column for the data in the LastName column of the data source. Center a header named Last Name and make it bold. |
| `<cfgridcolumn name="Dept_ID"`<br>`  header="Dept"`<br>`  width=35`<br>`  headeralign="center"`<br>`  headerbold="Yes">` | Create a 35-pixel wide column for the data in the Dept_ID column of the data source. Center a header named Dept and make it bold. |

## Updating the database with cfgridupdate

The `cfgridupdate` tag provides a simple mechanism for updating the database, including inserting and deleting records. It can add, update, and delete records simultaneously. It is particularly convenient because it automatically handles collecting the `cfgrid` changes from the various form variables and generates appropriate SQL statements to update your data source.

In most cases, you should use `cfgridupdate` to update your database. However, this tag does not provide the complete SQL control that `cfquery` provides. In particular:

- It can update only a single table.
- You have no control over the order of changes. Rows are deleted first, then rows are inserted, then any changes are made to existing rows.
- Updating stops when an error occurs. It is possible that some database changes are made, but the tag does not provide any information on them.

### To update the data source with cfgridupdate

1 Open a new file in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<cfgridupdate grid="Employee_grid"
  datasource="CompanyInfo"
  tablename="Employee">
```

3 Save the file as `handle_grid.cfm`.

4 View `grid2.cfm` in your browser, make changes to the grid, and then submit them.

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfgridupdate grid="Employee_grid"` | Update the database from the Employee_grid grid. |
| `datasource="CompanyInfo"` | Update the CompanyInfo data source. |
| `tablename="Employee"` | Update the Employee table. |

## Updating the database with cfquery

You can use the `cfquery` tag to update your database from the `cfgrid` changes. This provides you with full control over how the updates are made and allows you to handle any errors that arise.

### To update the data source with cfquery:

1 Open a new file in ColdFusion 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>
    </cfoutput>

    <cfif Form.employee_grid.rowstatus.action[counter] is "D">

      <cfquery name="DeleteExistingEmployee"
        datasource="CompanyInfo">
        DELETE FROM Employee
        WHERE
     Emp_ID=#Form.employee_grid.original.Emp_ID
     [Counter]#
      </cfquery>

    <cfelseif Form.employee_grid.rowstatus.action[counter] is "U">

      <cfquery name="UpdateExistingEmployee"
        datasource="CompanyInfo">
        UPDATE Employee
        SET
          LastName='#Form.employee_grid.LastName[Counter]#',
          Dept_ID=#Form.employee_grid.Dept_ID[Counter]#
        WHERE
          Emp_ID=#Form.employee_grid.original.Emp_ID
     [Counter]#
      </cfquery>

    <cfelseif Form.employee_grid.rowstatus.action[counter] is "I">

      <cfquery name="InsertNewEmployee"
        datasource="CompanyInfo">
        INSERT into Employee
          (LastName, Dept_ID)
        VALUES ('#Form.employee_grid.LastName[Counter]#',
     #Form.employee_grid.Dept_ID[Counter]#)
      </cfquery>

    </cfif>
  </cfloop>
</cfif>

</body>
</html>
```

3 Rename your existing handle_grid.cfm file if you wish to save it, then save this file as handle_grid.cfm.

4 View `grid2.cfm` in your browser, make changes to the grid, and then submit them.

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfif isdefined ("Form.employee_grid.rowstatus.action")> <cfloop index = "Counter" from = "1" to = #arraylen(Form.employee_grid.rowstatus.action)#>` | If there is an array of edit types, then the table needs changing. Otherwise, do nothing. Loop through the remaining code once for each row to be changed. Counter is the common index into the arrays of change information for the row being changed. |
| `<cfoutput> The row action for #Counter# is: #Form.employee_grid.rowstatus.action[Counter]# <br> </cfoutput>` | Display the action code for this row: U, I, or D. |
| `<cfif Form.employee_grid.rowstatus.action[counter] is "D"> <cfquery name="DeleteExistingEmployee" datasource="CompanyInfo"> DELETE FROM Employee WHERE Emp_ID=#Form.employee_grid.original .Emp_ID[Counter]# </cfquery>` | If the action is to delete a row, generate a SQL DELETE query specifying the Emp_ID (the primary key) of the row to be deleted. |
| `<cfelseif Form.employee_grid.rowstatus.action [counter] is "U"> <cfquery name="UpdateExistingEmployee" datasource="CompanyInfo"> UPDATE Employee SET LastName= '#Form.employee_grid.LastName[Counter]#', Dept_ID= #Form.employee_grid.Dept_ID[Counter]# WHERE Emp_ID=#Form.employee_grid.original. Emp_ID[Counter]# </cfquery>` | Otherwise, if the action is to update a row, generate a SQL UPDATE query to update the LastName and Dept_ID fields for the row specified by the Emp_ID primary table key. |

| Code | Description |
|---|---|
| `<cfelseif Form.employee_grid.rowstatus.action[counter] is "I">`<br><br>`<cfquery name="InsertNewEmployee"`<br>`datasource="CompanyInfo">`<br>`INSERT into Employee`<br>`(LastName, Dept_ID)`<br>`VALUES`<br>`('#Form.employee_grid.LastName[Counter]#',`<br>`#Form.employee_grid.Dept_ID[Counter]#)`<br>`</cfquery>` | Otherwise, if the action is to insert a row, generate a SQL INSERT query to insert the Employee's last name and department ID from the grid row into the database. The Insert operation assumes that the DBMS automatically increments the Emp_ID primary key. If you use the Dbase version of the CompanyInfo database that is provided for UNIX installations, the record is inserted without an Emp_ID number. |
| `</cfif>`<br>`</cfloop>`<br>`</cfif>` | Close the `cfif` tag used to select among deleting, updating, and inserting.<br>Close the loop used for each row to be changed.<br>Close the `cfif` tag that surrounds all the active code. |

## 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, label font name, size, boldface, italics, and color, as well as slider scale increments, range, positioning, tick marks, and behavior. Slider bars are useful because they are highly visual and users cannot enter invalid values.

### To create a slider control:

1 Create a new file in ColdFusion Studio.

2 Modify the file so that it appears a follows:

```
<cfform name="Form1" action="submit.cfm"
  method="Post">

  <cfslider name="myslider"
    bgcolor="cyan"
    bold="Yes"
    range="0,1000"
    scale="100"
    value="600"
    fontsize="14"
    label="Slider %value%"
    height="60"
    tickmarkmajor="True"
    width="400">

</cfform>
```

3 Save the file as `slider.cfm` and view it in your browser.

To get the value of the slider in the action page, use the variable Form.*slider_name*; in this case, Form.myslider.

## 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.

The following example shows a basic `cftextinput` control. This example validates a date entry, which means that a user must enter a valid date in the form *mm/dd/yy* (the year can be up to four digits). For a complete list of validation formats, see the *CFML Reference*.

```
Please enter a date:<br>
<cfform name="Form1"
  action="submit.cfm"
  method="Post">

  <cftextinput name="entertext"
    value="mm/dd/yy"
    maxlength="10"
    validate="date"
    font="Trebuchet MS">
  <br>
  <br>
  <input type="Submit"
      value="Submit">

</cfform>
```

To get the value of the input text in the action page, use the variable Form.*textinput_name*; in this case, Form.entertext.

## Building Drop-Down List Boxes

The drop-down list box that you can create with `cfselect` is similar to the HTML `select` tag. However, `cfselect` gives you more control over user inputs, provides error handling, and, most importantly, allows you to automatically populate the selection list from a query.

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 that you want to display.

### To populate a drop-down list box with query data using cfselect:

1 Open a new file in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<cfquery name="getNames"
  datasource="CompanyInfo">
  SELECT * FROM Employee
</cfquery>

<cfform name="Form1" action="submit.cfm"
  method="Post">

  <cfselect name="employees"
    query="getNames"
    value="Emp_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 tag includes the `multiple` attribute, the user can select multiple entries in the list box. Also, because the `value` tag specifies Emp_ID, the primary key for the Employee table, Employee IDs (not first names) get passed in the Form.Employee variable to the application page specified in the `cfform action` attribute.

# 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 Java Applets page (under Extensions on the Server tab). In the Administrator, you define the interface to the applet, encapsulating it so that each invocation of the `cfapplet` tag is very simple.

The `cfapplet` tag offers several advantages over using the HTML `applet` tag:

- **Return values** Since `cfapplet` requires a form field `name` attribute, you can avoid coding additional JavaScript to capture the applet's return values. You can reference return values like any other ColdFusion form variable: `Form.`*`variablename`*.
- **Ease of use** Since the applet's interface is defined in the Administrator, each instance of the `cfapplet` tag in your pages only needs to reference the applet name and specify a form variable name.
- **Parameter defaults** ColdFusion uses the parameter value pairs you defined in the Administrator. You can override these values by specifying parameter value pairs in `cfapplet`.

When an applet is registered, you enter just the applet source and the form variable name:

```
<cfapplet appletsource="Calculator"
        name="calc_value">
```

By contrast, with the HTML `applet` tag, you must declare all the applet's parameters every time you want to use it in a ColdFusion page.

## Registering a Java applet

Before you can use a Java applet in your ColdFusion pages, you must register the applet in the Administrator.

### To register a Java applet:

1. Open the ColdFusion Administrator by clicking on the Administrator icon in the ColdFusion Program group and entering the Administrator password (if required).
2. Click Java Applets on Administrator Server tab to open the Java Applets page.
3. Click the Register New Applet button to open the Add/Registered Java Applet page.
4. Enter a name for the applet that you want to register. Enter the information your applet requires, and choose the height, width, vertical and horizontal space, and alignment that you want. Enter the Parameter names and their default values.
5. Click Create to complete the process.

## Applet registration fields

The following table explains the applet registration fields:

| Field | Description |
|---|---|
| Codebase | Enter the base URL of the applet: the directory that contains the applet components. The applet class files must be located within the Web browser root directory. Example:<br>http://*servername*/classes |
| Code | This is the name of the file that contains the compiled applet. The filename is relative to the code base URL. The *.class file extension is not required. |
| Method | Enter the name of a method in the applet that returns a string value. If you specify the method name in the `cfapplet` tag `name` attribute, the value returned by the method is available in the form's action page as Form.*name*. If the applet has no method, leave this field blank. |
| Height | Enter a measurement in pixels for the vertical space for the applet. |
| Width | Enter a measurement in pixels for the horizontal space for the applet. |
| 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 that 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 typically provides 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 provides guidelines on valid entries. |

## Using cfapplet to embed an applet

After you register an applet, you can use the `cfapplet` tag to place the applet in a ColdFusion page. The `cfapplet` tag has two required attributes: `appletsource` and `name`. Since you registered the applet, and you defined each applet parameter with a default value, you can invoke the applet with a very simple form of the `cfapplet` tag:

```
<cfapplet appletSource="appletname" name="form_variable">
```

### Overriding alignment and positioning values

To override any of the values defined in the Administrator for the applet, you can use the optional `cfapplet` parameters to specify custom values. For example, the following `cfapplet` tag specifies custom spacing and alignment values:

```
<cfapplet appletSource="myapplet"
        name="applet1_var"
        height=400
        width=200
        vspace=125
        hspace=125
        align="left">
```

### Overriding parameter values

You can also override the values that you assigned to applet parameters in the Administrator by providing new values for any parameter. In order to override a parameter, you must have already defined the parameter and a default value for it in the ColdFusion Administrator Applets page.

```
<cfapplet appletSource="myapplet"
        name="applet1_var"
        Param1="registered parameter1"
        Param2="registered parameter2">
```

## Handling form variables from an applet

The `cfapplet` tag requires you to specify a form variable name for the applet. This variable, referenced like other ColdFusion form variables, `Form.`*`variable_name`* holds the value the applet method returns when it is executed in the `cfform`.

Not all Java applets return values. For instance, many graphical widgets do not return a specific value; they do their flipping, spinning, fading, exploding, and that is all. For this kind of applet, the method field in the Administrator remains empty. Other applets, however, do have a method that returns a value. You can only use one method for each applet that you register. If an applet includes more than one method that you want to access, you can register the applet with a unique name for each additional method you want to use.

**To reference a Java applet return value in your application page:**

1. Specify the name of the method in the Add/Registered Java Applet page of the ColdFusion Administrator.
2. Specify the method name in the `name` attribute of the `cfapplet` tag when you code your `cfform`.

When your page executes the applet, a form variable is created with the name that you specified. If you do not specify a method, no form variable is created.

Chapter 10

# 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. These ways include the following techniques:

- If you are using ColdFusion Studio, you can write code snippets, which you can copy into pages.
- You use the `cfinclude` tag to include a ColdFusion page in another page. Included pages behave just as though you typed the included code directly into the calling page.
- You can create custom CFML tags. Unlike included pages, these custom tags act as other tags do. You pass parameters to the custom tags from the calling page and the custom tag pages have their own local Variables scope.

The following sections describe these techniques in more detail.

## Reusing Common Code with cfinclude

Often, you 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 page that calls the included page is sometimes referred to as the **calling** page. Each time the calling page is requested, the included page's file contents are added in that page for processing.

For `cfinclude` syntax, see the *CFML Reference*.

### To reference code in a calling page:

1 Create a file header.fm that displays your company's logo. Your page could consist of just the following lines or it could include many lines to define an entire header.

```
<img src="mylogo.gif">
<br>
```

(Of course, for this code to work you must also put your company's logo as a gif file in the same directory as header.cfm.)

2 Open the file askemp.cfm in ColdFusion Studio.

3 Include header.cfm in this page by adding the following line just below the `<body>` tag:

```
<cfinclude template="header.cfm">
```

4 Save the page.

5 Open getemp.cfm in ColdFusion Studio.

6 Include the header.cfm file in this page:

```
<cfinclude template="header.cfm">
```

7 View askemp.cfm in a browser, then submit the form so that you display getemp.cfm.

The header should appear on both pages.

**Note**
The file header.cfm must be in the same directory where you saved askemp.cfm and getemp.cfm (or a subdirectory). If it is not, make sure it is in a directory that has a mapping defined in ColdFusion Administrator, or move it to the appropriate directory.

## Using Custom Tags

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 reuse while offering off-the-shelf solutions to many programming chores.

You use a custom tag just as you would use a standard HTML or ColdFusion tag, for example, you might call a custom tag to generate a happy birthday message as follows:

```
<CF_HappyBirthday name="Ted Cantor" birthDate="December, 5, 1987">
```

A custom tag can also have a body and end tag, for example:

```
<CF_HappyBirthdayMessge name="Ellen Janes" birthDate="June, 8, 1993">
<P> Happy Birthday Ellen!</P>
<P> May you have many more!</P>
</CF_HappyBirthdayMessage>
```

You call custom tags by adding the `cf_` prefix to the filename (without the .cfm suffix). For example, use the tag name `cf_getweather` to call the file getweather.cfm. You must store custom tags that you call directly in either the same directory as the calling page, in the CFUSION\CustomTags directory, or in a subdirectory of the CFUSION\CustomTags directory. Each file defines a single custom tag.

You can also use the `cfmodule` tag to call custom tags. The `cfmodule` tag lets you specify the location of the custom tag file. The `cfmodule` tag is useful if you are concerned about possible name conflicts when invoking a custom tag or if the application must use a variable to dynamically call a custom tag at runtime. For more information on using the `cfmodule` tag, see the*CFML Reference*.

## Using existing custom tags

Before creating a custom tag in CFML, you should review the Custom Tag section of the ColdFusion Developer Exchange. 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 21, "Building Custom CFXAPI Tags" on page 389.

If you do not find a tag that meets your specific needs, you can create your own custom tags in CFML.

## Creating custom CFML tags

Creating a custom tag in CFML is no different from writing any CFML page. You can use all CFML constructs, as well as HTML. You are free to use any naming convention that fits your development practice. Unique descriptive names make it easy for you and others to find the right tag.

**Note**
While tag names in ColdFusion pages are case-insensitive, custom tag filenames must be lowercase on UNIX.

## Variable scopes and special variables

ColdFusion provides several variable scope types and built-in variables that help you pass information between calling pages and custom tag pages.

- Use the Attributes scope in the custom tag page to refer to the attributes passed by the calling page. For example, assume the calling page has the following line:

  ```
  cf_mytag Firstname="Thadeus" Lastname="Jones">
  ```

  In this case, the mytag.cfm custom tag page refers to the passed attributes as `Attributes.Firstname` and `Attributes.Lastname`.
- Use the Caller scope in the custom tag page to refer to local variables in the calling tag page. You use the Caller scope to return values to the calling page, but you can also use this scope to access any local variables on the calling page.
- Use the Request scope for variables in nested tags. The Request scope is available to the base page, all pages it includes, all custom tag pages it calls, and all custom tag pages called by the included pages and custom tag pages. Collaborating custom tags that are not nested in a single tag can exchange data via the request structure. The Request scope is represented as a structure named `Request`.

Custom tag pages also have access to system data structure called `thisTag`. The `thisTag` structure contains information about the tag and its execution environment. The `thisTag` variable is described in “Executing Custom Tags,” on page 185.

## Using tag attributes

Custom tag attribute values are passed from the calling page to the custom tag page as name-value pairs.

CFML custom tags support required and optional attributes. 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 with a `default` attribute at the top of a custom tag to test for and assign defaults for optional attributes that are passed from a calling page.
- Use the `cfparam` tag without a `default` attribute or a `cfif` tag with an `IsDefined` function at the top of a custom tag to test for required attributes that must be passed from a calling page.
- Use the `Attributes.`*`attribute_name`* syntax when referring to passed attributes to distinguish them from custom tag page local variables.
- Attributes are case-insensitive.
- Attributes can be listed in any order within a tag.
- Attribute = value pairs for a tag must be separated by a space.
- Passed values that contain spaces must be enclosed in double-quotes.

## Passing values to and from custom tags

Because custom tags are individual ColdFusion pages, variables and other data are not automatically shared between a custom tag and the calling page.

To pass data from the calling page to the custom tag, specify attribute name-value pairs in the custom tag, just as you do for normal HTML and CFML tags. In the custom tag you use the Attributes scope to access these variables.For example, to pass the value of the NameYouEntered variable to the cf_getMD tag, you can call the custom tag as follows:

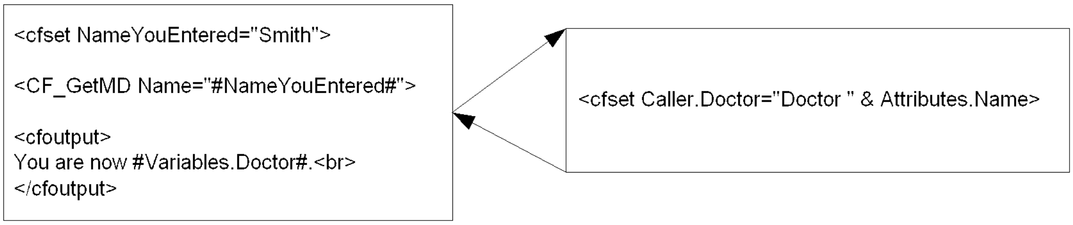
```
<cf_getMD Name="#NameYouEntered#">
```

In the getmd.cfm file, you refer to the passed attribute as `Attributes.Name`.

To pass values back to the calling page, use the Caller scope. The custom tag page can also access variables already set on the calling by simply prefixing the calling page's local variable name with Caller. For example, use the following code to set the variable Doctor on the calling page:

```
<cfset Caller.Doctor="Doctor " & Attributes.Name>
```

The following figure shows the relation between the variables on the calling page and the custom tag:



```
<cfset NameYouEntered="Smith">

<CF_GetMD Name="#NameYouEntered#">

<cfoutput>
You are now #Variables.Doctor#.<br>
</cfoutput>
```

```
<cfset Caller.Doctor="Doctor " & Attributes.Name>
```

## Passing custom tag attributes via CFML structures

You can use the reserved attribute `attributecollection` to pass attributes to custom tags. `Attributecollection` must reference a structure that contains the attribute names as the keys and the attribute values as the values. You can freely mix `attributecollection` with other attributes when you call a custom tag.

The key-value pairs in the structure specified by `attributecollection` get copied into the called page Attributes scope. This has the same effect as specifying the `attributecollection` entries as individual attributes when you call the custom tag. The custom tag page refers to the attributes passed using `attributecollection` the same way as it does other attributes; for example as Attributes.CustomerName or Attributes.Department_number.

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

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 ColdFusion generates a Template exception.

The following example uses an `attributecollection` to pass two of four attributes:

```
<cfset zort=StructNew()>
<cfset zort.x = "-X-">
<cfset zort.y = "-Y-">
<cf_testtwo a="blab" attributecollection=#zort# foo="16">
```

If testtwo.cfm contains this CFML:

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

its output is the following statement:

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

## Custom tag example

In this example, we create a custom tag that uses an attribute that is passed to it to set the value of a variable called Doctor on the calling page.

### To create a custom tag:

1 Create a new application page (the calling page) in ColdFusion 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 the Custom tag --->

<!--- display the results of the custom tag --->
<cfoutput>
You are now #Variables.Doctor#.<br>
</cfoutput>
</body>
</html>
```

3 Save the page as `callingpage.cfm`.

4 Create another new page (the custom tag) in ColdFusion Studio.

5 Enter the following code:

```
<!--- The value of the variable Attributes.Name comes from the
     calling page.
     If the calling page does not set it, make it "Who". --->
<cfparam name="Attributes.Name" default="Who">

<!--- Create a variable called Doctor, make its value "Doctor "
     followed by the value of the variable Attributes.Name.
     Make its scope Caller so it is passed back to the calling page
--->
<cfset Caller.Doctor="Doctor " & Attributes.Name>
```

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.

## Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfset NameYouEntered="Smith">` | In the calling page, create a variable NameYouEntered and assign it the value "Smith." |
| `<cfoutput>` `Before you leave this page, you’re` `#NameYouEntered#.<br>` `</cfoutput>` | In the calling page, display the value of the NameYouEntered variable before calling the custom tag. |
| `<CF_GetMD Name="#NameYouEntered#">` | In the calling page, call the GetMD custom tag and pass it the Name attribute whose value is the value of the local variable NameYouEntered. |
| `<cfparam name="Attributes.Name" default="Who">` | The custom tag page normally gets the Name variable in the Attributes scope from the calling page. Assign it the value "Who" if the calling page did not pass an attribute. |
| `<cfset Caller.Doctor="Doctor " & Attributes.Name>` | In the custom tag page, create a variable called Doctor in the Caller scope so it will exist in the calling page as a local variable. Set its value to the concatenation of the string “Doctor “ and the value of the Atributes.Name variable. |
| `<cfoutput>` `You are now #Variables.Doctor#.<br>` `</cfoutput>` | In the calling page, display the value of the Doctor variable returned by the custom tag page. (We use the Variables scope prefix to emphasize the fact that the variable is returned as a local variable.) |

**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 in the custom tag page. This includes the variables in the Form, Url, Cgi, Request, Cookies, Server, Application, Session, and Client scopes.

## Nesting Custom Tags

A custom tag can call other custom tags, thereby **nesting** tags. ColdFusion uses nested tags such as `cfgraph` and `cfgraphdata`, `cfhttp` and `cfhttppam`, and `cftree` and `cftreeitem`. The ability to nest tags allows you to provide similar functionality.

The calling tag is known as an **ancestor**, **parent**, or **base** tag, while the tags that ancestor tags call are known as **descendant**, **child**, or **sub** tags. Together, the ancestor and all descendent tags are called **collaborating** tags.

The following table lists the terms that describe the relationships between nested tags:

| Calling tag | Tag nested within the calling tag | Description |
|---|---|---|
| ancestor | descendant | An ancestor is any tag that contains other tags between its start and end tags. A descendant is any tag called by a tag. |
| parent | child | Parent and child are synonyms for ancestor and descendant. |
| base tag | sub tag | A base tag is an ancestor that you explicitly associate with a descendant, called a sub tag, with `cfassociate`. |

In order to nest tags, the parent tag must have a closing tag.

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

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

- The **start** mode, in which the base tag is processed for the first time.
- The **inactive** mode, in which sub tags and other code contained within the base tag are processed. No processing occurs in the base tag during this phase.
- The **end** mode, in which the base tag is processed a second time. The end mode occurs when ColdFusion reaches the custom tag's end tag.

### 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. 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 while encapsulating each tag's implementation so that others cannot see it.

When you decide to you use nested tags, you must address the following issues:

- What data should be accessible?
- Which tags can communicate to which tags?
- How are the source and targets of the data exchange identified?
- What CFML mechanism is used for the data exchange?

## What data is accessible?

To enable developers to obtain maximum productivity in an environment with few restrictions, CFML custom tags can expose all their data to collaborating tags.

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

To preserve encapsulation, put all tag data access and modification operations into custom tags. For example, rather than 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, the descendent tags have no instance data. A tag, therefore, can only access data from its ancestors, not its descendents. Ancestor data is 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 up the hierarchy of nested tags, including those in included pages and custom tag references, to the start of the base page for the request. Both `cfinclude` tags and custom tags appear on the tag context stack.

## High-level data exchange

There are many cases in which descendant tags are used only as a means for data validation and exchange with an ancestor tag, such as `cfhttp/cfhttpparam` and `cftree/cftreeitem`. You can use the `cfassociate` tag to encapsulate this processing.

The cfassociate tag has the following format:

```
<cfassociate baseTag="tagName" dataCollection="collectionName"
```

The baseTag attribute is the name of the base tag that gets access to this tag's attributes. The dataCollection attribute is the name of the structure in which the base tag stores the sub-tag data. Its default value is AssocAttribs. You only need to specify a dataCollection attribute if the base tag can have more than one type of sub tag. It is convenient for keeping separate collections of attributes, one per tag type.

When cfassociate is encountered in a sub tag, the sub tag's attributes are automatically saved in the base tag. The attributes are in a structure appended to the end of an array whose name is 'thisTag.*collectionName*'.

The cfassociate tag performs the following operations:

```
<!--- Get base tag instance data --->
        <cfset data = getBaseTagData(baseTag).thisTag>
        <!--- Create a string with the attribute collection name --->
        <cfset collection_Name = "data.#dataCollection#">
        <!--- Create the attribute collection, if necessary --->
        <cfif not isDefined(collectionName)>
        <cfset #collection_Name# = arrayNew(1)>
        </cfif>
        <!--- Append the current attributes to the array --->
        <cfset temp=arrayAppend(evaluate(collectionName), attributes)>
```

The 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 functions provide access to ancestral data:

- GetBaseTagList() Returns a comma-delimited list of uppercase 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 (not just the local variables) of the nth ancestor with a given

name. By default, the closest ancestor is returned. If there is no ancestor by the given name or if the ancestor does not expose any data (such as CFIF), an exception is thrown.

## Example: Ancestor data access

This example creates two custom tags and a simple page that calls each of the custom tags. The first custom tag calls the second. The second tag reports on its status and provides information about its ancestors.

### To create the calling page:

1 Create a new application page (the calling page) in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
Call cf_nestag1 which calls cf_nestag2<br>
<cf_nestag1>
<hr>

Call cf_nestag2 directly<br>
<cf_nestag2>
<hr>

Using a loop to call call cf_nestag2<br>
<cfloop index=i from=1 to=2>
<cf_nestag2>
</cfloop>
```

3 Save the page as nesttest.cfm.

### To create the first custom tag page:

1 Create a new application page (the calling page) in ColdFusion Studio.

2 Put the following single line in the file:

```
<cf_nestag2>
```

3 Save the page as nestag1.cfm.

### To create the second custom tag page:

1 Create a new application page (the calling page) in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<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. You are the second entry because
     the first entry in the context stack is the cfif tag at
     the top of this file --->
  <cfoutput>
  <p>I’m custom tag #ListGetAt(ancestorlist,2)#</p>
  <!--- output all the contents of the stack a line at a time --->
  <cfloop index="loopcount" from="1" to=#listlen(ancestorlist)#>
  Ancestorlist entry #loopcount# n is
     #ListGetAt(ancestorlist,loopcount)#<br>
  </cfloop><br>
  </cfoutput>

  <!--- Determine whether you'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'>
      custom tag code either because it is in
      its start or end execution mode.
    <cfelse>
      body of the tag
    </cfif>
    <p>
  <cfelse>
    <!--- Say you are lonely --->
    I'm not nested inside any custom tags. :^( <p>
  </cfif>
</cfif>
```

3 Save the page as nestag2.cfm.

4 Open the file nesttest.cfm in your browser.

# Executing Custom Tags

The following sections provide information on executing custom tags.

## Tag instance data

When a custom tag page executes, ColdFusion keeps data related to the tag instance. The `thisTag` built in structured variable preserves this data with a unique identifier. The behavior is similar to the `File` tag-specific variable (sometimes called the File scope).

The following variables are generated by the `thisTag` structure:

| Variable | Description |
|---|---|
| `ExecutionMode` | Valid values are "start" and "end." |
| `HasEndTag` | Used for code validation. It distinguishes between custom tags that are called with and without end tags. |
| `GeneratedContent` | The content that has been generated by the tag. This includes anything in the body of the tag, including the results of any active content, such as ColdFusion variables and functions. You can process this content as a variable. |
| `AssocAttribs` | Holds the attributes of all nested tags if you use `cfassociate` to make them available to the parent tags. |

## Modes of execution

ColdFusion invokes a custom tag page 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 page gets 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.

The same CFML page is executed for both the start and end tag of a custom tag.

## Specifying execution modes

A variable with the reserved name `thisTag.ExecutionMode` will specify the mode of invocation of a custom tag page. The variable has one of the following values:

- Start   start tag execution
- End   end tag execution

When the body of the custom tag (not the custom tag page or template) executes, the value of the ExecutionMode variable is *inactive*.

A custom tag page that performs processing in both modes can have the following format:

```
<cfif thisTag.ExecutionMode is 'start'>
        <!--- Start tag processing --->
        <cfelse>
        <!--- End tag processing --->
        </CFIF>
```

You can also use `cfswitch`:

```
<cfswitch expression=#thisTag.ExecutionMode#>
        <cfcase value= 'start'>
        <!--- Start tag processing --->
        </cfcase>
        <cfcase value='end'>
        <!--- End tag processing --->
        </cfcase>
        </cfswitch>
```

## Terminating tag execution

The `cfexit` tag terminates execution of a custom tag. The `cfexit` tag'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.

You can also use the `method` attribute to specify that the tag body executes again. This enables custom tags to act as high-level iterators, emulating `cfloop` behavior.

The following table summarizes `cfexit` behavior:

| method attribute value | Location of cfexit call | Behavior |
|---|---|---|
| ExitTag (default) | Base page | Acts like cfabort |
| | ExecutionMode=start | Continue after end tag |
| | ExecutionMode=end | Continue after end tag |
| ExitTemplate | Base page | Acts like cfabort |
| | ExecutionMode=start | Continue from first child in body |
| | ExecutionMode=end | Continue after end tag |
| Loop | Base page | Error |
| | ExecutionMode=start | Error |
| | ExecutionMode=end | Continue from first child in body |

## 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 results of processing the body of a given tag. This includes all text and HTML code in the body, the results of evaluating ColdFusion variables, expressions, and functions, and the results generated by descendant tags. Any changes to the value of this variable results 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 page. Because ColdFusion loads the first instance it finds of the custom tag called by a page, 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 page. 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 subfolders 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 multideveloper environment or distribute your tags publicly, you can use additional ColdFusion capabilities:

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

## Resolving filename conflicts

To avoid errors caused by duplicate custom tag filenames, use the `cfmodule` tag in the calling page. You must use either a `template` or `name` attribute in the tag, but you cannot use both The following table lists the basic `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.<br>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 subdirectory under the Custom Tags root directory.<br>Example: `<cfmodule name="MyApp.GetUserOptions">` identifies the file `GetUserOptions.cfm` in the `Custom Tags\MyApp` directory under the ColdFusion root directory. |
| attributes | 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. For details, see *Advanced ColdFusion Administration*.

## Encoding custom tags

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

The `cfencode` tag uses the following syntax:

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

The following table describes the options:

| Option | Description |
|---|---|
| input file | Name of the file you want to encode. `cfencode` does not process an encoded file. |
| output file | Path and filename of the output file.<br>**Warning**: If you do not specify an output filename, 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 11

# Preventing and Handling Errors

ColdFusion includes many tools and techniques for ensuring that your code works properly. These tools include sophisticated debugging and code validation tools, error logging tools, and error handling mechanisms. This chapter describes these tools and presents approaches to troubleshooting common problems.

ColdFusion Studio also provides interfaces for debugging application pages and for dynamically validating multiple levels of HTML and CFML code. This chapter does not discuss debugging in ColdFusion Studio.

## Contents

## Debug Settings in the ColdFusion Administrator

ColdFusion can provide important debugging information for every application page requested by a browser. When you enable debugging, the output displays in a block following normal page output.

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

**Note**
By default, when you enable any of the debugging and logging options, debug output becomes visible to all users. To restrict debug output to specific IP addresses, use the Debugging IPs page of the ColdFusion Administrator to specify the addresses that can receive debugging messages.

### 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
```

**Note**
If you do not restrict access to debugging information, any browser can use this parameter to get debugging information. To restrict access to specific IP addresses, use the Debugging IPs page of the ColdFusion Administrator to specify the addresses that can receive debugging messages.

### 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 displays a diagnostic message in the user's browser. The message includes a link that allows the user to e-mail a report of the error to the site administrator. You enable the mail link feature in the Mail Logging page of the ColdFusion Administrator. Errors are written to a log file for later review.

ColdFusion returns the following information:

- Database errors, including the ODBC error code, the extended error message returned from 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

You must select the Display the template path in error messages option on the ColdFusion Administrator Debug Options page to include the path of the page that encounters the problem in each error message.

**Tip**

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

For information on using the Logging settings and Mail Logging settings, see *Advanced ColdFusion Administration*

## CFML Code Validation

ColdFusion provides two methods of validating your CFML code:

- Runtime validation
- The CFML Syntax Checker

### Runtime validation

The ColdFusion Application Server features two modes of attribute checking for processing application pages: strict and relaxed. Macromedia recommends that you always use the strictest possible level of CFML validation.

To enable strict validation:

1 Open the ColdFusion Administrator Server Settings page.

2 Select the Enable Strict Attribute Validation check box.

The code validator inspects all code and validates most attributes before execution begins, when the CFML is converted into executable pseudocode. However, tags with a switch attribute, such as `action` or `method`, for which the value is provided at runtime, are validated during execution. Validation of such attributes causes a slight performance penalty.

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 tag 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

You can run the CFML syntax checker to validate your pages. It scans your pages and returns a list of pages that do not pass the syntax check, with error messages indicating the cause of each failure. The CFML Syntax checker application page is:

*`webroot`*`/cfdocs/cfmlsyntaxcheck/cfmlsyntaxcheck.cfm.`

**Note**
The CFML syntax checker cannot detect an invalid attribute combination for attributes whose values are provided at runtime because it does not execute the CFML page it checks.

## Troubleshooting Common Problems

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

### ODBC data source configuration

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

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

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 might need to consult your database administrator to help 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` attribute 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 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 do not understand.

Make sure all your CFML tags have matching end tags where appropriate. It is a common error to omit the end tag for the `cfquery`, `cfoutput`, `cftable`, or `cfif` tag.

When developing pages in ColdFusion Studio, use the Tag Completion feature, which adds an closing 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.

**Problem**: You suspect that there are problems with the structure or contents of a complex data variable, such as a structure, array, query object, or WDDX-encoded variable.

Use the `cfdump` tag to generate a table-formatted display of the variable's structure and contents. For example, to dump a structure named `relatives`, use the following line. Note that you must surround the variable name with pound signs.

```
<cfdump var=#relatives#>
```

# Error Handling in ColdFusion

By default, ColdFusion generates its own error messages when it encounters errors. In addition, it provides a variety of tools and techniques for you to customize error information and handle errors when they occur, including the following techniques:

- You can specify custom pages for ColdFusion to display when a ColdFusion page is missing or if it encounters an exception error during the processing of a page (the Site-wide Error Handling page). You specify these pages on the ColdFusion Administrator Server Settings page.
- You can use the `cferror` tag to specify ColdFusion pages to handle specific types of errors.
- You can use the `cftry`, `cfcatch`, and `cfthrow` tags to catch and handle exception errors directly.

The remaining sections in this chapter provide the following information:

- The basic building blocks for understating types of ColdFusion errors and how ColdFusion handles them
- How to use the `cferror` tag to specify error-handling pages
- Logging errors
- How to handle ColdFusion exceptions

## Understanding ColdFusion errors

Before you can effectively managee ColdFusion errors, you must understand the error types and how ColdFusion handles them.

### ColdFusion error types

ColdFusion errors can help you to debug your application and provide feedback to users. There are several types of errors in ColdFusion:

- **Missing template** These errors occur if ColdFusion Server gets a request for a page that it cannot find.
- **Validation** These errors occur when a user violates the server-side form field validation rules in a form being submitted. You specify server-side form validation by using hidden form fields. Data validation errors should not be confused with code validation.
- **Exception** Exceptions are events that disrupt an application’s normal flow of instructions. There are three general types of exceptions:
  - Error responses from external services, such as an ODBC driver or CORBA server
  - CFML errors or the results of `cfthrow` or `cfabort` tags
  - Internal errors in the ColdFusion Server

## How ColdFusion handles errors

The following pseudo-code program illustrates how ColdFusion handles errors. The rest of this expands on this outline:

```
Case missing template error
  ColdFusion displays either a standard error page or an error page
        that you specify as the Missing Template Handler in the
        Administrator
        Server Settings Missing Template Handler field.

Case validation error
  If cferror in application.cfm specifies a validation error handler
    Use error page specified by cferror
  Else
    Use standard CFML validation error message format
        Endif

Case exception error
  If a cferror tag in aplication.cfm specifies a monitor error handler
        for the exception type
    Use the Monitor error page; when the page exits continue handling
        the error as follows
  Endif
  If code with error is inside a cftry tag and the exception type is
         specified in a cfcatch tag (see Note)
    Execute code in cfcatch tag
  Else If a cferror tag specifies an exception error handler for the
        exception type
    Use error page specified by cferror
  Else If Administrator Settings Site-wide Error Handler field
        specifies an error handler page
    Use custom error page
  Else If a cferror tag specifies an request error handler
    Use error page specified by cferror
  Else
      Use standard CFML error message
        Endif
```

**Note**
If the current tag is nested inside other tags, the CFML processor checks the entire stack of open tags until it finds a suitable `cftry/cfcatch` combination or reaches the end of the stack.

## Generating Custom Error Messages with cferror

By default, ColdFusion uses a standard page for most errors. Custom error pages allow you to control the error information that users see. You can specify custom error pages for different types of errors and handle different types of errors in different ways. For example, you can create specific pages to handle errors that could be recoverable, such as request timeouts. You can also make your error messages consistent with the look and feel of your application.

You can specify the following types of custom error pages:

| Type | Description |
|---|---|
| Validation | Handles server-side form field data validation errors. The validation error page cannot include CFML tags, but can display special error page variables. |
| Exception | Handles exception errors. You can specify different error pages for different types of exceptions. |
| Request | Handles any exception that is not otherwise handled. The request error page runs after the CFML language processor finishes. As a result, the request error page cannot include CFML tags, but can display error page variables. A request error page is useful as a backup if errors occur in other error handlers. |
| Monitor | Handles exceptions before any other error-handling mechanism runs. You can specify different monitor pages for different types of exceptions.<br>When the monitor page completes, error handling continues with any `cftry/cfcatch` code or other `cferror` error handlers for the error type.<br>You should only use monitor handlers for debugging, for example, to log error information, and not include them in production code. |

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. For more information, see "Understanding the Web Application Framework" on page 214.

For detailed information on the `cferror` tag, see the *CFML Reference*.

## Creating an error application page

Error application pages for validation and request errors cannot use ColdFusion tags; they can only use HTML tags. Error application pages for exception and monitor errors can use all of CFML, including tags, expressions, and functions.

Even validation and request error pages have access to specific CFML error variables such as Error.Diagnostics (for request errors) and Error.InvalidFields (for validation errors). All CFML error variables start with the prefix Error. To include these variables in your HTML, surround the variable names with pound signs, but do *not* use the `cfoutput` tag; for example:

```
<p>
ColdFusion found the following errors in the data you entered:
</p>
#error.InvalidFields#<br>
```

### Error page variables

The following variables are available on error pages:

| Error type | Error variable | Description |
|---|---|---|
| Exception<br>Request<br>Monitor | `error.type` | The exception type. For a list of error types, see “Types of recoverable exceptions supported” on page 205. |
| | `error.diagnostics` | Detailed error diagnostics from ColdFusion Server. |
| | `error.mailTo` | E-mail address of administrator who should be notified (corresponds to the value set in the `mailTo` attribute of `cferror`). |
| | `error.dateTime` | Date and time when the error occurred. |
| | `error.browser` | Browser that was running when the error occurred. |
| | `error.generatedContent` | The failed request’s generated content. |
| | `error.remoteAddress` | IP address of the remote client. |
| | `error.HTTPReferer` | Page from which the client accessed the link to the page where the error occurred. |
| | `error.template` | Page being executed when the error occurred. |
| | `error.queryString` | URL query string of the client's request. |

| Error type | Error variable | Description |
|---|---|---|
| Validation | `error.validationHeader` | Text for header of default validation message. |
| | `error.invalidFields` | Unordered list of validation errors that occurred. This includes any text that you specify in the value attribute or a hidden tag used to validate form input. |
| | `error.validationFooter` | Text for footer of default validation message. |

## Example of a request error page

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

```
<html>
<head>
  <title>Products - Error</title>
</head>
<body>

<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>
  Date and time: #error.DateTime# <br>
  Page: #error.template# <br>
  Remote Address: #error.remoteAddress# <br>
  HTTP Referer: #error.HTTPReferer#<br>
  <br>
  Diagnostics:<br>
  #error.diagnostics#
</td></tr></table>

</cfoutput>
</body>
</html>
```

## Example of a validation error page

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 correctly complete all the fields
in the form. The following problems occurred:</p>

#error.invalidFields#

</body>
</html>
```

## Logging Errors

ColdFusion Server provides extensive capabilities for generating, managing, and viewing log files, as described in *Advanced ColdFusion Administration*.

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

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

Otherwise you must use the `cflog` tag in your error handling code to generate log entries.

The `cflog` tag lets you specify the following information:

- A custom file or standard ColdFusion log file in which to write the message.
- Text to write to the log file.
- Message severity (type): Information Warning, Fatal, or Error.
- Whether to log any of the following: application name, thread ID, system date, or system time. By default, all get logged.

For example, you could use a `cflog` tag in an exception error page to log the error information to an application-specific log file.

```
<html>
<head>
  <title>Products - Error</title>
</head>
<body>

<h2>Sorry</h2>

<p>An error occurred when you requested this page.
The error has been logged and we will work to correct the problem.
```

```
We apologize for the inconvenience. </p>

<cflog file="myapp_errors"
     text="Exception error --
       Exception type: #error.type#
       Template: #error.template#,
       Remote Address: #error.remoteAddress#,
       HTTP Rerference: #error.HTTPReferer#
       Diagnositcs: #error.diagnostics#"
     type="Error">

</body>
</html>
```

## Reviewing the code

The following table describes the highlighted code and its function:

| Code | Description |
|---|---|
| `<cflog file="myapp_errors"`<br>`  text="Exception error`<br>`   Exception type: #error.type#`<br>`   Template: #Error.Template#,`<br>`   Remote Address:`<br>`#Error.RemoteAddress#,`<br>`   Diagnositcs: #Error.Diagnostics#"`<br>`  type="Error">` | When this page is processed, log an error message to the file myapp_errors.log file in the ColdFusion log directory containing the thread ID, date and time, application, and an error message that includes the exception type, the path of the page that caused the error, the remote address that called the page, and the error's diagnostic message. |

## Handling Exceptions in ColdFusion

Ordinarily, when ColdFusion encounters an error, it stops processing and displays an error message or error page (as specified by the `cferror` tag). 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 is a good idea to enclose an entire application page in a `cftry` block. You then follow the `cftry` block with `cfcatch` blocks, which respond to potential errors. When an exception occurs within the `cftry` block, processing is thrown to the `cfcatch` block for that type of exception.

**Note**
For cases when the error handler is not able to successfully handle the thrown error, use the `cfrethrow` tag within a `cfcatch` block.

Here is an outline for using `cftry` and `cfcatch` to handle errors:

```
<cftry>
        ... Put your page's application code here ...
        <cfcatch type="exception type1">
        ... Add exception processing code here ...
        </cfcatch>
        <cfcatch type="exception type2">
        ... Add exception processing code here ...
        </cfcatch>
        <cfcatch type="Any">
        ... Add exception processing code appropriate for all other
        exceptions here ...
        </cfcatch>
        </cftry>
```

To catch errors in a single problematic SQL statement, for example, you might narrow the focus by surrounding the ColdFusion tag that contains the SQL statement with a `cftry` block and following it with a `cfcatch type="Database"` tag that displays the information in the `cfcatch.SQLState` variable.

**Note**
Do not attempt to enclose an entire application in a `cftry` block by putting the `cftry` tag in Application.cfm because you cannot be sure that there always will be a matching `cftry` end tag.

For information on the `cftry`, `cfcatch`, `cfrethrow`, and `cfthrow` tags, see the *CFML Reference*.

## 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. You can also use these types as the values of the `exception` attribute in `cferror type=exception` tags.

You can specify the following basic exception types. For a list of advanced exception types, see the *CFML Reference*.

| Type | Tag(s) | Notes |
|---|---|---|
| Database failures | `cfcatch type="Database"` | Catch failed database operations, such as failed SQL statements, ODBC problems, and so on. |
| Template errors | `cfcatch type="Template"` | Catch general application page errors.<br>The `cfinclude`, `cfmodule`, and `cferror` tags can generate a template exception. |
| Missing included file errors | `cfcatch type="missingInclude"` | Catch errors where included files are missing. |
| 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 type="Lock"` | Catch failed locking operations, such as when a `cflock` critical section times out or fails at runtime. |
| Application-defined exception events raised by `cfthrow` | `cfcatch type="Application"`<br><br>`a cfcatch block that has no type attribute` | Catch only those custom exceptions that are specified as having the Application type in the `cfthrow` tag that defines them. |
| Custom exceptions raised by `cfthrow` | `cfcatch type="`*`custom_exception_type`*`"` | Catch a custom a exception type raised by `cfthrow`. |
| Unspecified exceptions | `cfcatch type="Any"` | Catch any exceptions that are not specifically handled in another error handler, including unexpected internal errors. |

Specifying the type as any causes the ColdFusion Application Server to catch internal exceptions, memory allocation errors, and access violations, which you might 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 in 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. The following variables are available in most cfcatch blocks:

| 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. |
| `cfcatch.errorCode` | Any exception that is a part of the CFML exception hierarchy supplies a value for this variable.<br>For `type="Application"`, `cfthrow` tags can 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 pathnames of the files that contain the tags in the tag stack. |

### Tag context information

On the Debugging Settings page in the ColdFusion Administrator you can select the Enable CFML stack trace option. When you enable this option, `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**
Clear the Enable CFML stack trace option to avoid having production servers expend resources creating a traceback stack by default. When you turn off this setting, `cfcatch`.tag`Context` is a zero-length array.

## Database exceptions

The following additional variables are available whenever the exception type is database:

| Property variable | Description |
|---|---|
| `cfcatch.nativeErrorCode` | The native error code associated with this exception. Database drivers typically provide error codes to assist in the diagnosis of failing database operations. The values assumed by cfcatch.NativeErrorCode are driver-dependent.<br>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.<br>If no SQLState value was provided, the value of SQLState is -1. |

## Expression exceptions

The following variable is only available for expression exceptions:

| Property variable | Description |
|---|---|
| `cfcatch.errnumber` | An internal expression error number, valid only when type="Expression". |

### Locking exceptions

The following additional information is available for exceptions related to `cflock` sections:

| Property variable | Description |
|---|---|
| `cfcatch.lockName` | The name of the affected lock. This is set to "anonymous" if the lock name is unknown. |
| `cfcatch.lockOperation` | The operation that failed. This is set to "unknown" if the failed operation is unknown. |

### Missing include exceptions

The following additional variable is available if the error is caused by a missing file specified by a `cfinclude` tag:

| Property variable | Description |
|---|---|
| `cfcatch.missingFileName` | The name of the missing file. |

## 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. In particularly problematic cases, you might enclose an exception-prone tag 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 the CompanyInfo data source used in many of the examples in this book and a sample included file, `includeme.cfm`.

If an exception occurs during the `cfquery` statement's execution, the application page flow switches to the `cfcatch type="Database"` exception handler. It then resumes with the next statement after the `cftry` block, once the `cfcatch type="Database"` handler completes.

Similarly, the `cfcatch type="MissingInclude"` block handles exceptions raised by the `cfinclude` tag. Any unknown, but possibly recoverable, exceptions are handled by the `cfcatch type="Any"` block.

```
<!--- Wrap code you want to check in a cftry block --->
<cfset EmpID=3>
<cftry>
  <cfquery name="test" datasource="CompanyInfo">
    SELECT Dept_ID, FirstName, LastName
    FROM Employee
    WHERE Emp_ID=#EmpID#
  </cfquery>

  <html>
  <head>
    <title>Test cftry/cfcatch</title>
  </head>

  <body>
  <hr>
  <cfinclude template="includeme.cfm">
  <cfoutput query="test">
  <p>Department: #Dept_ID#<br>
  Last Name: #LastName#<br>
  First Name: #FirstName#</p>
  </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">
  <cfoutput>
    <h1>Other Error: #cfcatch.Type#</h1>
    <ul>
      <li><b>Message:</b> #cfcatch.message#
      <li><b>Detail:</b> #cfcatch.Detail#
    </ul>
    </cfoutput>
  </cfcatch>
</cftry>
  </body>
  </html>
```

### To test the code:

1 Make sure there is no includeme.cfm file and display the page. The `cfcatch type="MissingInclude"` block displays the error.

2 Create a non-empty includeme.cfm file and display the page. If your database is configured properly you should see an employee entry and not get any error.

3 In the `cfquery` tag change the line:

```
FROM Employee
```

to:

```
FROM Employer
```

Display the page. This time the `cfcatch type="Database"` block displays an error message.

4 Correct Employer back to Employee. Change the `cfoutput` line:

```
<p>Department: #Dept_ID#<br>
```

to:

```
<p>Department: #DepartmentID#<br>
```

Display the page. This time the `cfcatch type="Any"` block displays an error message indicating an expression error.

## 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="Invalid_field.codeValue"
  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 use this syntax:

```
<cfcatch type="Invalid_field.codeValue">
```

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

```
<cfthrow type="MyApp.BusinessRuleException.InvalidAccount">
```

would be handled by any of the following `cfcatch` error handlers.

```
<cfcatch type="MyApp.BusinessRuleException.InvalidAccount">
<cfcatch type="MyApp.BusinessRuleException">
<cfcatch type="MyApp">
```

The handler that matches most exactly handles the error. Therefore, in this case, the `MyApp.BusinessRuleException.InvalidAccount` handler gets invoked. However, if you used the following `cfthrow` tag:

```
<cfthrow type="MyApp.BusinessRuleException.InvalidVendorCode
```

the `MyApp.BusinessRuleException` handler receives the error.

The type comparison is case-insensitive. To match types exactly, rather than performing pattern matching, use the `cfsetting` attribute `catchExceptionsByPattern="No"`.

Chapter 12

# Using the Application Framework

The ColdFusion Web Application Framework is a powerful tool that you can use to help structure your ColdFusion applications. This chapter 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.

The ColdFusion Web Application Framework is based on four basic components:

- Application-level settings and functions
- Client, Session, Application, and Server scope variables
- Custom error handling
- Web server security integration

With these components, you can easily combine your ColdFusion application pages to create sophisticated Web applications.

### Application-level settings and functions

ColdFusion provides application-level facilities that help you control settings, variables, and features available across the entire application. After you define an application, you can use the application-level features in addition to all of the other features in ColdFusion.

You specify application-level settings in the Application.cfm and OnRequestEnd.cfm files. Application.cfm is executed when ColdFusion starts processing each page in your application, and OnRequestEnd.cfm is processed after all other processing is completed for the page.

### Client, Session, Application, and Server scope variables

ColdFusion provides four variable scopes that let you to maintain data that must last beyond the scope of the current page.

| Variable Scope | Description |
|---|---|
| Client | Contains variables that are available for a single client browser over multiple browser sessions in an application. Client variables are stored as cookies, database entries, or Registry values. Client variables can time out after an extended period.<br>You cannot access the Client scope as a data structure. You should use the Client scope prefix in the variable name, but it is not required. |
| Session | Contains variables that are available for a single client browser for a single browser session in an application. Session variables are stored in memory and time out after a period of inactivity or when the server shuts down.<br>You can access the Session scope as a data structure. You must use the Session scope prefix in the variable name. |

| Variable Scope | Description |
|---|---|
| Application | Contains variables that are available to all pages in an application for all clients. Application variables are stored in memory and time out after a period of inactivity or when the server shuts down.<br>You can access the Application scope as a data structure. You must use the Application scope prefix in the variable name. |
| Server | Contains variables that are available to all applications in a server and all clients. Server variables are stored in memory. They do not time out, and are only deleted when the server stops running.<br>You cannot access the Server scope as a data structure. You must use the Server scope prefix in the variable name. |

ColdFusion does not attempt to automatically evaluate Application, Session, or Server variables. You must use variable prefixes with these variables, as in Session.variablename or Application.variablename. As a general rule you should prefix all these variables with their scope identifier.

ColdFusion provides locking functions to manage access to Session, Application, and Server variables. Because these variables are kept in your server's memory, you *must* lock them when you use them to prevent errors that arise from simultaneous access.

**Caution**
Understanding lock management and using locks effectively is vital to correctly using Session, Application, and Server scope variables. For more information on locking, see "Locking Code with cflock" on page 233.

## Custom error handling

You can use the `cferror` tag to display customized HTML pages when errors occur in your application. 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.

For more information, see "Generating Custom Error Messages with cferror" on page 199.

## 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.

For more information, see Chapter 19, "Application Security" on page 355.

## Mapping an Application Framework

An important step in designing a ColdFusion application is mapping its directory structure.

Before you start building the application, establish a root directory for the application. You can store application pages in subdirectories of the root directory.

## Processing Application.cfm and OnRequestEnd.cfm

ColdFusion uses similar but different rules to locate and process Application.cfm and OnRequestEnd.

### Processing Application.cfm

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 is in the current directory tree, ColdFusion uses the first one it finds.

Only one Application.cfm file is ever processed for each ColdFusion application page. The Application.cfm file is automatically included in each application page, as if by a `cfinclude`tag. If the Application.cfm page 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.

If an application page has a `cfinclude` tag pointing to an additional application page, ColdFusion does not initiate another search for an Application.cfm page when it includes the additional page.

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

### Processing OnRequestEnd.cfm

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 is never executed if it resides in another directory.*

The `OnRequestEnd.cfm` file is not 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 on UNIX systems, you must spell the `OnRequestEnd.cfm` file with capital O, R, and E.

## Defining the directory structure

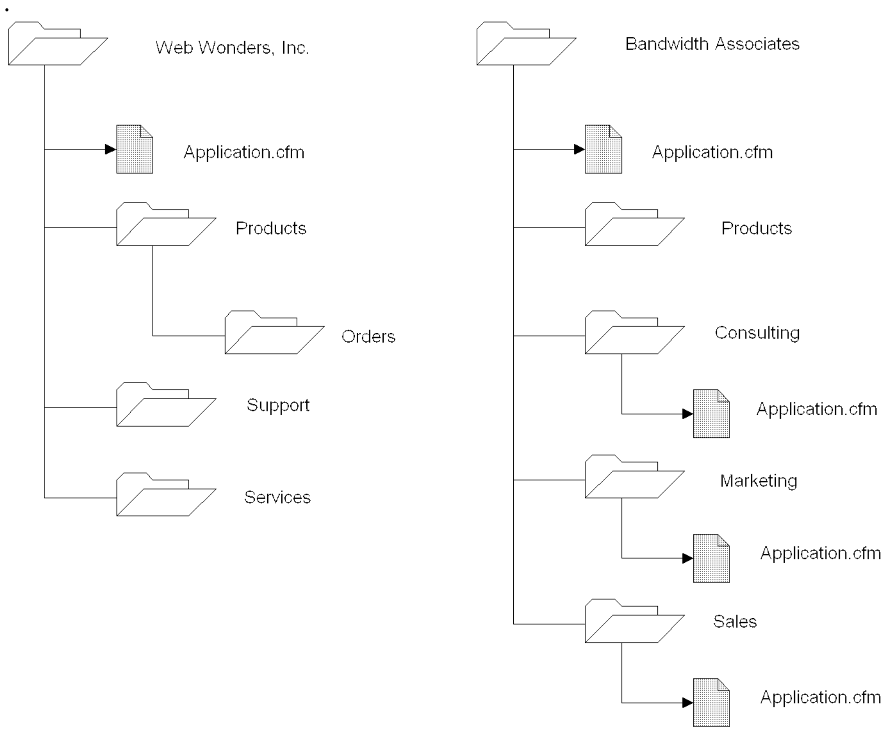
Defining a root directory for an application has a number of advantages:

- **Development** The application is easier to develop and maintain because the application page files are well organized.
- **Portability** You can more easily move the application to another server or another part of a server without changing any code in the application page files.
- **Application-level settings** Application pages that 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 following directory trees 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 example 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 have their own Application.cfm file.


Web Wonders, Inc.
Application.cfm
Products
Orders
Support
Services
Bandwidth Associates
Application.cfm
Products
Consulting
Application.cfm
Marketing
Application.cfm
Sales
Application.cfm

# Creating the Application.cfm File

The special application-wide page called Application.cfm defines application-level settings and functions such as:

- Application name
- Client state management options
- Application and Session variable management options
- Default variables
- Application-specific custom error pages
- Data sources
- Default style settings
- Exclusive locks
- Other application-level constants

## 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 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 ColdFusion 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 application default variables and constants

It is often useful to set default variables and application-level constants in the Application.cfm file. For example, you can designate:

- A data source that you are 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, stored in
        a data source called mycompany --->
<cfapplication name="Products"
  clientmanagement="Yes"
  clientstorage="mycompany"
  sessionmanagement="Yes">

<!--- Set custom global error handling pages for this application--->
<cferror type="REQUEST"
  template="requesterr.cfm"
  mailto="admin@company.com">
<cferror type="VALIDATION"
  template="validationerr.cfm">

<!--- Set application-specific constants. These are put in the
        Variables scope of every page in the application--->
<cfset homepage="http://www.mycompany.com">
<cfset primarydatasource="CompanyDB">
<!--- set global error handling for this application --->

<!--- set Session variable for this application.--->
<!--- Note that the cfset tag is surrounded by a cflock tag --->
<cflock  timeout="30"
    scope="Session"
    type="exclusive">
  <cfset session.current_location = "Davis, Porter, Alewife">
</cflock>
<cfset mainpage = "default.cfm">
<cfset sm_location = "dpa">
<cfset current_page = "#cgi.path_info#?#cgi.query_string#">
```

# Managing the Client State

Because the Web is a stateless system, each connection that a browser makes to a Web server is unique to the Web server. However, 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**.

ColdFusion provides tools to maintain client state by seamlessly tracking variables for a browser as the user moves from page to page within the application. These variables can be used in place of other methods for tracking client state, such as using URL parameters, hidden form fields, and HTTP cookies.

## About Client and Session variables

ColdFusion provides you with two tools for managing the client state: Client variables and Session variables. Both types of variables are tied to a specific client, but they have significant differences in how they are managed and when they should be used:

| Variable Type | Description |
|---|---|
| Client | Data saved as cookies, database entries, or Registry entries, but is accessed more slowly than data stored in memory.<br>Each type of data storage has its own timeout period. You can specify the database and Registry data timeouts in ColdFusion Administrator. Cookie life depends on the user’s system.<br>Data is stored on a per-application basis. For example, if you store Client variables as cookies, the user has a separate cookie for each ColdFusion application provided by a server.<br>Client variables must be strings (or represented as strings). They cannot be arrays or structures.<br>You do not need to lock code that uses Client variables. |
| Session | Data is stored in memory so it is accessed quickly.<br>Data is lost when the client browser is inactive for a timeout period. You specify the timeout in the ColdFusion Administrator and Application.cfm.<br>Data is available to a single application only.<br>Session variables can be any ColdFusion Data type.<br>You can access the Session scope as a ColdFusion structure.<br>You must lock Session code that uses variables to prevent simultaneous access, for example from different frames. |

As a general rule, it is best to use Session variables for any values that need to exist for only a single browser session. You should reserve Client variables for client-specific data that you want available for multiple browser sessions, such as client preferences.

## About client cookies

Both types of variables normally require ColdFusion to store two client identification variables as cookies on the client's system:

- CFID, a sequential client ID
- CFToken, a random-number client security token

These cookies uniquely identify the client to the ColdFusion Server, which also maintains copies of the variables. You can configure your application so that it does not use client cookies at all, but you must then pass these variables to all pages your application calls.

## Managing client state in a clustered environment

To maintain your application's client state in a clustered server environment, you can store the server-side client identification variables in a common, back-end repository that all Web servers in a multiserver clustered environment can access. You enable use of a common repository by specifying the `cfapplication setdomaincookies` attribute in your Application.cfm page.

This attribute specifies that the server-side copies of the CFID and CFToken variables used to identify the client to ColdFusion are stored at the domain level (for example, .macromedia.com). If CFID and CFToken combinations already exist on each host in the cluster, ColdFusion migrates the host-level variables on each cluster member to the single, common domain-level variable. Following the setting or migration of host-level cookie variables to domain-level variables, ColdFusion creates a new cookie variable (CFMagic) that tells ColdFusion that domain-level cookies have been set.

## Managing client state 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 the CFID and CFToken variables.

To maintain client state without cookies, set the `setClientCookies` attribute of the `cfapplication` tag to No. Then, you must maintain client state by passing CFID and CFToken between pages, either in hidden form fields or appended to URLs. You accomplish this using the variable `Client.URLToken` or `Session.URLToken`. (The values of these two variables are the same.)

# Configuring and Using Client Variables

To use Client variables effectively, you set up the Client variable options and use the variables on your ColdFusion pages.

## Setting up Client variable options

If you want to enable Client variables, 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 ColdFusion 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

After you enable client state management, you must determine where you want to store Client variables. The system-wide default is to store Client variables in the Registry. But you can choose to store them instead in a SQL database or in cookies.

The ColdFusion Administrator Server Client Variables page controls the default Client variable location. You can override the default location by specifying a `clientstorage` attribute in the `cfapplication` tag.

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

- `Registry`(the default)
- Name of a data source configured in the ColdFusion Administrator
- `Cookie`

As a general rule, it is most efficient to store Client variables in a database.

### 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 do not work.

Consider these additional limitations before implementing cookie storage for Client variables:

- Many browsers allow 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.
- Some browsers set a size limit of 4K bytes per cookie. ColdFusion encodes nonalphanumeric data in cookies with a URL encoding scheme that expands at a 3-1 ratio, which means you should not store large amounts of data per client. ColdFusion throws an error if you try to store more than 4000 encoded bytes of data for a client.

### Specifying Client variable storage in Application.cfm

The following example shows how to enable Client variable management using a sample database called *mydatasource*.

#### To specify the Client variable storage method:

1 Open the file Application.cfm in ColdFusion 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.

## Using Client variables

When you enable Client variables for an application, you can use them to keep track of long-term information that is associated with a particular client.

### 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">
```

After you set a Client variable in this manner, it is available for use within any page in your application that is accessed by the client for whom the variable is set.

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

```
<cfparam name="Client.FavoriteColor" default="Red">
```

## Accessing and changing Client variables

You use the same syntax to access a Client variable as other types of variables. You can use Client variables 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

The Client scope has six built-in, read-only variables that your application can use:

| Variable | Description |
|---|---|
| `Client.CFID` | The client ID, normally stored on the client system as a cookie. |
| `Client.CFToken` | The client security token, normally stored on the client system as a cookie. |
| `Client.URLToken` | A combination of the CFID and CFToken in the form `CFID=`*`IDNum`*`&CFTOKEN=`*`tokenNum`*. This variable is useful if the client does not support cookies and you must pass the `CFID` and `CFToken` variables from page to page. |
| `Client.HitCount` | The number of page requests made by the client. |
| `Client.LastVisit` | The last time the client visited the application. |
| `Client.TimeCreated` | The time the `CFID` and `CFToken` variables that identify the client to ColdFusion were first created. |

You can use the `Client.HitCount` and time information variables in customizing 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>
```

## 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 the names of the Client variables for the current application. The standard system-provided Client variables (CFID, CFToken, URLToken, HitCount, TimeCreated, and LastVisit) are not returned in the list.

## Deleting Client variables

Unlike normal variables, Client variables and their values are not stored in volatile memory and persist over time. To delete a Client variable, use the `DeleteClientVariable` function; for example:

```
<cfset IsDeleteSuccessful=DeleteClientVariable("MyClientVariable")>
```

The `DeleteClientVariable` function deletes only Client variables for the application specified by `cfapplication`, if any. For more information on this function, see the *CFML Reference*.

Also, using the Client Variables page of the ColdFusion Administrator Server tab, you can edit the Client variable storage to remove Client variables stored in either the Registry or a database after a set number of days. (The default value is 90 days when Client variables are stored in the registry, 10 days when stored in a data source.)

For more information about setting timeout values, see *Advanced ColdFusion Administration*.

**Note**
You cannot delete the system-provided Client variables (CFID, CFToken, URLToken, HitCount, TimeCreated, and LastVisit).

## Using Client variables with cflocation

If you use the `cflocation` tag to redirect to a path that ends in .dbm or .cfm, the Client.URLToken is automatically appended to the URL. You can suppress this behavior by adding the attribute `addtoken="No"` to the `cflocation` tag.

## Client variable caching

All Client variable reads and writes are cached to help decrease the overhead of client state management operations. For information on variables and server clustering, see *Advanced ColdFusion Administration*.

## Exporting the Client variable database

If your Client variable database is stored in the Windows system registry and you need to move it to another machine, you can export the registry key that stores your Client variables and take it to your new server. The system tegistry allows you to export and import Registry entries.

### To export your Client variable database from the registry in Windows:

1 Open the Registry editor.

2 Find and select the following key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Allaire\ColdFusion\
     CurrentVersion\Clients
```

3 On the Registry menu, click Export Registry File.

4 Enter a name for the registry file.

After you create a registry file, you can copy it to a new machine and import it by selecting Import Registry File on the Registry Editor Registry menu.

On UNIX systems, the registry entries are kept in /opt/coldfusion/registry/cf.registry, a text file that you can copy and edit directly.

## Using Session Variables

Use Session variables when you need the variables for a single site visit or set of requests. For example, you might use Session variables to store a user's selections in a shopping cart application. (Use Client variables when you need the variable for future visits.)

**Caution**
You must use the `cflock` tag to prevent multiple requests from accessing any Session variable simultaneously. Failure to do so can result in data corruption.

### Enabling Session variables

You enable Session variables in the `cfapplication` tag in your Application.cfm file.
To enable Session variables:

- Set `sessionManagement="Yes"`
- Use the `name` attribute to specify the application's name.
- Optionally, use the `sessionTimeout` attribute to set an application-specific session timeout value. Use the `CreateTimeSpan` function to specify the number of days, hours, minutes, and seconds for the timeout.

The following code is an example of turning on session management:

```
<!--- Name the application, and turn on Session management
        with a 45-minute timeout --->
<cfapplication name="GetLeadApp"
        sessionmanagement="Yes"
        sessiontimeout=#CreateTimeSpan(0,0,45,0)#>
```

### 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 timeout period) after that client's last connection. However, because of the stateless nature of the Web, it is not always possible to define a precise point at which a session ends. In the real world, a session ends when the user finishes using an application. In most cases, however, a Web application has no way of knowing if a user is finished or just lingering over a page.

You can impose some structure on Session variable duration by specifying a timeout period. If the user does not access a page of the application within this timeout period, ColdFusion interprets this as the end of the session and clears any variables associated with that session.

The default timeout for Session variables is set to 20 minutes. You can change the timeout on the Memory Variables page of the ColdFusion Administrator Server tab. For more information, see *Advanced ColdFusion Administration*.

You can also set the timeout period for Session variables inside a specific application (thereby overruling the Administrator default setting) by using the `cfapplication` tag `sessionTimeout` attribute.

## 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 can initialize itself with user-specific data the first time a user accesses one of the application's pages. This information can remain available while that user continues to use that application. For example, you can retrieve information about a specific user's preferences from a database once, the first time a user accesses any page of an application. This information remains available throughout that user's session, thereby avoiding the overhead of retrieving the preferences again and again.

Like Client variables, Session variables 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 is the combination of the client ID and the application name.

## Standard Session variables

The Session Client scope has four built-in, read-only variables that your application can use:

| Variable | Description |
|---|---|
| `Session.CFID` | The client ID, normally stored on the client system as a cookie. |
| `Session.CFToken` | The client security token, normally stored on the client system as a cookie. |
| `Session.URLToken` | A combination of the CFID and CFToken in the form `CFID=`*`IDNum`*`&CFTOKEN=`*`tokenNum`*. Use this variable if the client does not support cookies and you must pass the `CFID` and `CFToken` variables from page to page. |
| `Session.SessionID` | A unique identifier for the session. You use this variable in `cflock` tags to identify the scope of the lock. |

If you are also using Client variables, the `Session`.CFID, `Session.CFToken`, and `Session.URL token` are identical to the corresponding Client variables.

## Getting a list of Session variables

The variable Session scope is registered as a ColdFusion structure. This lets you use the ColdFusion Structure functions to get a list of Session variables. For example, you can use `cfloop` with the `StructFind` function to output a list of Session variables defined for a specific application.

To find a list of Client variables, you use the `GetClientVariablesList` function.

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

## Using Application Variables

Application variables are available to all pages within an application, that is, pages that have the same application name. Because they are persistent, you can pass values between pages with a minimum of effort.

You set the application name in the `cfapplication` tag, normally on your application's Application.cfm page. The name you establish in the `cfapplication` tag is accessible elsewhere in the application by using the `Application.ApplicationName` variable. For example, you use this variable in the `cflock` tag to restrict access to Application variables to one request at a time.

Unlike Client and Session variables, Application variables do not require that a client name (client ID) is associated with them. Thus, they are available to any clients that specify the same application name. Also, you do not need to use the `cfapplication` tag to enable Application variables. Using the ColdFusion Administrator you can enable or disable Application variables in all applications; you cannot override this setting for an individual application.

Like Session variables, ColdFusion stores Application variables in the ColdFusion Server's memory and you can set timeout values for these variables either with `cfapplication`, or by specifying timeouts in the ColdFusion Administrator.

For information on setting timeouts for variables, see *Advanced ColdFusion Administration*.

### 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 which client is running that application. Using Application variables, an application could initialize itself, say, when the first user access any page of that application. This information can 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 either a specific period of inactivity passes or the ColdFusion Server shuts 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 or Client variables.

## Application variable timeouts

Application variables have a specific lifetime. If no clients access the application within the specified timeout period, ColdFusion Server destroys its Application variables.

The default timeout period for Application variables is two days. On the Variables page of the ColdFusion Administrator, you can define timeout values for application and Session variables. For more information, see *Advanced ColdFusion Administration*.

You can set the timeout 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.

## Tips for using Application variables

In general, 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 first starts. Then the values of these variables are 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 are not overwritten by other instances of the same application that might be running simultaneously on the server. This is not a problem if you treat these variables as write-once, read-many, but it can be a problem if they are written to indiscriminately.

## Getting a list of Application variables

Like the Session scope, the Application scope is registered as a ColdFusion structure. 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.

## Using Server Variables

Server variables are associated with a single ColdFusion Server. They are available to all applications that run on the server. Use server variables for data that must be accessed across clients and applications.

Server variables do not time out, but are lost when the server shuts down. You cannot delete a Server variable. Therefore, you should limit the amount of data you store in Sever variables.

Server variables are stored on a single server. As a result, you should not use Server variables if you use ColdFusion on a server cluster.

ColdFusion provides the following standard read-only Server variables:

| Variable | Description |
|---|---|
| Server.ColdFusion.ProductName | The name of the product, that is, ColdFusion Server |
| Server.ColdFusion.ProductVersion | The version number for the server that is running, such as 5,0,0 |
| Server.ColdFusion.ProductLevel | The server product level, such as Enterprise |
| Server.ColdFusion.SerialNumber | The serial number assigned to this server installation |
| Server.ColdFusion.Locales | The locales, such as English (US) and Spanish (Mexican), supported by the server |
| Server.OS.Name | The name of the operating system, such as Windows NT |
| Server.OS.AdditionalInformation | Additional information provided by the operating system, such as the Service Pack number |
| Server.OS.Version | The version number of the operating system, such as 4.0 |
| Server.OS.BuildNumber | The specific operating system build, such as 1381 |

# Locking Code with cflock

The `cflock` tag controls simultaneous access to ColdFusion code. The `cflock` tag enables you to:

- Protect sections of code that access and manipulate shared data such as Session, Application, and Server variables.
- Ensure that file updates do not fail because files are open for writing by other applications or ColdFusion tags.
- Ensure that applications do not try to simultaneously access ColdFusion extension tags written using the CFX API that are not thread-safe. This is only necessary for CFX tags that are written in C++ and use shared (global) data structures without protecting them from simultaneous access (are not thread-safe).
- Ensure that applications do not try to simultaneously access databases that are not thread-safe. This is not necessary for most database systems.

Failure to use `cflock` in these circumstances can result in data corruption and can result in hanging the ColdFusion Server. Symptoms of this corruption include the following:

- Unexpected error messages, particularly Unknown Exception errors
- cfserver process crashing, or stopping and restarting
- Unexpected values in shared variables
- Excessive growth in memory used by the cfserver process
- Operating system instability

## Using cflock

You protect access to code by surrounding it in a `cflock` tag; for example:

```
<cflock scope="Application"
  timeout="10"
  type="Exclusive">
  <cfif not isdefined("application.number")>
    <cfset Application.number = 1>
  </cfif>
</cflock>
```

## How cflock works

ColdFusion Server is a multithreaded Web application server that can process multiple page requests at a time. As a result, the server can attempt to access the same information simultaneously as the result of two or more requests. While it is safe to read data simultaneously, attempting to write data simultaneously or read and write it at the same time can result in corrupted memory and can cause the process to crash.

The `cflock` tag enables you to ensure that concurrently executing requests do not access the same section of code simultaneously and thus manipulate shared data structures, files, or CFXs inconsistently. It is important to remember that `cflock` protects code sections *not* variables.

## Lock types

The `cflock` tag offers two modes of locking, specified by the `type` attribute:

- **Exclusive locks** (the default lock type) allow only one request to process the locked code. No other requests are allowed to start running code inside the tag while a request has an exclusive lock. ColdFusion issues exclusive locks on a first-come, first-serve basis.

  Enclose all code that creates or modifies Session, Application, or Server variables in exclusive `cflock` tags.

- **Read-only locks** allow multiple requests to execute concurrently, provided that no exclusive locks are executing. ColdFusion allows you to set variables inside read-only lock tag blocks. However, if you do set a shared variable inside a read-only lock tag, you lose the advantages of locking. As a result, you must be careful not to set any Session, Application, or Server variables inside a read-only `cflock` tag body.

  Enclose code that reads or tests Session, Application, or Server variables in exclusive `cflock` tags. You specify a read-only lock by setting the `type="readOnly"` attribute in the `cflock` tag.

## Lock scopes and names

The `cflock` tag prevents simultaneous access to a sections of code, not variables. If you have two sections of code that access the same variable, they too must be synchronized to prevent them form running simultaneously. You do this by identifying the locks with either `scope` or `name` attributes.

**Note**
ColdFusion does not require you to identify Exclusive locks. If you omit the identifier, the lock is anonymous and you cannot synchronize the code in the `cflock` tag block with any other code. It is acceptable to use an anonymous lock only when the resource you are protecting is used nowhere else in your code. You must always identify read-only locks.

### Controlling access to data with the scope attribute

When the code that you are locking accesses Session, Application, or Server variables, synchronize access by using the `cflock` `scope` attribute.

You can set the attribute to any of the following values:

| Scope | Meaning |
|---|---|
| Server | All code sections with this attribute on the server share a single lock. |
| Application | All code sections with this attribute in the same application share a single lock. |
| Session | All code sections with this attribute that run in the same session and application share a single lock. |

If multiple code sections share a lock, the following rules apply:

- When code in a `cflock` tag block with the type Exclusive is running, code in blocks with the same lock are not allowed run. They wait until the code with the Exclusive lock completes.
- When code in a `cflock` tag block with the type read-only is running, code in other blocks with the same lock and the read-only type can run, but any blocks with the exclusive type cannot run and wait until all code with the read-only lock completes.

### Controlling locking access to files and CFX tags with the name attribute

The `cflock` `name` attribute provides a second mechanism for identifying locks. Use this attribute when you use locks to protect code that manges file access or calls non-thread-safe CFX code.

When you use the `name` attribute, specify the same name for each section of code that accesses a specific file or a specific CFX tag.

## Controlling lock timeouts

You must include a `timeout` attribute in your `cflock` tag. It specifies the maximum time, in seconds, to wait to obtain the lock if it is not available. By default, if the lock does not become available within the timeout period, ColdFusion generates an exception error, which you can handle using `cftry` and `cfcatch`.

If you set the `cflock` `throwOntTmeout` attribute to No, processing continues after the timeout at the line after the `</cflock>` end tag.

Under normal circumstances it should not take more than a few seconds to obtain a lock. Very large timeouts can block request threads for long periods of time and radically decrease throughput. Always use the smallest timeout value that does not result in significant numbers of timeouts.

To prevent unnecessary timeouts, lock the minimum amount of code possible. Whenever possible lock only code that sets or reads variables, not business logic or database queries. One useful technique is to perform a time-consuming activity outside of a `cflock` tag and assign the results to a Variables scope variable, then assign the shared scope variable to the Variables scope variable's value inside a `cflock` block.

For example, if you want to assign the results of a query to a Session variable, first get the query results using a Variables scope variable in unlocked code. Then, assign the query results to a Session variable inside a locked code section. The following code illustrates this technique:

```
<cfquery name="Variables.qUser" datasource="#request.dsn#">
SELECT FirstName, LastName
FROM Users
WHERE UserID = #request.UserID#
</cfquery>
<cflock scope="Session" timeout="2" type="exclusive">
<cfset Session.qUser = Variables.qUser>
</cflock>
```

## Using administrative lock management

You can specify several types of automatic locking and lock checking in ColdFusion Administrator. Use these options when you are developing your code and if you must maintain existing, poorly locked code.

### Automatic lock checking and locking

The Locking page on the Server tab in the ColdFusion Administrator lets you specify the following for variables in each of the three shared memory scopes: Session, Application, and Server.

| Type | Description |
|---|---|
| No automatic checking or locking | ColdFusion does not check for lock use and does not prevent any invalid access of shared variables. |
| Full checking | ColdFusion generates an exception error when your application attempts to use any variable in the scope without protecting it with a lock. |
| Automatic read locking | If your application reads a variable without protecting it, ColdFusion creates a read-only lock for the duration of the read. As a result, ColdFusion blocks any attempt to write to the variable (using code within a lock) until the read completes and the read-only lock is released. If your application writes to any variable in the scope without protecting it with a lock, ColdFusion generates an exception error. |

Selecting the No automatic checking or locking option results in the most efficient code, but requires you to follow the full rules of locking. You should only select this option after you finish debugging your program.

Full checking is very useful for debugging your code. You get errors to help indicate missing locks. You can leave this feature on in production code to protect against inadvertently unlocked variable accesses, but it adds processor overhead for checking all accesses to shared variables, and all locking problems cause exceptions.

Automatic read locking also adds overhead because ColdFusion must insert read locks and check variable access for locking. However, it can be useful if you already have a site that does not use locking properly. In this case, you must only lock all writes and do not have to add locks around all reads.

### Single-threading sessions

The ColdFusion Administrator also allows you to specify single-threaded sessions. If you select this option, each session is handled by a single thread in the ColdFusion Server. As a result, one request from the client must complete before the next one can begin. In this case, you do not need to lock Session variables, but the performance of frames-based pages might be reduced because it prevents simultaneous processing of requests from multiple frames.

## Nesting locks and avoiding deadlocks

Inconsistent nesting of `cflock` tags and inconsistent naming of locks can cause deadlocks (blocked code). If you are nesting locks, you must consistently nest `cflock` tags in the same order and use consistent lock scopes (or names).

A **deadlock** is a state in which no request can execute the locked section of the page. Thus, all requests to the protected section of the page are blocked until there is a timeout. The following table shows one scenario that would cause a deadlock:

| 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 is resolved by a lock timeout.

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 should lock code sections in a well-specified order and name the locks consistently. In particular, if you need to lock access to the Server, Application, and Session scopes, you must do so in the following order.

1. Lock the Session scope. In the `cflock` tag, specify the scope 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 preceding list if you do not need to lock a particular scope. For example, you can omit steps 3 and 4 if you do not need to lock the Server scope.

## Examples of cflock

The following examples show how to use `cflock` in a variety of situations.

### Example with Application, Server, and Session variables

This example shows how you can use `cflock` to guarantee the consistency of data updates to variables in the Application, Server, and Session scopes.

This example does not handle exceptions that arise if a lock times out. As a result, users see the default exception error page on lock timeouts.

The following sample code might be part of the Application.cfm file:

```
<cfapplication name="ETurtle"
  sessiontimeout=#createtimespan(0,1,30,0)#
  sessionmanagement="yes">

<!--- Initialize the session and application
variables that will be used by E-Turtleneck. Use
the session lock scope for the Session variables. --->

<cflock scope="Session"
  timeout="10" type ="Exclusive">
  <cfif not isdefined("session.size")>
    <cfset session.size = "">
  </cfif>
  <cfif not isdefined("session.color")>
    <cfset session.color = "">
  </cfif>
</cflock>

<!--- Use the application lock for the
Application variable. This variable keeps
track of the total number of turtlenecks sold.
use the application lock scope for application variables. --->

<cflock scope="Application"
  timeout="10"
  type="Exclusive">
  <cfif not isdefined("application.number")>
    <cfset application.number = 1>
  </cfif>
</cflock>

<!--- Always display the number of turtlenecks sold --->

<cflock scope="Application"
  timeout="10"
  type ="ReadOnly">
  <cfoutput>
  E-Turtleneck is proud to say that we have sold
  #application.number# turtlenecks to date.
  </cfoutput>
</cflock>
```

The remaining sample code could appear inside the application page where customers place orders.

In this simple example, the Application.cfm page displays the Application.number variable value. Because Application.cfm is processed before any code on each CFML page, the number that displays after you click the submit button does not include the new order. One way you can resolve this problem is by using the OnRequestEnd.cfm page to display the value at the bottom of each page in the application.

```
<html>
<head>
<title>cflock Example</title>
</head>

<body>
<h3>cflock Example</h3>

<cfif isdefined("form.submit")>

<!--- Lock Session variables --->
<!--- Note that we use the automatically generated Session
  ID as the order number --->
<cflock scope="Session"
  timeout="10" type="ReadOnly">
  <cfoutput>Thank you for shopping E-Turtleneck.
  Today you have chosen a turtleneck in size
  <b>#form.size#</b> and in the color <b>#form.color#</b>.
  Your order 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="10"
  type="Exclusive">
  <cfparam name=session.size default=#form.size#>
  <cfparam name=session.color default=#form.color#>
</cflock>
```

```
<!--- Lock Application Variable application.number to
update the total number of turtlenecks sold. --->

<cflock scope="Application"
  timeout="30" type="Exclusive">
  <cfset application.number=application.number + 1>
</cflock>

<!--- Show the form only if it has not been submitted. --->
<cfelse>
<form action="cflock.cfm" method="Post">

<p> Congratulations! You have just selected
the longest wearing, most comfortable turtleneck
in the world. Please indicate the color and size
you want to buy.</p>

<table cellspacing="2" cellpadding="2" border="0">
<tr>
<td>Select a color.</td>
<td><select type="Text" name="color">
    <option>red
    <option>white
    <option>blue
    <option>turquoise
    <option>black
    <option>forest green
    </select>
  </td>
</tr>
<tr>
  <td>Select a size.</td>
  <td><select type="Text" name="size">
    <option>small
    <option>medium
    <option>large
    <option>xlarge
    </select>
  </td>
</tr>
<tr>
  <td></td>
  <td><input type="Submit" name="submit" value="Submit">
  </td>
</tr>
</table>
</form>
</cfif>

</body>
</html>
```

## Example of synchronizing access to a file system

The following example shows 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 30 seconds, a request waiting to obtain an exclusive lock to this code might time out. Also, note the use of a dynamic value for the `name` attribute so that a different locks controls access to each file. As a result, locking access to one file does not delay access to any other file.

```
<cflock name=#filename# timeout=30 type="Exclusive">
  <cffile action="Append"
    file=#fileName#
    output=#textToAppend#>
</cflock>
```

## Example of protecting ColdFusion Extensions

The following example illustrates how you can build a custom tag wrapper around a CFX tag that is not thread-safe. The wrapper forwards attributes to the non-thread-safe CFX tag that is used inside a `cflock` tag.

```
<cfparam name="Attributes.AttributeOne" default="">
<cfparam name="Attributes.AttributeTwo" default="">
<cfparam name="Attributes.AttributeThree" default="">

<cflock timeout=5
    type="Exclusive"
    name="cfx_not_thread_safe">
  <cfx_not_thread_safe attributeone=#attributes.attributeone#
    attributetwo=#attributes.attributetwo#
    attributethree=#attributes.attributethree#>
</cflock>
```

Chapter 13

# 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. You can also use CFScript to write custom functions that you can use anywhere a ColdFusion expression is allowed.

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>` tags. You can use ColdFusion expressions, but not CFML tags, inside a CFScript region.

## CFScript example

The following example shows how you can rewrite a block of `cfset` tags 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#<br>
</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. For information on the `WriteOutput` function, see the *CFML Reference*.

## Supported statements

CFScript supports the following statements:

| | | |
|---|---|---|
| if-else | while | do-while |
| for | break | continue |
| for-in | switch-case | var (in custom functions only) |
| return (in custom functions only) | | |

# The CFScript Language

This section explains the syntax of the CFScript language.

## 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.

## 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.

## Expressions

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

For information about CFML expression, operators, and functions, see the *CFML Reference*.

**Note**
You cannot use CFML tags in CFScript.

## Statements

In CFScript, semicolons define the end of a statement. Line breaks are insignificant. Enclose multiple statements in curly braces to group them for use in an expression:

`{ statement; statement; statement; }`

The following statements are supported in CFScript:

**Assignment:** *lval* = *expr* ;

*lval* can be a simple variable, an array reference, or a member of a structure. For example:

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

**CFML expression:** *expr* ;

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

**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* ;

*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:

```
for(Loop=1;
  Loop LT 10;
  Loop = Loop + 1)
  a[loop]=loop;


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

// Use braces to indicate multiple statements 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);
loop = 1;
while (loop LT 10)
{
 a[loop] = loop + 5;
 loop = loop + 1;
}
```

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

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


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

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

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

No two constants can 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* ;

*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] EQ 0) continue;
  a[loop]=1;
}
```

**break:** break out of the current switch statement or loop

```
indx = 0;
for( ; ; )
{
  indx=indx+1;
  if(Find("key",strings[indx],1))
    break;
}
```

**return**, **var**: See “Defining and Using Custom Functions” on page 250

## 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 | function |
| var | return | | |

## Differences from JavaScript

CFScript is similar to JavaScript, however, there are some key differences in CFScript:

- It uses ColdFusion expressions, which are neither a subset nor a superset of JavaScript expressions. For example, there is no < operator in CFScript.
- It does not have user-defined variable declarations.
- It is case-insensitive.
- All statements end in a semicolon, and line breaks in the code are ignored.
- 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.

## Interaction of CFScript with CFML

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

ColdFusion generates an error if a `cfscript` tag block does not contain at least one CFScript statement, and CFScript comments are not considered statements. To comment out all the contents of a `cfscript` tag block, put ColdFusion comment tags around the entire block, including the `<cfscript>` and `</cfscript>` tags.

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

```
<cfoutput query="patients">

  <cfscript>
  //‘testres' is a column in the "patients" 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#.<br>
</cfoutput>
```

## Defining and Using Custom Functions

You can define **custom** functions (also known as **user-defined functions**) and use them in your application pages as you do standard ColdFusion functions. This allows you to create a function for an algorithm or procedure that you use frequently, and then use the function wherever you need the procedure. If you must change the procedure, you change only one piece of code. You can use your function anywhere that you can use a ColdFusion expression: in tag attributes, between # signs in output, and in CFScript code.

### Defining functions

You define functions using CFScript, in a manner similar to defining JavaScript functions. The function must return a value. Functions can be recursive, that is, the body of a function can call the function.

You can define a function in the following places:

- On the page where it is called (even after it is called, although this is not recommended).
- On a page that you include using a `cfinclude` tag. The `cfinclude` tag must be executed before the function gets called. For example, you can define all your application's functions on a single page and place a `cfinclude` tag at the top of pages that use the functions.

**Syntax** Use the following syntax inside a `cfscript` tag to define a function:

```
function functionName( [paramName1[, paramName2...]] )
{
   CFScript Statements
}
```

*`functionName`*

The name of the function. You cannot use the name of an standard ColdFusion function name. You cannot use the same name for two different function definitions. Function names cannot include periods.

*`paramName1...`*

Names of the parameters required by the function. The number of arguments passed into the function must equal or exceed the number of parameters in the function definition. If the calling page omits any of the required parameters, ColdFusion generates a mismatched argument count error.

The body of the function definition must consist of one or more valid CFScript statements.

The following two statements are allowed only in function definitions. Each function *must* have a **`return`** statement:

**`var`** *`variableName = initialValue;`*

Creates and initializes a variable that is local to the function (**function variable**). This variable has meaning only inside the function and is not saved between calls to the function. It has precedence in the function body over any variables with the same name that exist in any other scopes. You never prefix a function variable with a scope identifier, and their names cannot include periods.

All **`var`** statements must be at the top of the function declaration, before any other statements. You must initialize all variables when you declare them. You cannot use the same name for a function variable and a parameter.

**`return`** *`expression;`*

Evaluates *expression*, returns its value to the page that called the function, and exits the function. You can return any valid ColdFusion variable type, including structures, queries, and arrays.

Each function *must* execute a **`return`** statement.

## Calling functions

You can call a function anywhere that you can use an expression, including in the body of a `cfoutput` tag, in a ColdFusion Script, or in a tag attribute value. One function can call another function, and you can use a function as a parameter to another function.

You call custom functions the same way you call any built-in ColdFusion functions.

## Using arguments and variables

The following sections discuss optional arguments and their use, how arguments get passed, including the effects that the passing methods have, and how you use variables inside functions.

### Optional arguments and the Arguments array

A function can have optional arguments that you do not specify as parameters in the definition. For example, you can write the following function that requires two arguments and supports three additional optional arguments:

```
function MyFunction(MyArg1, MyArg2)
```

Any of the following function calls are then valid:

```
MyFunction(Value1, Value2)
MyFunction(Value1, Value2, Value3)
MyFunction(Value1, Value2, Value3, Value4)
MyFunction(Value1, Value2, Value3, Value4, Value5)
```

Each function has a built-in `Arguments` array containing all arguments passed to the function: the required arguments specified by the function parameters followed by any additional arguments included in the function call.

The function can determine the number of arguments passed to it by using the ColdFusion function call `ArrayLen(Arguments)`.

The function must retrieve the optional arguments by using the `Arguments` array. For example, if the following function:

```
function MyFunction(MyArg1, MyArg2)
```

has three optional arguments, you can refer to the first two, required, arguments as `MyArg1` and `MyArg2` or as `Arguments[1]` and `Arguments[2]`. You *must* refer to the third and fourth and fifth, optional, arguments as `Arguments[3]`, `Arguments[4]`, and `Arguments[5]`.

However, you must be careful if you mix references to the same argument using both its parameter name and its place in the `Arguments` array. Mixing is fine if all you do is read the argument. If you write the argument, you should consistently use either the parameter name or the Arguments array.

## Passing arguments

ColdFusion passes arrays and simple data types including integers, strings, and time and date values into the function by value. It passes queries and structures into the function by reference. As a result, if you change the value of a query or structure argument variable in the function, it changes the value of the variable that the calling page used in calling the function. However, if you change the value of an array or string argument variable in the function, it does not change the value of the string variable in the calling page.

## Using variables

A function can access all variables that are available in the calling page. In addition, the function has its own private scope that contains the function parameters, the `Arguments` array, and the `var`-declared variables. This scope is only accessible inside the current instance of the function. As soon as the function exits, all the variables are removed.

A function can use and change any variable that is available in the calling page, including variables in the caller's Variables (local) scope, as if the function were part of the calling page. For example, if you know that the calling page has a local variable called Customer_name (and there is no `var` variable named Customer_name) the function can read and change the variable by referring to it as "Customer_name" or (using better coding practice) "Variables.Customer_name".

Similarly, you can create a local variable inside a function and then refer to it anywhere in the calling page *after* the function call. You cannot refer to the variable before you call the function.

Because function `var` variables do not take a scope identifier and exist only while the function executes, function variable names can be independent of variable names used elsewhere in your application. If a function must use a variable from another scope that has the same name as a function variable, just prefix the external variable with its scope identifier, such as Variables or Form.

For example, if you use the variable name x for a function scope (var) variable and for a Variables scope (local) variable in the body of a page that calls the function, the two variables are independent of each other. You cannot refer to the function variable in the body of the page, but you can refer to the local variable in the function as `Variables.x`.

## Identifying custom functions

You can use the `IsCustomFunction` function to determine whether a name represents a custom function. The function throws an error if its argument is not defined as a ColdFusion object. As a result, if your code context does not ensure that the name exists, you should use the `isDefined` function to ensure that it is defined. For example:

```
<cfscript>
if( IsDefined("MyFunc"))
  if( IsCustomFunction( MyFunc ))
    WriteOutput( "MyFunc is a custom function");
  else
    WriteOutput( "Myfunc is defined but is NOT a custom function");
else
  WriteOutput( "MyFunc is not defined");
</cfscript>
```

Note that the you do *not* surround the argument to `IsCustomFunction` in quotation marks.

## Examples of custom functons

The following simple examples show the use of custom functions.

**Example 1** This function takes a principal amount, an annual percentage rate, and a loan duration in months and returns the total amount of interest to be paid over the period. You can optionally use the percent sign for the percentage rate, and include the dollar sign and comma separators for the principal amount.

```
<cfscript>
function TotalInterest(principal, annualPercent, months)
{
  Var years = 0;
  Var interestRate = 0;
  Var totalInterest = 0;
  principal = trim(principal);
  principal = REReplace(principal,"[\$,]","","ALL");
  years = months / 12;
  annualPercent = Replace(annualPercent,"%","","ALL");
  interestRate = annualPercent / 100;
  totalInterest = principal*(((1+ interestRate)^years)-1);
  Return DollarFormat(totalInterest);
}
</cfscript>
```

You could use the TotalInterest function in a `cfoutput` tag of a form's action page as follows:

```
<cfoutput>
Loan amount: #Form.Principal#<br>
Annual percentage rate: #Form.AnnualPercent#<br>
Loan duration: #Form.Months# months<br>
TOTAL INTEREST: #TotalInterest(Form.Principal, Form.AnnualPercent,
Form.Months)#<br>
</cfoutput>
```

**Example 2** This function shows the use of optional arguments. It takes two or more arguments and adds them together.

```
<cfscript>
function Sum2(a,b)
{
    var sum = a + b;
    var arg_count = ArrayLen(Arguments);
    var opt_arg = 3;
    for( ; opt_arg LTE arg_count; opt_arg = opt_arg + 1 )
    {
      sum = sum + Arguments[opt_arg];
    }
    return sum;
}
</cfscript>
```

## Using custom functions effectively

These notes provide information that will help you use custom functions more effectively.

### Using Application.cfm

Consider putting custom functions that you use frequently on the Application.cfm page.

### Queries as function parameters

When you call a custom function in the body of a tag that has a `query` attribute, such as `cfloop query=...` tag, a query column name parameter is normally passed as a single element of the column, not the entire column. Therefore, functions that manipulate query data and are called within the bodies of ColdFusion tags with query attributes should only manipulate one query row.

You can use functions that manipulate many rows of a query outside such tags. There you can pass in a query and loop over it in the function. The following example, which changes text in a query column to uppercase, illustrates using a function to modify multiple query rows.

```
function UCaseColumn(myquery, colName)
{
  var currentRow = 1;
  for (; currentRow lte myquery.RecordCount;
    currentRow = currentRow + 1)
  {
    myquery[colName][currentRow] =
      UCase(myquery[colName][currentRow]);
  }
}
```

## Evaluating strings in functions

If your custom function evaluates parameters that contain strings, you must make sure that all variable names in the string are fully qualified to avoid conflicts with function local variables. In the following example, you get the expected results if you pass the fully qualified argument, Variables.myname, but you get the unexpected function local variable value if you pass the argument unqualified, as myname.

```
<CFScript>
  myname = "globalName";
  function readname( name )
  {
    var myname = "localName";
    return (Evaluate( name ));
  }
  </CFScript>

  <cfoutput>
  The result of calling readname with "myname" is:
      #readname("myname")# <br>
<!--- whoops, collides with local variable name --->
  The result of calling readname with "variables.myname" is:
#readname("variables.myname")#
<!--- ok. Finds the name passed in  --->
  </cfoutput>
```

## Passing arrays to custom functions

Arrays, unlike structures, are passed to custom functions by value. This means the function gets a new copy of the array and the array in the calling page is unchanged by the function. For more efficiency, and if you want a function to modify an array in the calling page, store an array as a member of a structure, pass the structure, and reference the array through the structure.

The following example passes an array and a structure containing the array to a function. The function changes the array contents using both of the parameters. The code calls the function and displays the resulting array contents. The change the function makes using the structure appears, but the change the function makes using the directly passed array does not affect the array outside the function.

```
<CFScript>
  //Create a two-element array inside a structure
  mystruct = StructNew();
  mystruct.myarray = ArrayNew(1);;
  mystruct.myarray[1] = "This is the original element one";
  mystruct.myarray[2] = "This is the original element two";

  //Define a custom function to manipulate both
  //an array in a structure (using struct_param) and
  //an array that is passed directly (using array_param).
  function setarray( struct_param, array_param )
  {
    //Change the first element of the array passed in the structure
    struct_param.myarray[1] = "This is the NEW element one";
    //Change the seond element of the directly-passed array
    array_param[2] = "This is the NEW element two";
    return "success";
  }
  //Call the function passing the structure and the array
  setarray( mystruct, mystruct.myarray);
</CFScript>

<CFOutput>
<!--- The element one is changed --->
  <br>#mystruct.myarray[1]#
<!--- Element two is unchanged because the function got a copy --->
  <br>#mystruct.myarray[2]#
</CFoutput>
```

## Error handling

You can handle errors in custom functions by writing a status variable indicating success or failure and some information about the failure. You can also return a special value to indicate failure. The following sketch outlines possible combinations of both these approaches:

```
function bigCalc(x, y, errorInfo)
{
  // Clear error state
  // This allows errorInfo struct to be reused
  structClear(errorInfo);

  var isOK = true;

  // Do work, populate fields in errorInfo such as
  // errorNumber, errorMsg, errorDetail, whatever
  ...

  if (isOK)
  {
    return calculatedValue;
  }
  else
  {
    // Need to return error value
    // Caller will look at value and then decide to look at errorInfo
    // Alternatively, caller can look at errorInfo and see whether
    // some pre-defined fields are available
    return -1; // or "" or whatever we have agreed to...
  }
}

errorInfo = structNew();

result = bigCalc(x, y, errorInfo);

if (result eq -1)
{
  // use information from errorInfo
}


or

if (structKeyExists(errorInfo, "errorMsg"))
{
  // error
}

anotherResult = bigCalc(x + 10, y + 30);

...
```

Chapter 14

# Using Regular Expressions in Functions

This chapter describes how regular expressions work in the following ColdFusion functions:

- `REFind`
- `REFindNoCase`
- `REReplace`
- `REReplaceNoCase`

This chapter does *not* apply to regular expressions used in the `cfinput` and `cftextinput` tags. These tags use JavaScript regular expressions, which have a slightly different syntax than ColdFusion regular expressions.

For information on JavaScript regular expressions, see "Input Validation with cfform Controls," in Chapter 9. For detailed descriptions of the ColdFusion functions that use regular expressions, see the *CFML Reference*.

## Contents

## About Regular Expressions

Regular expressions allow you to perform very powerful and flexible string search and replace operations. In traditional search and replace operations, as in the `Find` and `Replace` functions of ColdFusion, you must provide the exact text to be searched for. This makes searches for dynamic data very difficult, if not impossible. For example, how can you find the first occurrence in a string of any word that consists entirely of capital letters that has spaces around it? Using regular expressions, the task is trivial:

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

You often process large amounts of dynamic textual data. Regular expressions can be invaluable in writing complex ColdFusion applications.

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

## Basic Regular Expression Rules

This section describes the basic rules for creating regular expressions (REs), including single-character regular expressions.

The following are the basic rules for regular expressions:

- Special characters are: `+ * ? . [ ^ $ ( ) { | \`
- Any character that is not a special character matches itself.
- A backslash (\) followed by any special character matches the literal character itself, that is, the backslash escapes the special character.
- A period (.) matches any character, including newline. To match any character except a newline, use [^#chr(13)##chr(10)#], which excludes the ASCII carriage return and line feed codes. The corresponding escape codes are \r and \n.
- A set of characters enclosed in brackets ([]) is a one-character RE that matches any of the characters in that set. For example, "[akm]" matches an "a", "k", or "m". A dash in a set of characters inside braces indicates a range of characters; for example, [a-z] matches any lowercase letter.
- If the first character of a character set is the caret (^), the RE matches any character except those in the set. It does not match the empty string; for example: [^akm] matches any character except "a", "k", or "m". The caret loses its special meaning if it is not the first character of the set.
- To make a regular expression case insensitive, substitute individual characters with character sets; for example, [Nn][Ii][Cc][Kk].
- If you want to include ] (closing square bracket) in square brackets it must be the first character. Otherwise, it does not work even if you use \]. The following example illustrates this:

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

## Character classes

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

```
REReplace ("Macromedia Web Site","[[:space:]]","*","ALL")
```

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

```
Macromedia*Web*Site
```

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

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

## Multicharacter Regular Expressions

You can use the following rules to build multicharacter regular expressions:

- Parentheses group parts of regular expressions together into grouped subexpressions that you can be treat as a single unit; for example, (ha)+ matches one or more instances of “ha”.
- A plus sign (+) following a one-character regular expression or grouped subexpressions matches one or more occurrences of the regular expression; for example, [a-z]+ matches one or more lowercase characters.
- An asterisk (*) following a one-character regular expression or grouped subexpressions matches zero or more occurrences of the regular expression; for example, [a-z]* matches zero or more lowercase characters. Since a regular expression followed by an * can match the empty string, you can get unexpected results when there is no actual match. For example,

  ```
  <cfoutput>REReplace("Hello","[T]*","7","ALL") -
       #REReplace("Hello","[T]*","7","ALL")#<BR></cfoutput>
  ```

  results in the following output:

  ```
  REReplace("Hello","[T]*","7","ALL") - 7H7e7l7l7o
  ```

  Here the regular expression [T]* can match empty strings. It first matches the empty string before “H” in “Hello”. Next, (note that the “ALL” artgument tells `REReplace` to replace all instances of an expression), the empty string before “e” is matched and so on until the empty string before “o” is matched. This result might be unexpected. The workarounds for these types of problems are specific to each case. In some cases you can use [T]+, which requires at least one “T”, instead of [T]*. Alternatively, you might be able to specify an additional pattern after [T]*. In the following example the regular expression has a “W” at the end:

  ```
  <cfoutput>REReplace("Hello World","[T]*W","7","ALL") –
  #REReplace("Hello World","[T]*W","7","ALL")#<BR></cfoutput>
  ```

  This expression results in the following more predictable output:

  ```
  REReplace("Hello World","[T]*W","7","ALL") - Hello 7orld
  ```

- A one-character regular expression or grouped subexpression followed by a question mark (?) matches zero or one occurrences of the regular expression; for example, xy?z matches either “xyz” or “xz”.
- The concatenation of regular expressions creates a regular expression that matches the corresponding concatenation of strings; for example, [A-Z][a-z]* matches any capitalized word.
- The OR character (|) allows a choice between two regular expressions; for example, jell(y|ies) matches either “jelly” or “jellies”.
- The following suffixes match repetitions of a regular expresion:
  - {m,n}, where m is 0 or greater and n is greater than or equal to m, forces a match of m through n (inclusive) occurrences of the preceding regular expression; for example, (ba){2,4} matches “baba”, “bababa”, and “babababa”, but not “ba” or “bababababa”.
  - {m,} forces a match of at least m occurrences of the preceding regular expression. The syntax {,n} is not allowed.

An excellent reference on regular expressions is *Mastering Regular Expressions*, Jeffrey E. F. Friedl. O'Reilly & Associates, Inc., 1997. ISBN: 1-56592-257-3, http://www.oreilly.com.

## Limiting input string size

In CFML regular expression functions, large input strings (greater than approximately 20,000 characters) cause a debug assertion failure and a regular expression error occurs. To avoid this, break your input into smaller chunks, as the following example shows. Here the variable input has a size greater than 50000.

```
<cfset test = mid(input, 1, 20000)>
<cfset out1 = REReplace(test, "[
        #Chr(9)##Chr(13)##Chr(10)#]+#Chr(13)##Chr(10)#", "#chr(10)#",
        "ALL")>
<cfset test = mid(input, 20001, 20000)>
<cfset out2 = REReplace(test, "[
        #Chr(9)##Chr(13)##Chr(10)#]+#Chr(13)##Chr(10)#", "#chr(10)#",
        "ALL")>
<cfset test = mid(input, 40001, len(input) - 40000)>
<cfset out3 = REReplace(test, "[
        #Chr(9)##Chr(13)##Chr(10)#]+#Chr(13)##Chr(10)#", "#chr(10)#",
        "ALL")>
<cfset result = out1 & out2 & out3>
```

## Anchoring a regular expression to a string

You can anchor all or part of a regular expression to either the beginning or end of the string being searched:

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

## Using Backreferences

ColdFusion Server supports **backreferencing**, which allows you to match text in previously matched sets of parentheses. A slash followed by a digit *n* (\*n*) refers to the n$^{th}$ subexpression in parentheses.

One use for backreferencing is in searching for doubled words; for example, to find instances of "the the" or "is is" in text. The following example shows the syntax for backreferencing in regular expressions in ColdFusion:

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

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

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

## Using backreferences in replacement strings

You can use backreferences in replacement strings. Backreferences in the replacement string refer to matched subexpressions in the regular expression search. For example, to replace all repeated words in a text string with a single word, use the following syntax:

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

This results in the sentence:

```
“There is a cat in in the kitchen”
```

You can use the optional fourth parameter in `REReplace`, *scope*, to replace all repeated words, as in the following code:

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

This results in the following string:

```
“There is a cat in the kitchen”
```

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

## Returning Matched Subexpressions

The `REFind` and `REFindNoCase` functions allow you to get information about matched subexpressions. If you set these functions' fourth parameter, `ReturnSubExpression`, to True, the function returns a CFML structure with two arrays containing the positions and lengths of text strings that match the subexpressions of the first instance of a matched regular expression pattern.

The returned structure has two keys, `pos` and `len`. The `pos` array contains the position of the subexpressions. The `len` array has the length of each subexpression. The first element of each array contains information about the complete matched expression, and indices 2 onwards contain information about the parenthesized elements. If there are no occurrences of the regular expression, the `pos` and `len` arrays each contain 1 element with a value of 0.

In the following example the first match for the expression ([A-Aa-z]+)[ ]+, is "is is". The expression [A-Za-z]+ is a subexpression of the complete regular expression.

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

When ColdFusion executes the `ReFind` function, subExprs.pos[1]=7, subExprs.len[1]=5, subExprs.pos[2]=7, and subExprs.len[2]=2.

The entries subExprs.pos[1] and subExprs.len[1] refer to the entire matched expression ("is is"), while subExprs.pos[2] and subExprs.len[2] refer to the first parenthesized subexpression ("is"). Because `REFind` returns information on the first regular expression match only, the subExprs structure does not contain information about the second match to the regular expression, "in in".

For a full discussion of subexpression usage, see the sections on REFind and REFindNoCase in the ColdFusion Functions chapter of the *CFML Reference*.

## Regular Expression Examples

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

| Expression | Description |
|---|---|
| `[\?&]value=` | A URL parameter value in a URL. |
| `[A-Z]:(\\[A-Z0-9_]+)+` | An uppercase DOS/Windows full path that (a) is not the root of a drive, and (b) has only letters, numbers, and underscores in its text. |
| `[A-Za-z][A-Za-z0-9_]*` | A ColdFusion variable with no qualifier. |
| `([A-Za-z][A-Za-z0-9_]*)(\.[A-Za-z][A-Za-z0-9_]*)?` | A ColdFusion variable with no more than one qualifier; for example, Form.VarName, but not Form.Image.VarName. |
| `(\+\|-)?[1-9][0-9]*` | An integer that does not begin with a zero and has an optional sign. |
| `(\+\|-)?[1-9][0-9]*(\.[0-9]*)?` | A real number. |
| `(\+\|-)?[1-9]\.[0-9]*E(\+\|-)?[0-9]+` | A real number in engineering notation. |
| `a{2,4}` | Two to four occurrences of “a”: aa, aaa, aaaa. |
| `(ba){3,}` | At least three “ba” pairs: bababa, babababa, and so on.. |

## Regular expressions in CFML

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

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

| **Expression** | **Returns** |
|---|---|
| `REFind ("[Uu]\.?[Ss]\.?[Aa}\.?", Report )` | The position in the variable Report of the first occurrence of the abbreviation USA. The letters can be in either case and the abbreviation can have a period after any letter. |
| `REFindNoCase("a+c","ABCAACCDD")` | 4 |
| `REReplace("There is is coffee in the the kitchen","([A-Za-z]+)[ ]+\1","*","ALL")` | There * coffee in * kitchen |
| `REReplace(report, "<[^>]*>", "", "All")` | Removes all HTML tags from a string value of the report variable. |

Chapter 15

# 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.

ColdFusion 5 supports two Verity search engines: the default Verity search engine (VDK mode) and a restricted version of the Verity K2 Server. The ColdFusion installation and administration documentation provides information about using both engines. ColdFusion CFML tags for the two engines are identical except for how you specify the `collection` and `external` tag attributes.

This chapter provides and overview of how use the Verity search engines to index and search data for your application.

## Contents

## Searching a ColdFusion Web Site

Until now, you have 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 you need full-text search capabilities. The Verity search engines are bundled with ColdFusion to provide full-text indexing and searching.

Here are some of the ways to use Verity in ColdFusion:

- The ColdFusion online documentation employs Verity to let you to search the installed document set.
- 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. Because 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 e-mail generated by ColdFusion application pages and create a searching mechanism for the indexed messages.
- Build collections of inventory data and make those collections available for searching from your ColdFusion application pages.
- Support international users in a range of languages from the `cfindex`, `cfcollection`, 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. Searching query results has a clear advantage over using SQL to search a database directly 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) in addition to the document text and displays the title in the search results list.

Indexing Web pages returns the URL for each document. This is a valuable document management feature.

# Supported File Types

The ColdFusion Verity implementation supports a wide array of file and document types. As a result, you can index Web pages, ColdFusion applications, and many binary document types and produce search results that include summaries of these documents.

To support multiple WYSIWYG document types, Verity bundles the KeyView Filter Kit. The KeyView Filter Kit includes document filters that support the indexing and viewing of over 45 native document formats. Numerous popular document suites and formats are supported, including Microsoft Office 95, 97 and 2000, Corel WordPerfect, Microsoft Word, Microsoft Excel, Lotus AMI Pro, and Lotus 1-2-3.

The Verity KeyView filters support the following formats:

## Word Processing

- Applix Words (v4.2, 4.3, 4.4)
- HTML (Verity Zone Filter)
- Lotus AmiPro (v2.3)
- Lotus Ami Professional Write Plus
- Lotus Word Pro (v96, 97)
- Microsoft RTF
- Microsoft Word (v2, 6, 95, 97, 2000)
- Microsoft Word Mac (v4, 5, 6)
- Microsoft Word PC (v4.,5, 6)
- Microsoft Works
- Microsoft Write
- PDF (Verity PDF Filter)
- Text files (Verity Text Filter)
- Unicode
- WordPerfect (v5.x, 6, 7, 8)
- WordPerfect Mac (v2, 3)
- XyWrite (v4.12)

## Spreadsheets

- Applix Spreadsheets (v4.3, 4.4)
- Corel QuattroPro (v7, 8)
- Lotus 1-2-3 (v2, 3, 4, 5, 96, 97)
- Microsoft Excel (v3, 4, 5, 96, 97, 2000)
- Microsoft Works spreadsheet

## Presentation Graphics

- Corel Presentations (v.7.0, 8.0)
- Lotus Freelance (v.96, 97)
- Microsoft PowerPoint (v4.0, 95, 97, 2000)

## Support for International Languages

The ColdFusion International Language Search Pack is not shipped with ColdFusion, but is available on a separate CD-ROM free of charge. To order the Language Search Pack, contact Macromedia Customer Service or visit the online store at http://www.coldfusion.com/store. It can be installed to index data in any of 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 other supported languages, you must select the language from the drop-down list when you create a collection on the ColdFusion Administrator Verity page.

The `cfindex` and `cfsearch` tags have an optional `language` attribute that you use to specify the language of the collection you are searching. If you do not specify a language in these tags, ColdFusion checks the registry for the collection's language. If this is defined, ColdFusion uses the language. Otherwise, it assumes that the language is English.

By default the dictionaries are installed in \cfusion\verity\common in Windows and in /opt/coldfusion/verity/common in UNIX.

ColdFusion supports Verity Locales Using LinguistX™ to support localization. These predefined locales have been developed by Verity based on technology from Inxight LinguistX. These languages are supported: English, Danish, Finnish, French, German, Dutch, Italian, Norwegian (Bokmal and Nynorsk), Portuguese, Spanish, and Swedish.

ColdFusion also supports Verity Locales for Asian Languages to enable localization in languages not supported by Verity Locales Using LinguistX. The Verity Locales for Asian Languages are based on ICU (IBM® Classes for Unicode) technology from IBM. These languages are supported: Japanese, Korean, Traditional Chinese, Simplified Chinese, Russian.

## Creating a Searchable Data Source

You must do the following steps to create a searchable data source:

1 Create a collection.

  You can do this either through the ColdFusion Administrator or programmatically.

2 Populate and index the collection.

  You must select the data and generate the index. You can perform this task either through the ColdFusion Administrator or programmatically.

3 Design a search interface and a results page so that users can access the searchable data source.

You can use the Verity Wizard in ColdFusion Studio to create ColdFusion pages that index and search a collection. (The wizard does not create a page for creating the collection.) To run the wizard, select **File > New** and select the Verity Wizard from the CFML tab in the New Document dialog box.

## 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 pages
- Binary documents (see "Supported File Types" on page 271)
- Result sets returned from `cfquery`, `cfldap`, and `cfpop` queries

You can build a collection from individual documents or from an entire directory tree. Collections can be stored anywhere, so you have a great deal of flexibility in accessing indexed data.

You can use either of the following methods to create a Verity collection:

- Make selections on the ColdFusion Administrator Verity Collections page
- Code the `cfcollection` tag

### Using the ColdFusion Administrator to create a collection

You can use the ColdFusion Administrator to create a new collection or map an existing collection.

#### Creating a new collection

Use the following procedure to create a new collection.

#### To create a collection:

1 Open the ColdFusion Administrator Verity Collections 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 the online documents.

2 In the top section of the page, enter a name for the collection.

3 Enter a path for the directory location of the new collection.

By default, new collections are added to \Cfusion\Verity\Collections\ in Windows and /opt/coldfusion/verity/collections in UNIX.

4 If you have an International Language Search Pack installed, you can select a language other than English for the collection from the drop-down list.

5 Make sure Create a new collection is selected.

6 Click Submit Changes.

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.

### Enabling access to an existing collection

You can easily enable access to a collection on the network by creating a local reference (an alias) for that collection. The collection only needs to be a valid Verity collection; it does not matter whether the collection was created within ColdFusion or another tool.

Use the following procedure to enable access to an existing collection.

### To add an existing collection:

1 Open the ColdFusion Administrator Verity Collections page.

2 In the top section of the ColdFusion Administrator Verity Collections page, enter the collection alias.

3 Enter the full path to the collection.

4 Select a language, if needed.

5 Click Map an existing collection.

6 Click Submit Changes.

## Creating a collection with the cfcollection tag

Creating and maintaining collections from a CFML application eliminates the need to access the ColdFusion Administrator. You might prefer this technique if you need to schedule collection creation. It also allows you to allow users to create collections without exposing them to the ColdFusion Administrator.

### To create a simple collection form page:

1 Open a new file in ColdFusion 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 ColdFusion 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

When you create a new collection, it is just an empty shell. You can use either of the following methods to populate a Verity collection:

- Use the ColdFusion Administrator Verity Collections page.
- Code the `cfindex` tag.

**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:

| Use the Administrator if | Use the cfindex tag if |
|---|---|
| You want to index document files. | You want to index ColdFusion query results. |
| The collection will not be updated frequently. | Your collection needs to be updated frequently. |
| You want to generate the collection without writing any CFML code. | You need to dynamically populate or update a collection from a ColdFusion application page. |
| 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.

3 Edit the File Extensions box so that it lists the types of files to index, either as a single file type or multiple file types separated by commas.

4 Type in the directory path containing the files to be indexed or click Browse Server and navigate to the directory in which to begin the index.

5 Select the Recursively Index Sub Directories box if you want to extend the indexing operation to all directories below the selected path.

6 (Optional) Enter a Return URL to prepend to all indexed files. This step lets you easily create a link to any of the files in the index. A typical entry is http://localhost/wwwroot/.

7 (Optional) Select a language. You must have the International Language Search Pack installed.

8 Click Submit Changes 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.

This interface lets you easily build a very specific index based on the file extension and path information you enter. In most cases, you do not need to change your server file structures to accommodate the generation of indices.

In your ColdFusion application, you can populate and search multiple collections, each of which can focus on a specific group of documents or queries, according to subject, document type, location, or any other logical grouping. Because searches can be performed against multiple collections, you have substantial flexibility in designing your search interface.

## Using cfindex

You can use a form page an action page similar to the following examples to select a collection and index it.

### To select which collection to index:

1 Open a new file in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<html>
<head>
  <title>Select the Collection to Index</title>
</head>

<h2>Specify the 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 ColdFusion 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

After you create and index 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. This section describes how to get a basic search application up and running. Later sections explore these options in detail.

### Using the Verity Wizard in ColdFusion Studio

To quickly create a search application for an existing collection, select **File > New** in ColdFusion Studio and select the Verity Wizard in the CFML tab of the New Document dialog box. The wizard creates a set of application pages based on the entries you make in the wizard pages.

You can customize the search interface by adding instructional text for users and applying styles to the form pages.

### Basic search operations

The following procedure describes the basic search operations.

#### To search the collection:

1 Create a file in ColdFusion Studio.

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`.

---

**Note**

To use `cfsearch` to search a Verity K2 Server collection, the `collection` attribute must be the collection's unique alias name as defined in the k2server.ini and the `external` attribute must be "No" (the default).

---

### To present the results of the search to the user:

1 Create a new file in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<html>
<head>
  <title>Search output page</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, and then submit it.

## Creating summaries

As part of the indexing process, Verity automatically produces a summary of every document file or every query result set that gets indexed. 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 Knowledge Base article "Verity Custom1, Custom2 and Summary Fields" (ID# 1081) at http://www.coldfusion.com/Support/KnowledgeBase/SearchForm.cfm.

The `cfsearch` tag returns the summary for each found document in the query variable *search_query*.Summary. For example, to add a summary for each search result returned by the collectionsearchaction.cfm page, change the `cfoutput query` tag as follows:

```
<cfoutput query="Search1">
  <a href="#Search1.URL#">#Search1.title#</a><br>
  #Summary#<br>
</cfoutput>
```

For information on an advanced summarization technique, see the Knowledge Base article "Verity: Synchronizing information stored in Verity Collection" (ID# 1116) at http://www.coldfusion.com/Support/KnowledgeBase/SearchForm.cfm.

## cfsearch properties

Each `cfsearch` query object includes three variables that provide information about the 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 Query Results

The following sections describe the reasons and procedures for indexing the results of database, LDAP, and pop queries.

## Indexing database query results

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. You can search Verity collections containing textual data much more efficiently with `cfindex` than by 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 database tables.

Indexing the result set from a ColdFusion query involves an extra step not required when you index documents. You must code the query and output parameters, and then point the `cfindex` tag at the result set from a `cfquery`, `cfldap`, or `cfpop` query.

### To index a ColdFusion query:

1 Create a collection on the ColdFusion Administrator Verity Collections 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 attribute, which corresponds to the primary key of the data source table, and a body attribute, the column or columns that you want to search for the index. 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 information to be indexed and names the table's primary key, the Message_ID column, as the `key` value. Note that the `title` attribute names the Subject column. You can use the `title` attribute 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
```

## Indexing cfldap query results

The widespread use of the Lightweight Directory Access Protocol to build searchable directory structures, both internally and across the Web, gives you opportunities to add value to the sites you create. You can index contact information or other data from an LDAP-accessible server and allow users to search it.

When creating an index from an LDAP query, remember the following considerations:

- Because LDAP structures vary greatly, you must know the server's directory schema and the exact name of every LDAP attribute you intend to use in a query.
- The records on an LDAP server can be subject to frequent change. You might 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 message number is reset as messages are added and deleted. To avoid mismatches between the unique message number identifiers on the server and in the Verity collection, you should 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.

The following example updates the pop_query collection with the current mail for user1 and searches and returns the message number and subject line for all messages containing the word “action”:

```
<!--- Run POP query --->
<cfpop action="getall"
  name="p_messages"
  server="mail.company.com"
  userName="user1"
  password="user1">

<!--- Output POP query result set --->
<cfoutput query="p_messages">
  #messagenumber# <br>
  #from# <br>
  #to# <br>
  #subject# <br>
  #body# <br>
<hr>
</cfoutput>

<!--- 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>
```

# 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, and you must invoke all aspects of the search explicitly. A simple query expression employs operators by default. You can assemble an explicit query expression programmatically, or you can 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. You can use the following search techniques to search a Verity collection:

- Word searches
- Proximity searches
- Concept–based searches
- Field searches in which documents are matched based on matching predefined custom attributes
- Scoring operators

## Simple query expressions

Simple queries let end users enter simple, comma-delimited strings and use wildcard characters. Users can enter multiple words separated by commas, in which case the comma is treated like a logical OR. If a user omits 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, a simple query expression can include the AND, OR, and NOT operators without angle brackets.

A simple query automatically employs the STEM operator and the MANY modifier. STEM searches for words that derive from those entered in the query expression, so entering “find” returns documents that contain “find,” “finding,” “finds,” and so on. The MANY modifier presents the documents returned in the search as a list based on a relevancy score.

## Explicit query expressions

You can construct explicit queries using a variety of operators, which are described later in this section. Most operators in an explicit query expression must be 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 are stemmed, and whether the words that are found 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 operator, the search engine calculates each document’s score based on the density of the search term in the searched documents. The more frequent is the occurrence of a word in a document, the higher is 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 |

| Verity Search Operators | | |
|---|---|---|
| = | MATCHES | STARTS |
| > | NEAR | STEM |
| >= | NEAR/N | SUBSTRING |
| Accrue | OR | WILDCARD |
| AND | PARAGRAPH | WORD |

## Special characters

The search engine handles a number of characters in particular ways as described in the following table:

| 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; for example:

```
<FREETEXT>("\"Hello\", said Packard.")
"backslash (\\)"
```

## Composing search expressions

The following rules apply to the composition of search expressions.

### Precedence rules

Expressions are read from left to right. The AND operator takes precedence over the OR operator. However, terms enclosed in parentheses are evaluated first. When the search engine encounters nested parentheses, it starts with the innermost term.

### Prefix and infix notation

You use can using prefix notation or infix notation to define search strings that use any operator other than an evidence operator. As a result, either of the following expressions is valid:

- `AND (a,b)`

  This is prefix notation
- `a AND b`

  This is infix notation

When you use prefix notation, the expression specifies precedence explicitly. The following example means: Look for documents that contain b and c first, then documents that contain a:

```
OR (a, AND (b,c))
```

When you use infix notation, 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 the elements. The following example means: Look for documents that contain any combination of a and b together.

```
<OR> (a, b)
```

Note that in this example, angle brackets are used with the OR operator.

### Delimiters in expressions

You use angle brackets (< >), double quotation marks ("), and backslashes (\) to delimit various elements in a query expression, as described in the following table:

| | |
|---|---|
| Angle brackets | 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 | You use double quotation marks in expressions to search for a word that is otherwise reserved as an operator, such as AND, OR, and NOT. |
| Backslashes | To include a backslash in a search expression, insert two backslashes for each backslash character you want included in the search; for example, C:\\CFUSION\\BIN. |

## Searching with wildcards

The following table shows the wildcard characters that you can use to search Verity collections:

| Wildcard | Description |
|---|---|
| ? | Question. Matches any single alphanumeric character. |
| * | Asterisk. Matches 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. Matches any one the characters in the brackets, as in “sl[iau]m” which locates “slim,” “slam,” and “slum.” Square brackets indicate an implied OR. |

| Wildcard | Description |
|---|---|
| { } | Curly braces. Matches any one of a set of patterns separated by a comma, as in “hoist{s, ing, ed}”, which locates “hoists,” “hoisting,” and “hoisted”. |
| ^ | Caret. Matches 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

You must precede the following nonalphanumeric characters with 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`
```

You can use paired backquotes or backslashes to escape special characters. There is no functional difference between the two. For example, you can query for the term: <DDA> in the following ways:

`\<DDA\>` or `‘<DDA>‘`

## Operators and modifiers

The power of the `cfsearch` tag is in the control it provides 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 that a document must meet to be retrieved. You can use operators to refine your search or to influence the results in other ways. For example, you 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.

Ordinarily, you use operators in explicit searches, as shown here:

```
"<operator>search_string"
```

The following operator types are available:

| Operator type | Purpose |
|---|---|
| **Evidence** | Specifies basic and intelligent word searches. |
| **Proximity** | Specifies the relative location of words in a document. |
| **Relational** | Searches fields in a collection. |
| **Concept** | Identifies a concept in a document by combining the meanings of search elements. |
| **Score** | Manipulates the score returned by a search element. You can set the score percentage display to as many as four decimal places. |
| **Natural language** | Allows the use of natural language expressions in forming queries. |

#### Evidence operators

Evidence operators let you specify a basic word search or an intelligent word search. A **basic word search** finds documents that contain only the word or words specified in the query. An **intelligent word search** expands the query terms to create an expanded word list so that the search returns documents that contain variations of the query terms.

Documents retrieved using evidence operators are not ranked by relevance unless you use the MANY modifier.

The following tale describes the 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,” and “believer”. |
| WILDCARD | Matches wildcard characters included in search strings. Certain characters automatically indicate a wildcard specification, such as apostrophe (*) and question mark(?). For example, the query expression: `spam*` yields matches such as, “spam,” “spammer”, and “spamming”. |
| WORD | Performs a basic word search, selecting documents that include one or more instances of the specific word you enter. The WORD operator is automatically implied in any SIMPLE query. |
| THESAURUS | Expands the search to include the word you enter and its synonyms. |
| SOUNDEX | Expands the search to include the word you enter and one or more words that “sound like,” or whose letter pattern is similar to, the word specified. Collections do not have sound-alike indexes by default; to use this feature you must build sound-alike indexes. |
| TYPO/N | Expands the search to include the word you enter plus words that are similar to the query term. This operator performs “approximate pattern matching” to identify similar words. The optional N variable in the operator name expresses the maximum number of errors between the query term and a matched term, a value called the error distance. If N is not specified, an error distance of 2 is used. |

### 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 the 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. 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.<br>You can specify multiple search terms using multiple instances of NEAR/*N* as long as the value of N is the same:<br>`commute <NEAR/10> bicycle <NEAR/10>`<br>`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 in a paragraph, you must use the PARAGRAPH operator between each word or phrase.<br>`<PARAGRAPH> (mission, goal).` |
| 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:<br>`mission oak`<br>`“mission oak”`<br>`mission <PHRASE> oak` |
| SENTENCE | Selects documents that include all of the words you specify within the same sentence. Examples:<br>`jazz <SENTENCE> musician`<br>`<SENTENCE> (jazz, musician)` |
| IN | Selects documents that contain specified values in one or more document zones. A document zone represents a region of a document, such as the document's summary, date, or body text. The IN operator can be qualified with the WHEN operator, to search for a term only within the one or more zones upon which certain conditions have been placed. |

### Relational operators

Relational operators search document fields that you defined in the collection. Documents containing specified field values are returned. Documents retrieved using relational operators are not ranked by relevance, and you cannot use the MANY modifier with relational operators.

You use the following operators for numeric and date comparisons:

| Operator | Description |
|---|---|
| = | Equals |
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |

The following relational operators compare text and match words and parts of words:

| 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, “god” matches “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, “god” matches a document field containing only “god” and does 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. |
| 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, “god” matches “godliness,” “a god among men,” “godforsaken,” and so on. |

### Document fields

You can specify the values for the `cfindex` attributes TITLE, KEY, URL, and CUSTOM as document fields for use with relational operators in the `criteria` attribute. 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 Knowledge Base article, "Verity: Using Document Fields To Narrow Down Searches" (ID# 1082) on our Web site at http://www.coldfusion.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 example described in this section, 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:

| Year | Place | Text |
|---|---|---|
| 1990 | Utah | Text about Utah 1990 |
| 1990 | Oregon | Text about Oregon 1990 |
| 1991 | Utah | Text about Utah 1991 |
| 1991 | Oregon | Text about Oregon 1991 |
| 1992 | Utah | Text about Utah 1992 |

The following application page matches records that have 1990 in the TEXT column and are in the Place Utah. The search is performed against the collection that contains the TEXT column and then is narrowed further by searching for 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>

Query Results --- Should be 5 rows <br>
<cfoutput query="Gettext">
  #Identifier# <br>
</cfoutput>

Search Results -- should be 1 row <br>
<cfoutput query="GetText_Search">
  #GetText_Search.TITLE# <br>
</cfoutput>
```

### Concept operators

Concept operators combine the meaning of search elements to identify a concept in a document. Documents retrieved using concept operators are ranked by relevance. The following table describes each concept operator:

| Operator | Description |
|---|---|
| AND | Selects documents that contain all the search elements 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. |
| ALL | Selects documents that contain all of the search elements you specify. A score of 1.00 is assigned to each retrieved document. ALL and AND retrieve the same results, but queries using ALL are always assigned a score of 1.00. |
| ANY | Selects documents that contain at least one of the search elements you specify. A score of 1.00 is assigned to each retrieved document. ANY and OR retrieve the same results, but queries using ANY are always assigned a score of 1.00. |

### Score operators

Score operators govern how the search engine calculates scores for retrieved documents. The maximum score that a returned search element can have is 1.000. You can set the score percentage display to as many as four decimal places.

When you use a score operator, the search engine first calculates a separate score for each search element found in a document, and then performs a mathematical operation on the individual element scores to arrive at the final score for each document.

Note that the document's score is available as a result column. You can use the SCORE result column to get the relevancy score of any document retrieved. For example:

```
<cfoutput>
  <a href="#Search1.URL#">#Search1.Title#</a><br>
  Document Score=#Search1.SCORE#<BR>
</cfoutput>
```

The following table describes the score operators:

| Operator | Description |
|---|---|
| YESNO | Forces the score of an element to 1 if the element's score is non-zero:<br>`<YESNO>mainframe`<br>If the retrieval result of the search on "mainframe" is 0.75, the YESNO operator forces the result to 1. You can use YESNO to avoid relevance ranking. |
| PRODUCT | Multiplies the scores for the search elements in each document matching a query:<br>`<PRODUCT>(computers, laptops)`<br>Takes the product of the resulting scores. |
| SUM | Adds together the scores for the search element in each document matching a query, up to a maximum value of 1:<br>`<SUM>(computers, laptops)`<br>Takes the sum of the resulting scores. |
| COMPLEMENT | Calculates scores for documents matching a query by taking the complement (subtracting from 1) of the scores for the query's search elements. The new score is 1 minus the search element's original score.<br>`<COMPLEMENT>computers`<br>If the search element's original score is .785, the COMPLEMENT operator recalculates the score as .215. |

## Modifiers

You combine modifiers with operators to change the standard behavior of an operator in some way. For example, you can use the CASE modifier with an operator to specify that you want to match the case of the search word.

The following table describes the available modifiers.

| Modifier | Description |
|---|---|
| CASE | Specifies a case-sensitive search. Normally, Verity searches are case-insensitive for search text entered in all uppercase or all lowercase, and case-sensitive for mixed-case search strings.<br>The expression:<br>`<CASE>J[JAVA, java]`<br>Searches for “JAVA” and “Java.” |
| MANY | Counts the density of words, stemmed variations, or phrases in a document and produces a relevance-ranked score for retrieved documents. Use with the following operators:<br>• WORD<br>• WILDCARD<br>• STEM<br>• PHRASE<br>• SENTENCE<br>• PARAGRAPH<br>Here is an example:<br>`<PARAGRAPH><MANY>javascript <AND> vbscript`<br>You cannot use the MANY modifier with the following:<br>• AND<br>• OR<br>• ACCRUE<br>• Relational operators |
| NOT | Use to exclude documents that contain the specified word or phrase. Use only with the AND and OR operators.<br>Here is an example:<br>`Java <AND> programming <NOT> coffee` |
| ORDER | Use to specify that the search elements must occur in the same order in which they are specified in the query. Use with the following operators:<br>• PARAGRAPH<br>• SENTENCE<br>• NEAR/*N*<br>Place the ORDER modifier before any operator, as follows:<br>`<ORDER><PARAGRAPH>("server", "Java")` |

## Managing Collections

As with any data source, the maintenance requirements of a Verity collection are dictated by the number, 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 Collections page. For more information on this topic, see the Knowledge Base article “Maintaining Collections” (ID# 1080) at http://www.coldfusion.com/Support/KnowledgeBase/SearchForm.cfm.

The easiest way to perform collection management tasks is to create a ColdFusion page that runs the operations, and 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 is corrupted, it is probably safest to repopulate it.
- **Optimize** Packs the indexed data for better performance. You can use this procedure, which is similar to database optimization, as part of routine maintenance.

  You should not use the `Optimize` action in a `cfindex` tag except to maintain legacy code. The `cfcollection` tag is recommended instead. For more information on this command, see the Knowledge Base article “How To Optimize Your Verity Collection” (ID# 416) at http://www.coldfusion.com/Support/KnowledgeBase/SearchForm.cfm.
- **Purge** Removes all data marked for deletion from a collection.
- **Delete** as the `cfindex action` attribute Marks for deletion the specified `key` attribute value, or comma-separated values, from the collection. Use Purge to remove these items.
- **Delete** on the Administrator Verity Collections page or in `cfcollection` Deletes the entire collection.
- **Update** Repopulates 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, and then click Update on the index page.
- **Refresh** (`cfindex action` attribute only) Deletes all data and repopulates the collection.

### Securing a collection

Scenarios for restricting access to a Verity collection include:

- The ColdFusion Administrator might need to specify developer access to collections.
- A public site might need to limit user access to collections.

### To restrict access to a collection, follow these steps:

1. In the ColdFusion Administrator, click the Security tab and select Advanced Security > Security Configuration.
2. Select the Use Advanced Server Security box.
3. Click the Submit Changes button.
4. Click the Security Contexts button.
5. Enter a name for the secured collection and click Add Security Context.
6. (Optional) Enter a description for the secured collection.
7. Select Collection on the Enable Security for Resource Types list.
8. Click Add.

You can then develop an appropriate authentication interface to allow access to the secured collection.

Chapter 16

# Sending and Receiving E-mail

You can add interactive mail features to your ColdFusion applications, providing a 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 e-mail capability a vital link to your users.

## Contents

## Using ColdFusion with Mail Servers

Adding e-mail to your ColdFusion applications lets you respond automatically to user requests. You can use e-mail in your ColdFusion applications in many different ways. These are just a few examples:

- Trigger e-mail messages based on users' requests or orders.
- Allow users to request and receive additional information or documents through e-mail.
- Confirm customer information based on order entries or updates.
- Send invoices or reminders, using information pulled from database queries.

ColdFusion offers several ways to integrate e-mail into your applications. For sending e-mail, you generally use the Simple Mail Transfer Protocol (SMTP). For receiving mail, you use the Post Office Protocol (POP) to retrieve e-mail from the mail server. To use e-mail messaging in your ColdFusion applications, you must have access to an SMTP server and/or a valid POP account.

In your ColdFusion application pages, you use the `cfmail` and `cfpop` tags to send and receive mail respectively. The following sections describe ColdFusion e-mail features and offer examples of these tags.

## Sending E-mail Messages

Before you set up ColdFusion to send e-mail messages, you must have access to an SMTP e-mail server. Also, before you run application pages that refer to the e-mail server, you might want to configure the ColdFusion Administrator to use the SMTP server so that you do not have to hard-code it in your application.

### To configure ColdFusion for e-mail:

1. Open the Mail/Mail Logging page of the ColdFusion Administrator Server tab.
2. In the Mail Server box, enter the name or IP address of the SMTP mail server you want ColdFusion to use.
3. Select the Verify Mail Server Connection check box to make sure ColdFusion can access your mail server.
4. Leave the Server Port and Connection Timeout settings at their default values, unless you need different settings.
5. Click Submit Changes to save the settings.

The page displays a message indicating success or failure at connecting to the server.

For more information on the Administrator's mail settings, see *Advanced ColdFusion Administration*.

## Sending SMTP mail with cfmail

The `cfmail` tag provides support for sending SMTP e-mail from within ColdFusion applications. The `cfmail` tag is similar to the `cfoutput` tag, except that `cfmail` outputs the generated text as SMTP mail messages rather than to a page. The `cfmail` tag supports all the attributes and commands that you use with `cfoutput`, including `query`.

### To send a simple e-mail message:

1 Create a new file in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<html>
<head>
  <title>Sending a simple e-mail</title>
</head>

<body>
<h1>Sample e-mail</h1>
<cfmail
  from="Sender@Company.com"
  to="#URL.email#"
  subject="Sample e-mail"
>
This is a sample e-mail to show basic e-mail capability.

</cfmail>

The e-mail 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; for example:

`http://localhost/myapps/sendmail.cfm?email=myname@mycompany.com`

(Replace `myname@mycompany.com` with your e-mail address.)

The page sends the e-mail to you, through your SMTP server.

## Sample Uses of cfmail

An application page with the `cfmail` tag dynamically generates e-mail messages based on the tag's settings. Some of the tasks you can accomplish with `cfmail` include the following:

- Sending a mail message whose recipient and contents are determined by data the user enters in an HTML form
- Using a query to send a mail message to a database-driven list of recipients
- Using a query to send a customized mail message, such as a billing statement to alist of recipients that is dynamically populated from a database.
- Sending a MIME file attachment along with a mail message

### Sending form-based e-mail

In the following example, the contents of a customer inquiry form submittal are forwarded to the marketing department. Note that the same application page could also insert the customer inquiry into the database.

```
< cfmail
  from="#Form.EMailAddress#"
  to="marketing@MyCompany.com"
  subject="Customer Inquiry">

A customer inquiry was posted to our Web site:

Name: #Form.FirstName# #Form.LastName#
Subject: #Form.Subject#

#Form.InquiryText#

</cfmail>
```

### Sending query-based e-mail

In the following example, a query (ProductRequests) 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@MyCompany.com"
  to="marketing@MyCompany.com"
  subject="Widget status report">

Here is a list of people who have inquired about
MyCompany Widgets over the last seven days:

<cfoutput>
```

```
#ProductRequests.FirstName# #ProductRequests.LastName#
        (#ProductRequests.Company#) -
        #ProductRequests.EMailAddress#&##013;
</cfoutput>

Regards,
The WebMaster
webmaster@MyCompany.com

</cfmail>
```

Note the use of the `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. The `&##013;` in the `cfoutput` block forces a carriage return between output records.

## Sending e-mail to multiple recipients

In the following example, a query (BetaTesters) retrieves 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="BetaTesters"
  from="beta@MyCompany.com"
  to="#TesterEMail#"
  subject="Widget Beta Four Available">

To all Widget beta testers:

Widget Beta Four is now available
for downloading from the MyCompany site.
The URL for the download is:

http://beta.mycompany.com

Regards,
Widget Technical Support
beta@MyCompany.com

</cfmail>
```

Note that in this example, the contents of the `cfmail` tag body are not dynamic, that is, the tag does not use any # delimited dynamic parameters. What is dynamic is the list of e-mail addresses to which the message is sent. Note the use of the TesterEMail column from the BetaTesters query in the `to` attribute.

## Customizing E-mail 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 e-mail to each customer asking the person to verify that the contact information is still valid:

```
<cfmail query="GetCustomers"
  from="service@MyCompany.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,
Customer Service
service@MyCompany.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, and that the mail body therefore does not use a `cfoutput` tag. Also note the use of the other query columns (FirstName, LastName, and so on) 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 e-mail message:

```
<cfmail from="abeecho@MyCompany.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. When a `cfmail` tag is processed in an application page, the messages that are generated are not sent immediately. Instead, they are spooled to disk and processed in the background. This architecture has two distinct advantages:

- End users of your application are not required to wait for SMTP processing to complete before a page returns to them. This design is especially useful when a user action causes more than a handful of messages to be sent.
- Messages sent using `cfmail` are delivered reliably, even in the presence of unanticipated events like power outages or server crashes.

You can set how frequently ColdFusion Server checks for spooled mail on messages on the Mail/Mail Logging page of the ColdFusion Administrator Server tab. (The default interval is 60 seconds.) If ColdFusion is extremely busy or has a large existing queue of messages, however, delivery can occur some time after the spool interval.

### Sending mail as HTML

Most newer Internet mail applications are capable of reading and interpreting HTML code in a mail message. The `cfmail` tag lets you specify the message type as HTML. The `type="HTML"` attribute (the only valid value; the default is plain text) informs the receiving e-mail client that the message has embedded HTML tags that need to be processed. This feature is useful only when you are sending messages to mail clients that understand HTML. Also, you must escape any pound signs in the HTML, such as those used to specify colors, by using two # characters; for example `bgcolor="##C5D9E5"`.

### Error logging and undelivered messages

All errors that occur during the processing of SMTP messages are logged to the file mail.log in the ColdFusion log directory. The log entries contain the date and time of the error as well as diagnostic information on why the error occurred.

All messages not delivered because of an error are written to the \CFusion\Mail\UnDelivr directory on Windows systems and /opt/coldfusion/mail/undelivr on UNIX systems. The error log entry corresponding to the undelivered message contains the name of the file written to the UnDelivr directory.

For more information about the mail logging settings in the ColdFusion Administrator, see *Advanced ColdFusion Administration*.

## Receiving E-mail Messages

the Post Office Protocol tag, `cfpop`, expands your ability to add Internet mail client features and e-mail consolidation to applications. While a conventional mail client provides an adequate interface for personal mail, there are many cases in which 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 e-mail. Here are two instances in which implementing POP mail makes sense:

- If your site has generic mailboxes that are read by more than one person (*sales@yourcompany.com*), it can be more efficient to construct a ColdFusion mail front end to supplement individual user mail clients.
- In many applications, you can automate the processing of mail when the mail is formatted to serve a particular purpose; for example, when subscribing to a list server.

For more information on `cfpop` syntax and variables, see the *CFML Reference*.

## Using cfpop

Use the followig steps to add POP mail to your application

#### 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 must process: message header, message body, attachments, and so on.
3. Decide whether you must store the retrieved messages in a database.
4. Decide whether you must delete messages from the POP server after you retrieve them.
5. Incorporate the `cfpop` tag in your application and create a user interface for accessing a mailbox.
6. Build an application page to handle the output. Retrieved messages can include ASCII characters that do not display properly in the browser.

   You use the `cfoutput` tag with the `HTMLCodeFormat` and `HTMLEditFormat` functions to control output to the browser. These functions convert characters with special meanings in HTML, such as <, >, and &, into HTML escaped characters, such as &lt;, &gt;, and &amp;. The `HTMLCodeFormat` tag also surrounds the text in a `pre` tag block. Note the use of these functions in the examples in this chapter.

## 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

### Retrieving only message headers

The header includes the following information. When you use `cfpop` to get the header or the entire message, ColdFusion returns the values of each these fields in a query column, with one record per retrieved message:

- date
- from
- messageNumber
- replyTo
- subject
- cc
- to

#### To retrieve only the message header:

1 Create a new file in ColdFusion 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=#myusername#
  password=#mypassword#
  action="GetHeaderOnly"
  name="Sample">

<cfoutput query="Sample">
  MessageNumber: #HTMLEditFormat(Sample.messageNumber)# <br>
  To: #HTMLEditFormat(Sample.to)# <br>
  From: #HTMLEditFormat(Sample.from)# <br>
  Subject: #HTMLEditFormat(Sample.subject)# <br>
  Date: #HTMLEditFormat(Sample.date)#<br>
  Cc: #HTMLEditFormat(Sample.cc)# <br>
  ReplyTo: #HTMLEditFormat(Sample.replyTo)# <br><br>
</cfoutput>

</body>
</html>
```

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 and view it in the ColdFusion Studio Browse tab or your Web browser.

This code retrieves the message headers and stores them in a `cfpop` query result set called Sample.

The ColdFusion function `HTMLEditFormat` replaces characters that have meaning to HTML, such as the < and > signs that can surround detailed e-mail address information, with escaped characters such as `&lt;` and `&gt;`.

In addition, you can process the date returned by `cfpop` with `ParseDateTime`, which accepts an argument for converting POP date/time objects into a CFML date-time object.

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

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>
```

## Retrieving an entire message

When you use the `cfpop` tag with `action="GetAll"`, ColdFusion returns the same columns as with `getheaderonly`, plus two additional columns, `body` and `header`.

### To retrieve an entire message:

1 Create a new file in ColdFusion 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=#myusername#
  password=#mypassword#
  action="GetAll"
  name="Sample">

<cfoutput query="Sample">
  MessageNumber: #HTMLEditFormat(Sample.messageNumber)# <br>
  To: #Sample.to# <br>
  From: #HTMLEditFormat(Sample.from)# <br>
  Subject: #HTMLEditFormat(Sample.subject)# <br>
  Date: #HTMLEditFormat(Sample.date)#<br>
  Cc: #HTMLEditFormat(Sample.cc)# <br>
  ReplyTo: #HTMLEditFormat(Sample.replyTo)# <br>
  <br>
  Body:<br>
  #Sample.body#<br>
  <br>
  Header:<br>
  #HTMLCodeFormat(Sample.header)#<br>
  <hr>
  </cfoutput>

</body>
</html>
```

3 Change the following line so that it refers to a valid POP mail server, as well as to 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 and view it in the ColdFusion Studio Browse tab or your Web browser.

Note that this example does not use a CFML function to encode the body contents. As a result, the browser displays the formatted message as you would normally see it in a mail program that supports HTML messages.

## Retrieving attachments with messages

When you use the `cfpop` tag with `action="getAll"`, and use the `attachmentpath` attribute to specify the directory in which to store attachements, ColdFusion gets any attachment files from the POP server and puts them in the specified directory. It also returns two additional columns:

- `attachments` Contains a tab-separated list of all attachment names.
- `attachmentfiles` Contains a tab-separated list of the locations of the attachment files. Use the `cffile` tag to delete these temporary files.

You must make sure that the `attachmentpath` directory exists before you use the `cfpop` tag to get attachments. ColdFusion generates an error if it tries to write an attachment file to a nonexistent directory.

Not all messages have attachments. If a message has no attachments, `attachments` and `attachmentfiles` are empty strings.

### To retrieve all parts of a message, including attachments:

1 Create a new file in ColdFusion 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:\temp\attachments"
  name="Sample">

<cfoutput query="Sample">
  MessageNumber: #HTMLEditFormat(Sample.MessageNumber)# <br>
  To: #HTMLEditFormat(Sample.to)# <br>
  From: #HTMLEditFormat(Sample.from)# <br>
  Subject: #HTMLEditFormat(Sample.subject)# <br>
  Date: #HTMLEditFormat(Sample.date)# <br>
  Cc: #HTMLEditFormat(Sample.cc)# <br>
  ReplyTo: #HTMLEditFormat(Sample.ReplyTo)# <br>
  Attachments: #HTMLEditFormat(Sample.Attachments)# <br>
  Attachment Files: #HTMLEditFormat(Sample.AttachmentFiles)# <br>
  <br>
  Body:<br>
  #Sample.body# <br>
  <br>
  Header:<br>
  HTMLCodeFormat(Sample.header)# <br>
  <hr>
</cfoutput>

</body>
</html>
```

3 Change the following line so that it refers to a valid POP mail server, as well as to 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 and view it in the ColdFusion Studio Browse tab or your Web browser.

**Note**
To avoid duplicate filenames when saving attachments, set the `generateUniqueFilenames` attribute of `cfpop` to Yes.

## Deleting messages

By default, retrieved messages are not deleted from the POP mail server. If you want to delete retrieved messages, you must set the `action` attribute to Delete. You must also specify use the `messagenumber` attribute to specify the numbers of the messages to delete.

Using `cfpop` to delete a message permanently removes it from the server. If the `messagenumber` does not correspond to a message on the server, ColdFusion generates an error.

**Note**
Message numbers are reassigned at the end of every POP mail server communication that contains a delete action. For example, if you retrieve four messages from a POP mail server, the message numbers returned are 1,2,3,4. If you then delete messages 1 and 2 with a single `cfpop` tag, messages 3 and 4 are assigned message numbers 1 and 2, respectively.

### To delete messages:

1 Create a new file in ColdFusion 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 to 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.

---

**Caution**
When you view this page in your browser or the ColdFusion Studio Browse tab, it immediately deletes the messages from your POP server.

---

Chapter 17

# 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, described in “Performing File Operations with cfftp” on page 341.

## 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

You use the `action` attribute to specify any of the following file actions: `upload`, `move`, `rename`, `copy`, `delete`, `read`, `readBinary`, `write`, and `append`. The required attributes depend on the `action` specified. For example, if `action="write"`, ColdFusion expects the attributes associated with writing a text file.

**Note**
Consider the security and logical structure of directories on the server before allowing users access to them. You can disable the `cffile` tag on the Tag Restrictions page of the ColdFusion Administrator Security tab. Also, to access files that are not located on the local ColdFusion Server system, ColdFusion services must run using an account with permission to access the remote files and directories.

## Uploading Files

File uploading requires that you create two files:

- An HTML form to 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 ColdFusion 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.

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<form action="uploadfileaction.cfm"`<br>`    enctype="multipart/form-data"`<br>`    method="post">` | Create a form that contains file selection fields for upload by the user. The `enctype` attribute value tells the server that the form submission contains an uploaded file |
| `<input type="file"`<br>`    name="FiletoUpload"`<br>`    size="45">` | Allow the user to input a field. The `file` type instructs the browser to prepare to read and transmit a file from the user’s system to your server and automatically includes a Browse button to allow the user to look for the file instead of entering the entire path and filename. |

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 ColdFusion 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:\temp"
     nameConflict="overwrite"
     fileField="Form.FiletoUpload">


<cfoutput>
You uploaded the file #cffile.ClientFileName#.#cffile.ClientFileExt#
       successfully to
#cffile.ServerDirectory#\#cffile.ServerFileName#.#cffile.
          ServerFileExt#.
</cfoutput>

</body>
</html>
```

3 Change the following line to point to an appropriate location on your server:

```
destination="c:\temp"
```

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.

The file you specified is uploaded.

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cffile action="upload"` | Prepare to upload a file to the server. |
| `destination="c:\temp"` | Specify the destination of the file. |
| `nameConflict="overwrite"` | If the file already exists, overwrite it. |
| `fileField="Form.FiletoUpload">` | Specify the name of the file to upload. Note that you do not enclose the variable in pound signs. |
| `You uploaded the file #cffile.ClientFileName#.#cffile. ClientFileExt# successfully to #cffile.ServerDirectory#\#cffile. ServerFileName#.#cffile.ServerFileExt#.` | Inform the user of the file that was uploaded and its destination. For information on cffile scope variables, see "Evaluating the Results of a File Upload" on page 324. |

**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 filenames

When you save a file to the server, there is a risk that another file might already exist with the same name. In this case, there are a number of actions that you can take using the `nameConflict` attribute. For example, you can specify the parameter `nameConflict="makeunique"` in the `cffile` tag to create a unique filename while keeping the file extension the same. The unique name might not resemble the attempted name.

## Controlling the type of file uploaded

For some applications, you might want to restrict the type of file that is uploaded. For example, you might not want to accept graphic files in a document library.

You use the `accept` attribute to restrict the type of file that you allow in an upload. When an `accept` qualifier is present, the uploaded file's MIME content type must match the criteria specified or an error occurs. The `accept` attribute takes a comma-separated list of MIME data names, optionally with wildcards.

A file's MIME type is determined by the browser. Common types, 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 saves an image file only if it is in the GIF format:

```
<cffile action="Upload"
  fileField="Form.FiletoUpload"
  destination="c:\uploads\MyImage.GIF"
  nameConflict="Overwrite"
  accept="image/gif">
```

This `cffile` specification saves an image file only if it is in GIF or JPEG format:

```
<cffile action="Upload"
  fileField="Form.FiletoUpload"
  destination="c:\uploads\MyImage.GIF"
  nameConflict="Overwrite"
  accept="image/gif, image/jpeg">
```

This `cffile` specification saves any image file, regardless of the format:

```
<cffile action="Upload"
  fileField="Form.FiletoUpload"
  destination="c:\uploads\MyImage.GIF"
  nameConflict="Overwrite"
  accept="image/*">
```

**Note**
ColdFusion saves any uploaded file if you omit the `accept` attribute, leave it empty, or specify "*/*".

## Setting File and Directory Attributes

In Windows, you specify file attributes using the `cffile` `attributes` attribute. In UNIX, you specify file and directory permissions using the `cffile` and `cfdirectory` `mode` attribute.

## Windows

In Windows, you can set the following file attributes:

- ReadOnly
- Temporary
- Archive
- Hidden
- System
- Normal

To specify several attributes, use a comma-separated list, such as `attributes="ReadOnly,Archive"`. If you do not use `attributes`, the file's existing attributes are maintained. If you specify any other attributes with Normal, the additional attribute overrides the Normal setting.

### Example: Setting file attributes

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**
Make sure you include the trailing slash (\) when you specify the destination directory. Otherwise, ColdFusion treats the last element in the pathname as a filename.

## 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
- 2 = write
- 1 = execute

You enter permissions values in the mode attribute for each type of user: owner, group, and other in that order. For example, use the following code to assign read permissions for everyone:

```
mode=444
```

To give a file or directory owner read/write/execute permissions and read only permissions for everyone else:

```
mode=744
```

## Evaluating the Results of a File Upload

After a file upload is completed, you can retrieve status information using file upload 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 uploaded status variables, cffile is preferred; for example, cffile.ClientDirectory. (The File prefix is retained for backward compatibility.) You can use the file status variables anywhere that you use ColdFusion variables.

The following table describes the file status variables that are available after an upload:

| Variable | Description |
|---|---|
| attemptedServerFile | Initial name ColdFusion used attempting to save a file, for example, `myfile.txt`. See "Resolving conflicting filenames" on page 321. |
| clientDirectory | Directory on the client's system from which the file was uploaded. |
| clientFile | Full name of the source file on the client's system with the file extension; for example, myfile.txt. |
| clientFileName | Name of the source file on the client's system without an extension; for example, myfile. |
| clientFileExt | Extension of the source file on the client's system without a period; for example, txt not .txt. |
| 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 that the uploaded file was last accessed. |
| fileExisted | Indicates (Yes or No) whether the file already existed with the same path. |
| fileSize | Size of the uploaded file. |

| Variable | Description |
|---|---|
| fileWasAppended | Indicates (Yes or No) whether ColdFusion appended the uploaded file to an existing file. |
| fileWasOverwritten | Indicates (Yes or No) whether ColdFusion overwrote a file. |
| fileWasRenamed | Indicates (Yes or No) whether the uploaded file was renamed to avoid a name conflict. |
| fileWasSaved | Indicates (Yes or No) whether ColdFusion saved a file. |
| oldFileSize | Size of a file that was overwritten in the file upload operation. Empty if no file was overwritten. |
| serverDirectory | Directory where the file was saved on the server. |
| serveFile | Full name of the file saved on the server with the file extension; for example, myfile.txt. |
| serverFileName | Name of the file saved on the server without an extension; for example, myfile. |
| serverFileExt | Extension of the file saved on the server without a period; for example, txt, not **.**txt. |
| timeCreated | Date and time the uploaded file was created. |
| timeLastModified | Date and time of the last modification to the uploaded file. |

**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 in the following table show static values for many of the attributes. However, the value of all or part of any attribute in a `cffile` tag can be a dynamic parameter. This makes `cffile` a very powerful tool.

| Action | Example code |
|---|---|
| Move a file | `<cffile action="Move"`<br>`source="c:\files\upload\KeyMemo.doc"`<br>`destination="c:\files\memo\">` |
| Rename a file | `<cffile action="Rename"`<br>`source="c:\files\memo\KeyMemo.doc"`<br>`destination="c:\files\memo\OldMemo.doc">` |
| Copy a file | `<cffile action="Copy"`<br>`source="c:\files\upload\KeyMemo.doc"`<br>`destination="c:\files\backup\">` |
| Delete a file | `<cffile action="Delete"`<br>`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. As a result, you can do the following things:

- Create log files. (You can also use `cflog` to create and write to log files.)
- Generate static HTML documents.
- Use text files to store information that can be brought into Web pages.

## Reading a text file

You can use `cffile` to read an existing text file. The file is read into a local variable that you can use anywhere in the application page. For example, you could read a text file and then insert its contents into a database. Or you could read a text file and then use one of the string replacement functions to modify the contents.

### To read a text file:

1 Create a new file in ColdFusion 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 ColdFusion Studio.

2 Modify the file so that it appears as 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 your name: <input type="text" name="Name" size="25">
  <p>Enter the name of the file: <input type="text" name="FileName"
         size="25">
  <p>Enter your message:</p>
  <textarea name="message"cols=45 rows=6></textarea>
  </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 ColdFusion 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. If the file already exists, it is replaced.

You can use `cffile action="Append"` to append additional text to the end of an existing text file, for example, when you create 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. For details on the syntax of this tag, see the *CFML Reference*.

### Returning file information

When you use the `action="list"` attribute setting, `cfdirectory` returns five result columns that you can reference in a `cfoutput` tag:

- `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` (UNIX only) The octal value representing the permissions setting for the specified directory.

#### To view directory information:

1 Create a new file in ColdFusion 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 cellspacing=1 cellpadding=10>
<tr>
  <th>Name</th>
  <th>Size</th>
  <th>Type</th>
  <th>Modified</th>
  <th>Attributes</th>
  <th>Mode</th>
</tr>
<cfoutput query="mydirectory">
<tr>
  <td>#mydirectory.name#</td>
  <td>#mydirectory.size#</td>
  <td>#mydirectory.type#</td>
  <td>#mydirectory.dateLastModified#</td>
  <td>#mydirectory.attributes#</td>
  <td>#mydirectory.mode#</td>
</tr>
</cfoutput>
</table>

</body>
</html>
```

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.

Note that depending on whether your server is on a UNIX system or a Windows system, either the Attributes column or the Mode column is empty. Also, you can specify a filename in the `filter` attribute to get information on a single file.

Chapter 18

# Interacting with Remote Servers

This chapter describes how ColdFusion wraps the complexity of Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP) communications in a simplified tag syntax that lets you easily extend your site’s offerings across the Web.

## Contents

## Using cfhttp to Interact with the Web

The `cfhttp` tag, which enables you to retrieve information from a remote server, is one of the more powerful tags in the CFML tag set. You can use one of two methods to interact with a remote server using the `cfhttp` tag: Get or Post.

- Using the Get method, you can only send information to the remote server directly in the URL. This method is often used for a one-way transaction in which `cfhttp` retrieves an object.
- By comparison, the Post method can pass variables to a ColdFusion page or CGI program, which processes them and returns data to the calling page. The calling page then displays or further processes the data that was received. For example, when you use `cfhttp` to Post to another ColdFusion page, that page does not display. It processes the request and returns the results to the original ColdFusion page, which then uses the information as appropriate.

## Using the cfhttp Get Method

You use Get to retrieve files, including text and binary files, from a specified server. The retrieved information is stored in a special variable, `cfhttp.fileContent`. The following examples illustrate a few common Get operations.

### To retrieve a file and store it in a variable:

1 Open a new file in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<cfhttp method="Get"
  url="http://www.ci.newton.ma.us/main.htm"
  resolveurl="Yes">
<cfoutput>
  #cfhttp.FileContent# <br>
</cfoutput>
```

3 (Optional) Replace the `url` attribute value with the URL of a file you want to get.

4 Save the file as `getwebpage.cfm` in `myapps` under your Web root directory and view it in your browser.

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfhttp method="Get"`<br>`   url="http://www.ci.newton.ma.us/`<br>`    main.htm/`<br>`   resolveurl="Yes">` | Get the page specified in the URL and make the links absolute instead of relative so that they display properly. |
| `<cfoutput>`<br>`  #cfhttp.fileContent# <BR>`<br>`</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 ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<cfhttp
  method = "Get"
  url="http://www.ci.newton.ma.us/main.htm"
  path="c:\temp"
  file="newtonmain.htm">
```

3 (Optional) Replace the `url` attribute value with the URL of a file you want to save and change the filename.

4 (Optional) Change the path from C:\temp to a path on your hard drive.

5 Save the file as savewebpage.cfm and view it in a text editor. The file does not display properly in your browser because the Get operation saves only the specified Web page. It does not save the frame, image, or other files that the page might include.

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfhttp`<br>`    method = "Get"`<br>`    url="http://www.ci.newton.ma.us`<br>`        /main.htm"`<br>`    path="c:\temp"`<br>`    file="newtonmain.htm">` | Get the page specified in the URL and save it in the file specified in `path` and `file`.<br>When you use the `path` and `file` attributes, ColdFusion ignores any `resolveurl` attribute. As a result, frames and other included files cannot display when you view the saved page. |

### To get a binary file and save it:

1. Open a new file in ColdFusion Studio.
2. Modify the file so that it appears as follows:

```
<cfhttp
    method="Get"
    url="http://localhost/myapps/testfile.zip"
    path="c:\temp"
    file="MyTestFile.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.

## Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfhttp`<br>`    method="Get"`<br>`    url="http://localhost/myapps/testfile.zip"`<br>`    path="c:\temp"`<br>`    file="MyTestFile.zip">` | Get a binary file and save it in the `path` and `file` specified. |
| `<cfoutput>`<br>`    #cfhttp.MimeType#`<br>`</cfoutput>` | Display the MIME type of the file. |

# Creating a Query from a Text File

You can create a query object from a delimited text file by using the `cfhttp` tag and specifying `method="Get"` and the `name` attribute. This is a powerful method for processing and handling generated text files. After you create the query object, you can easily reference columns in the query and perform other ColdFusion operations on the data.

ColdFusion processes text files in the following manner:

- You can specify a field delimiter with the `delimiter` attribute. The default is a comma.
- If data in a field might include the delimiter character, you must surround the entire field with the text qualifier character, which you can specify with the `textqualifier` attribute. The default text qualifier is the double quotation mark (").
- `textqualifier=""` specifies that there is no text qualifier. `textqualifier=""""` (four " marks in a row) explicitly specifies the double quotation mark as the text qualifier.
- If there is a text qualifier, you must surround all field values with the text qualifier character.
- To include the text qualifier character in a field, use a double character. For example, if the text qualifier is ", use "" to include a quotation mark in the field.
- The first row of text is always interpreted as column headings, so that row is skipped. You can override the file's column heading names by specifying a different set of names in the `columns` attribute. You must specify a name for each column. You then use these new names in your CFML code. However, ColdFusion never treats the first row of the file as data.
- When duplicate column heading names are encountered, ColdFusion adds an underscore character to the duplicate column name to make it unique. For example, if two CustomerID columns are found, the second is renamed "CustomerID_".

### To create a query from a text file:

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

```
<!--- The text file has six columns separated by commas: --->
<!--- OrderID,OrderNum,OrderDate,ShipDate,ShipName,ShipAddress --->
<!--- This example uses the first row as the column names --->

<cfhttp method="Get"
  url="http://127.0.0.1/orders/june/orders.txt"
  name="juneorders">

<cfoutput query="juneorders">
  OrderID: #OrderID#<br>
  Order Number: #OrderNum#<br>
  Order Date: #OrderDate#<br>
</cfoutput>
```

```
<!--- Now substitute different column names --->
<!--- by using the columns attribute --->
<hr>
Now using replacement column names<br>

<cfhttp method="Get"
  url="http://127.0.0.1/orders/june/orders.txt"
  name="juneorders"
  columns="ID,Number,ODate,SDate,Name,Address"
  delimiter=",">

<cfoutput query="juneorders">
  Order ID: #ID#<br>
  Order Number: #Number#<br>
  Order Date: #SDate#<br>
</cfoutput>
```

3 Substitute the URL with the location of your text file.

4 Substitute 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 processing. For Post operations, you must use the `cfhttpparam` tag for each variable you want to post. The Post method passes data to a specified ColdFusion page or an executable that interprets the variables being sent and returns data.

For example, when you build an HTML form using the Post method, you specify the name of the page to which form data is passed. You use the Post method in `cfhttp` in a similar way. However, with `cfhttp`, the page that receives the Post does not, itself, display anything.

### To pass variables to a ColdFusion page:

1 Open a new file in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<html>
<head>
  <title>HTTP Post Test</title>
</head>

<body>
<H1>HTTP Post Test</H1>

<cfhttp method="Post"
  url="http://127.0.0.1/myapps/server.cfm">

  <cfhttpparam type="Cookie"
    value="cookiemonster"
    name="mycookie6">
  <cfhttpparam type="CGI"
    value="cgivar "
    name="mycgi">
  <cfhttpparam type="URL"
    value="theurl"
    name="myurl">
  <cfhttpparam type="Formfield"
    value="wbfreuh@macromedia.com"
    name="emailaddress">
  <cfhttpparam type="File"
    name="myfile"
    file="c:\testImage.gif">
</cfhttp>

<cfoutput>
File Content:<br>
  #cfhttp.filecontent#<br>
<br>
Mime Type:  #cfhttp.MimeType#<br>
</cfoutput>
```

```
</body>
</html>
```

3 Replace the path to the GIF file to a path on your server.

4 Save the file as posttest.cfm in `myapps` under your Web root directory.

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfhttp method="Post"`<br>`    url="http://127.0.0.1/myapps/`<br>`    server.cfm">` | Post an HTTP request to the specified page. |
| `<cfhttpparam type="Cookie"`<br>`    value="cookiemonster"`<br>`    name="mycookie6">` | Send a cookie in the request. |
| `<cfhttpparam type="CGI"`<br>`    value="cgivar "`<br>`    name="mycgi">` | Send a CGI variable in the request. |
| `    <cfhttpparam type="URL"`<br>`        value="theurl"`<br>`        name="myurl">` | Send a URL in the request. |
| `<cfhttpparam type="Formfield"`<br>`    value="wbfreuh@macromedia.com"`<br>`    name="emailaddress">` | Send a Form field in the request. |
| `    <cfhttpparam type="File"`<br>`        name="myfile"`<br>`        file="c:\testImage.gif">`<br>`</cfhttp>` | Send a file in the request.<br>The `</cfhttp>` tag ends the http request. |
| `<cfoutput>`<br>`File Content:<br>`<br>`    #cfhttp.filecontent#<br>` | Display the contents of the file that the page that is posted to creates by processing the request. In this example, this is the output from the `cfoutput` tag in server.cfm. |
| `Mime Type:  #cfhttp.MimeType#<br>`<br>`</cfoutput>` | Display the MIME type of the created file. |

### To view the variables:

1 Create a new file in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<cffile destination="C:\temp\Junk"
  nameconflict="Overwrite"
  filefield="Form.myfile"
  action="Upload"
  attributes="Normal">
```

```
<cfoutput>
  The URL variable is: #URL.myurl# <br>
  The Cookie variable is: #Cookie.mycookie6# <br>
  The CGI variable is: #CGI.mycgi#. <br>
  The Formfield variable is: #Form.emailaddress#. <br>
  The file was uploaded to #File.ServerDirectory#\#File.ServerFile#.
</cfoutput>
```

3 Replace C:\temp\Junk with an appropriate directory path on your hard drive.

4 Save the file as `server.cfm` in myapps under your Web root directory.

5 View posttest.cfm in your browser and look for the file in C:\temp\Junk (or your replacement path).

## Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cffile destination="C:\temp\Junk" nameconflict="Overwrite" filefield="Form.myfile" action="Upload" attributes="Normal">` | Write the transferred document to a file on the server. Note that you send the file using the `cfhttpparam type="File"` attribute, but the receiving page gets it as a Form variable, not a File variable. This `cffile` tag creates File variables, as follows. |
| `<cfoutput>` | Output information. The results are not displayed by this page. They are passed back to the posting page in its `cfhttp.filecontent` variable. |
| `The URL variable is: #URL.myurl# <br>` | Output the value of the URL variable sent in the HTTP request. |
| `The Cookie variable is: #Cookie.mycookie# <br>` | Output the value of the Cookie variable sent in the HTTP request. |
| `The CGI variable is: #CGI.mycgi# <br>` | Output the value of the CGI variable sent in the HTTP request. |
| `The Form variable is: #Form.emailaddress#. <br>` | Output the Form variable sent in the HTTP request. Note that you send the variable using the `type="formField"` attribute but the receiving page gets it as a Form variable. |
| `The file was uploaded to #File.ServerDirectory#\#File.ServerFile#. </cfoutput>` | Output the results of the `cffile` tag on this page. This time, the variables really are File variables. |

### To return results of a CGI program:

The following code runs the (theoretical) CGI program search.exe on the (equally theoretical) somesiteorother.com site and displays the results, including both the Mime type and Length of the response. The search.exe program must expect a "search" parameter.

```
<cfhttp method="Post"
       url="http://www.somesiteorother.com/search.exe"
       resolveurl="Yes">
     <cfhttpparam type="Formfield"
       name="search"
       value="Macromedia ColdFusion">
   </cfhttp>

   <cfoutput>
     Response Mime Type: #cfhttp.MimeType#<br>
     Response Length: #len(cfhttp.filecontent)# <br>
     Response Content: <br>
       #htmlcodeformat(cfhttp.filecontent)#<br>
   </cfoutput>
```

## Performing File Operations with cfftp

The `cfftp` tag lets you perform tasks on remote servers using File Transfer Protocol (FTP). You can use `cfftp` to cache connections for batch file transfers.

**Note**
To use `cfftp`, the Enable `cfftp` Tag option must be selected on the Tag Restrictions page of the Basic Security section of the ColdFusion Administrator Security tab.

For server/browser operations, use the `cffile`, `cfcontent`, and `cfdirectory` tags.

Using `cfftp` involves two major types of operations: connecting, and transferring files. The FTP protocol also provides commands for listing directories and performing other operations. For a complete list of attributes that support FTP operations and additional details on using the `cfftp` tag, see the *CFML Reference*.

### To open an FTP connection and retrieve a file listing:

1 Open a new file in ColdFusion Studio.

2 Modify the file so that it appears as follows:

```
<html>
<head>
  <title>FTP Test</title>
</head>

<body>
<h1>FTP Test</h1>
<!--- Open ftp connection --->
<cfftp connection="Myftp"
  server="MyServer"
  username="MyUserName"
  password="MyPassword"
  action="Open"
  stoponerror="Yes">

<!--- Get the current directory name. --->
<cfftp connection=Myftp
  action="GetCurrentDir"
  stoponerror="Yes">

<!--- output directory name --->
<cfoutput>
  The current directory is:  #cfftp.returnvalue#<p>
</cfoutput>

<!--- Get a listing of the directory. --->
<cfftp connection=Myftp
  action="listdir"
  directory="#cfftp.returnvalue#"
  name="dirlist"
  stoponerror="Yes">
```

```
<!--- Close the connection.--->
<cfftp action="close" connection="Myftp">
<p>Did the connection close successfully?
  <cfoutput>#cfftp.succeeded#</cfoutput></p>

<!--- output dirlist results --->
<hr>
<p>FTP Directory Listing:</p>

<cftable query="dirlist" colheaders="yes" htmltable>
  <cfcol header="<B>Name</b>" TEXT="#name#">
  <cfcol header="<B>Path</b>" TEXT="#path#">
  <cfcol header="<B>URL</b>" TEXT="#url#">
  <cfcol header="<B>Length</b>" TEXT="#length#">
  <cfcol header="<B>LastModified</b>"
  TEXT="#DateFormat(lastmodified)#">
  <cfcol header="<B>IsDirectory</b>"
    TEXT="#isdirectory#">
</cftable>
```

3 Change `MyServer` to the name of a server for which you have FTP permission.

4 Change `MyUserName` and `MyPassword` to a valid username and password.

To establish an anonymous connection, enter "anonymous" as the username and an e-mail address (by convention) for the password.

5 Save the file as `ftpconnect.cfm` in `myapps` under your Web root directory.

## Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfftp connection="Myftp"`<br>`    server="MyServer"`<br>`    username="MyUserName"`<br>`    password="MyPassword"`<br>`    action="Open"`<br>`    stoponerror="Yes">` | Open an FTP connection to the MyServer server and log on as MyUserName. If an error occurs, stop processing and display an error. You can use this connection in other `cfftp` tags by specifying the Myftp connection. |
| `<cfftp connection=Myftp`<br>`    action="GetCurrentDir"`<br>`    stoponerror="Yes">`<br>`<cfoutput>`<br>`    The current directory is:`<br>`#cfftp.returnvalue#<p>`<br>`</cfoutput>` | Use the Myftp connection to get the name of the current directory; stop processing if an error occurs.<br>Display the current directory. |

| Code | Description |
|---|---|
| `<cfftp connection=Myftp`<br>`    action="ListDir"`<br>`    directory="#cfftp.returnvalue#"`<br>`    name="dirlist"`<br>`    stoponerror="Yes">` | Use the Myftp connection to get a directory listing. Use the value returned by the last `cfftp` call (the current directory of the connection) to specify the directory to list. Save the results in a variable named dirlist (a query object). Stop processing if there is an error. |
| `<cfftp action="close"`<br>`connection="Myftp">`<br>`<p>Did the connection close`<br>`successfully?`<br>`    <cfoutput>#cfftp.succeeded#</`<br>`cfoutput></p>` | Close the connection, and do not stop processing if the operation fails (because you can still use the results). Instead, display the value of the `cfftp.succeeded` variable, which is Yes if the connection is closed, and No if the operation failed. |
| `<cftable query="dirlist"`<br>`      colheaders="yes" htmltable>`<br>`  <cfcol header="<B>Name</b>"`<br>`    TEXT="#name#">`<br>`  <cfcol header="<B>Path</b>"`<br>`    TEXT="#path#">`<br>`  <cfcol header="<B>URL</b>"`<br>`    TEXT="#url#">`<br>`  <cfcol header="<B>Length</b>"`<br>`    TEXT="#length#">`<br>`  <cfcol header="<B>LastModified</b>"`<br>`    TEXT="#DateFormat(lastmodified)#">`<br>`  <cfcol header="<B>IsDirectory</b>"`<br>`    TEXT="#isdirectory#">`<br>`</cftable>` | Display a table with the results of the ListDir FTP command. |

hafter you establish a connection with `cfftp`, you can reuse the connection to perform additional FTP operations until either you or the server closes the connection. When you access an already-active FTP connection, you do not need to re-specify the username, password, or server. In this case, make sure that when you use frames, only one frame uses the connection object.

**Note**
For a single simple FTP operation, such as GetFile or PutFile, you do not need to establish a connection. Specify all the necessary login information, including the server and any login and password, in the single `cfftp` request.

## Caching connections across multiple pages

The FTP connection established by `cfftp` is maintained only in the current page unless you explicitly assign the connection to a variable with Application or Session scope.

Assigning a `cfftp` connection to an Application variable could cause problems, since multiple users could access the same connection object at the same time. Creating a Session variable for a `cfftp` connection makes more sense, because the connection is available to only one client and does not last past the end of the session.

## Example: Caching a connection

```
<cflock scope="Session" timeout=10>
<cfftp action="Open"
  username="anonymous"
  password="me@home.com"
  server="ftp.eclipse.com"
  connection="Session.myconnection">
</cflock>
```

In this example, the connection cache remains available to other pages within the current session. You must enable Session variables in your application for this approach to work, and you must lock code that uses Session variables. For information on locking, see "Locking Code with cflock" on page 233.

**Note**
Changing a connection's characteristics, such the `retrycount` or `timeout` values, might require you to re-establish the connection.

## Connection actions and attributes

The following table shows the available `cfftp` actions and the attributes they require when you use a named (that is, cached) connection. If you do not specify an existing connection name, you must specify the `username`, `password`, and `server` attributes.

| Action | Attributes | Action | Attributes |
|---|---|---|---|
| Open | none | Rename | existing<br>new |
| Close | none | Remove | server<br>item |
| ChangeDir | directory | GetCurrentDir | none |
| CreateDir | directory | GetCurrentURL | none |
| ListDir | name<br>directory | ExistsDir | directory |
| RemoveDir | directory | ExistsFile | remotefile |
| GetFile | localfile<br>remotefile | Exists | item |
| PutFile | localfile<br>remotefile | | |

# 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 Extensible Markup Language (XML) 1.0 and you can use it to exchange data between CFML applications and other applications.

Additionally, you can use WDDX for server-to-browser and browser-to-server JavaScript data exchanges. You can transfer server data to the browser and convert it to JavaScript objects. You can **serialize** JavaScript data generated on the browser, which translates the native data structures into an abstract representation in XML, and transferr the data to the application server. Conversely, you can **deserialize** WDDX XML into a native data structure. You serialize and deserialize WDDX data using the `cfwddx` tag.

While WDDX is a valuable tool for ColdFusion developers, its usefulness is not limited to CFML. WDDX serialization of common programming data structures (such as arrays, record sets, and structures) enables data communication, using HTTP, across a range of languages and platforms. Also, you can use WDDX to store complex data in a database, file, or even Client variable.

WDDX was created in 1998, and many applications now expose WDDX capabilities. The best source of information about WDDX is http://www.openwddx.org/. This site offers 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

WDDX is an XML vocabulary for describing complex data structures such as arrays, associative arrays, and record sets in a generic fashion so you can move them between different application server platforms and between application servers and browsers using only HTTP. Target platforms for WDDX include ColdFusion, Active Server Pages (ASP), JavaScript, Perl, Java, Python, COM, Macromedia Flash, and PHP.

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 response to the problem of using traditional object-based applications on 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 general internetworking. Instead of the “dumb and disconnected” model of HTTP, distributed computing was built on the assumption of rich network services that 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.

In the disconnected world, however, these services are a barrier to development. At the most fundamental level, the wire protocols of Distributed COM and CORBA are blocked by most Web firewall software. The largest barrier, though, 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 works 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 and Web Services

Independently of WDDX, the Web community has been working on how to solve the general problem of open and flexible distributed computing using a model that is closer to the traditional messaging and client-server systems.

The result is a set of specifications for Web Services, such as Simple Object Access Protocol (SOAP), XML Protocol (XMLP), Web Service Description Language (WSDL) and Universal Description, Discovery, and Integration (UDDI). The importance of Web Services will increase as the technology matures.

WDDX does *not* compete with Web services. It is a complementary technology focused on solving simple problems of application integration on the Web in a pragmatic, productive manner at very low cost.

WDDX is a proven technology that has been shipping for several years. There are numerous applications that leverage WDDX as a foundation for distributed interoperability with other applications.

Currently, WDDX offers some unique advantages not present in Web Services:

- It can be used by lightweight clients, such as browsers or the Macromedia Flash player.
- It can be used to store complex data structures in files and databases.

Applications that take advantage of WDDX can continue to do so even when Web Services become widely available. If need be, these applications could easily be converted to use the upcoming Web Services standards. Only the service and data interchange formats—not the application model—must change.

## WDDX components

WDDX is based on XML, which is a World Wide Web Consortium (W3C) Recommendation. The core of WDDX is an XML vocabulary, a set of components for each of the target platforms to serialize and deserialize 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.

When you use WDDX in ColdFusion pages, you typically use the ColdFusion `cfwddx` tag and the JavaScript utility classes that are installed as `/CFIDE/scripts/wddx.js` on your local host.

## Working with application-level data

The real strength of WDDX is clear if you think of the client and server as a unified platform for applications. There is a subtle, but profound, distinction between this view and the traditional view of an application in which services are partitioned between the client and server.

In client-server, a client might query a database and get a record set 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 integrated into the client side of the application ahead of time.

While this style of data binding 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 by traditional data sources, such as databases. These structures are generally more complex and ad hoc, with dynamic structure. WDDX lets you work with this data without the overhead of setting up a data source 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 the transfer of complex, structured data seamlessly between different application server platforms. For example, an application based on ColdFusion at one business could send a purchase order to a supplier running a CGI-based system. The supplier can 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 other system components.

### 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 ColdFusion WDDX serializers (CFML and JavaScript) have a `useTimezoneInfo` attribute or property that specifies whether to use time zone information in the serialization process. The default value is True.

In the CFML implementation, `useTimezoneInfo` is an attribute of the `cfwddx action=cfml2wddx` tag. In the JavaScript implementation, `useTimezoneInfo` (note the case sensitivity of JavaScript) 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, 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 you 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 determining 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 structure with two string variables might have the following form after it is serialized into a WDDX XML representation:

```
<var name=’x’>
  <struct>

    <var name=’a’>
      <string>Property a</string>
    </var>

    <var name=’b’>
      <string>Property b</string>
    </var>

  </struct>
</var>
```

When the WDDX Deserializer object deserializes this XML, it creates a structure that is also created by either of the following scripts:

| JavaScript | CFScript |
|---|---|
| `x = new Object();`<br>`x.a = "Property a";`<br>`x.b = "Property b";` | `x = structNew();`<br>`x.a = "Property a";`<br>`x.b = "Property b";` |

The `WddxSerializer` and `WddxDeserializer` objects are defined in the file CFIDE/scripts/wddx.js. For detailed information on these JavaScript objects, see the *CFML Reference*.

## Converting CFML Data to a JavaScript Object

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

The application consists of four principal sections:

- Running a data query
- Including the WDDX JavaScript utility classes
- Calling the conversion function
- Outputting the object data in HTML

The following example uses the cfsnippets data source that is installed with ColdFusion:

```
<!--- Create a simple query  --->
<cfquery name = "q" datasource ="cfsnippets">
  SELECT Message_Id, Thread_id, Username, Posted
  FROM messages
</cfquery>

<!--- Load the wddx.js file, which includes the dump function --->
<!--- requests to the server --->
<script type="text/javascript"
  src="/CFIDE/scripts/wddx.js"></script>

<script>
  // Use WDDX to move from CFML data to JS
  <cfwddx action="cfml2js" input="#q#" topLevelVariable="qj">

  // Dump the record set to show that all the data has reached
  // the client successfully.
  document.write(qj.dump(true));
</script>
```

**Note**

To see how `cfwddx Action="cfml2js"` works, view the source to the page in your browser.

## 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 that generate complex JavaScript data collections, you can extend this basic approach very effectively. Note that this example uses the `WddxSerializer` JavaScript object to serialize the data, and the `cfwddx` tag to deserialize the data.

### To use the example:

1 Save it in the myapps directory as wddxSerializeDeserialze.cfm

2 Display http://localhost/myapps/wddxSerializeDeserialze.cfm in your browser.

3 Enter a first name and last name in the form fields.

4 Click Next.

   The name appears in the Names added so far box.

5 Repeat steps 3 and 4 to add as many names as you wish.

6 Click Serialize to serialize the resulting data.

   The resulting WWDX packet appears in the WDDX packet display box. This step is intended only for test purposes. Real applications handle the serialization automatically.

7 Click Submit to submit the data.

   The WDDX packet is transferred to the server-side processing code, which deserializes it and displays the information.

```
<!--- load the wddx.js file --->
<script type="text/javascript"
  src="/CFIDE/scripts/wddx.js"></script>

<!--- Data binding code --->
<script>

  // Generic serialization to a form field
  function serializeData(data, formField)
  {
    wddxSerializer = new WddxSerializer();
    wddxPacket = wddxSerializer.serialize(data);
    if (wddxPacket != null)
    {
      formField.value = wddxPacket;
    }
    else
    {
      alert("Couldn’t serialize data");
    }
  }
  // Person info record set with columns firstName and lastName
  // Make sure the case of field names is preserved
  var personInfo = new WddxRecordset(new Array("firstName",
  "lastName"), true);

  // Add next record to end of personInfo record set
  function doNext()
  {
    // Extract data
    var firstName = document.personForm.firstName.value;
    var lastName = document.personForm.lastName.value;

    // Add names to record set
    nRows = personInfo.getRowCount();
    personInfo.firstName[nRows] = firstName;
    personInfo.lastName[nRows] = lastName;

    // Clear input fields
    document.personForm.firstName.value = "";
    document.personForm.lastName.value = "";

    // Show added names on list
    // This gets a little tricky because of browser differences
    var newName = firstName + " " + lastName;
    if (navigator.appVersion.indexOf("MSIE") == -1)
    {
      document.personForm.names[length] =
        new Option(newName, "", false, false);
    }
    else
    {
      // IE version
      var entry = document.createElement("OPTION");
      entry.text = newName;
      document.personForm.names.add(entry);
    }

  }

</script>


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

  <!--- Input fields --->
  Personal information<p>
  First name: <input type=text name=firstName><BR>
  Last name: <input type=text name=lastName><BR>
  <p>
  <!--- Navigation & submission bar --->
  <input type="button" value="Next" onclick="doNext()">
  <input type="button" value="Serialize"
  onclick="serializeData(personInfo, document.personForm.wddxPacket)">
  <input type="submit" value="Submit">
  <P>
  Names added so far:<br>
  <select name="names" size="5">
  </select>
  <p></p>

  <!--- This is where the WDDX packet will be stored --->
  <!--- In a real application this would be a hidden input field. --->
  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>
```

## Storing Complex Data in a String

The following simple example uses WDDX to store complex data, a data structure that contains arrays as a string in a Client variable. It uses the `cfdump` tag to display the contents of the structure before serialization and after deserialization. It uses the `HTMLEditFormat` function in a `cfoutput` tag to display the contents of the Client variable. The `HTMLEditFormat` function is required to prevent the browser from trying to interpret (and throwing away) the XML tags in the variable.

```
<!--- Enable client state management --->
<cfapplication name="relatives" clientmanagement="Yes">

<!--- Build a complex data structure --->
<cfscript>
  relatives = structNew();
  relatives.father = "Bob";
  relatives.mother = "Mary";
  relatives.sisters = arrayNew(1);
  arrayAppend(relatives.sisters, "Joan");
  relatives.brothers = arrayNew(1);
  arrayAppend(relatives.brothers, "Tom");
  arrayAppend(relatives.brothers, "Jesse");
</cfscript>

A dump of the original relatives structure:<br>
<br>
<cfdump var="#relatives#"><br>
<br>

<!--- Convert data structure to string form and save it in the
       client scope --->
<cfwddx action="cfml2wddx" input="#relatives#"
output="Client.wddxRelatives">

The contents of the Client.wddxRelatives variable:<br>
<cfoutput>#HtmlEditFormat(Client.wddxRelatives)#</cfoutput><br>
<br>


<!--- Now read the data from client scope into a new structure --->
<cfwddx action="wddx2cfml" input="#Client.wddxRelatives#"
output="sameRelatives">

A dump of the sameRelatives structure <br>
generated from client.wddxRelatives<br>
<br>
<cfdump var="#sameRelatives#">
```

Chapter 19

# Application Security

ColdFusion supports several levels of security. This chapter explains how to deploy user security, which offers runtime security for ColdFusion applications. It also describes the Remote Development Services security feature, which authenticates developers accessing server resources through ColdFusion Studio.

For information on setting up security elements or using Administrator-controlled security features, see *Advanced ColdFusion Administration*.

## 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 the team. They can coordinate team development on shared servers with the assurance that sensitive data and applications are secure.
- **Application deployment** ColdFusion developers can create complex rules to programmatically control access to functionality within applications. You can confine applications to secure areas, thereby flexibly restricting the access that the 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 in which groups of developers, working in ColdFusion Studio, require different levels of access to ColdFusion files and data sources.

When you are 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 Windows 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, such as browsing, editing, database operations, debugging, and so on.
- **Advanced Security** Lets ColdFusion administrators restrict or permit access to file systems and data sources based on security contexts and policies established on 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 *Advanced ColdFusion Administration*.

## 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 use ColdFusion to build and host your company's intranet. The Human Resources department maintains a page on the intranet on which all employees can access timely information about the company, such as the latest company policies, upcoming events, and job postings. You want everyone to be able to read the information, but you want only certain authorized Human Resources employees to be able to add, update, or delete information.

In addition, you might want to let employees view customized information about their salaries, job levels, and performance reviews. You certainly would not want one employee to view sensitive information about another employee, but you would want managers to be able to see, and possibly update, information about their direct reports. User security authenticates and authorizes users each time that they try to access or work with sensitive data.

User security is made up of two components:

- Security contexts, configured on the Advanced Security page of the ColdFusion Administrator. 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 whether a user is allowed to access a particular resource and then takes appropriate action.

Before you can implement user security in your applications, you must make sure that your ColdFusion administrator installed Advanced Security on the server and 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 detailed information about installing Advanced Security and setting up a security framework, see *Advanced ColdFusion Administration*.

## 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 whether they are authorized for the part of the application they are 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 ensure that users are who they say they are. You typically use `cfauthenticate` in your application's Application.cfm file. Pass the authentication information to subsequent pages on which 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 whether 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 363 to see code examples that show how these tags and 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 Reference*.

### Encrypting application pages

For an added measure of security, you can encrypt strings in your applications using the `Encrypt` and `Decrypt` functions. For descriptions of these functions, see the *CFML Reference*.

## 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 on the Advanced Security page of the ColdFusion Administrator.
- `username` The user name required to access the protected resources.
- `password` The password required to access the protected resources.

You usually set the `username` and `password` attributes using variables that are 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="MyAppSecurityContextName"
       username=#userID#
       password=#pwd#>
```

If the user is not already defined in the system, ColdFusion throws a Security exception. You can either reject access to the resource or reroute 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.

For a longer code example, see “Example of User Authentication and Authorization” on page 363.

## Authentication and Authorization Functions

After you use `cfauthenticate` to check whether 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 whether the current session was authenticated by the `cfauthenticate` tag.
- `IsAuthorized` checks whether the authenticated user has access to the named resource, based on rules defined in the security context for which the user is authenticated.

### Using the IsAuthenticated function

The `IsAuthenticated` function checks whether a `cfauthenticate` tag successfully executed for the current request. If not, it looks for the `cfauth` cookie to determine whether the user is authenticated. If you do not set a `cfauth` cookie with `cfauthenticate`, you must call `cfauthenticate` for every request in the application.

The `IsAuthenticated` function returns True if the user is authenticated for the current request; otherwise, it returns False.

If you call `IsAuthenticated with` the optional *security_context_name* parameter, the function returns True if the user is authenticated in the named security context; otherwise it returns False. The `IsAuthenticated` function has the following form:

```
IsAuthenticated("security_context_name")
```

### Using the IsAuthorized function

After a user is authenticated, you can use the `IsAuthorized` function to check which resources the user is allowed to access. You define 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, as follows:

```
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 to update the named data source, or if the data source is not protected in the security context.

**Note**
The ColdFusion Server does not check user authorization unless you specifically request it with the `IsAuthorized` function. It is up to you to decide what action to take based on the results of the `IsAuthorized` call.

## Catching Security Exceptions

You can use the `cftry` and `cfcatch` tags 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 failures from the `IsAuthorized` or `IsAuthenticated` functions.

Set the `cfauthenticate` `throwOnFailure` attribute to `Yes` and enclose the 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 208.

### Example

This example shows the use of exception handling with `cfauthenticate` in an Application.cfm file. The `cfauthenticate` tag authenticates a user and sets the security context for an application.

If the user is not already defined in the system, you can either reject the page, request that the user respecify the username and password, or define a new user. The following example just rejects the page request and displays a message:

```
<html>
<head>
  <title>cfauthenticate Example</title>
</head>

<body>
<h3>cfauthenticate Example></h3>

<!--- This code is from an Application.cfm file --->

<cftry>
  <cfauthenticate securityContext="MyApplicationSC"
    username=#user#
    password=#pwd#>
  <cfcatch type="Security">
    <!--- The message to display --->
    <h3>Authentication error</h3>
<!--- display a message. Alternatively, you might place code
  here to define the user to the security context. --->
    <cfoutput>
    <p>#cfcatch.Message#</p>
    </cfoutput>
  </cfcatch>
</cftry>

<cfapplication name="Personnel">

</body>
</html>
```

## Using the cfimpersonate Tag

The `cfimpersonate` tag gives ColdFusion developers a way to execute a segment of code in a secure manner. This tag is useful when you want to briefly grant a type of access that you would normally withhold. Suppose you are an internet service provider (ISP) who hosts ColdFusion development services. You provide a set of custom tags that let your customers add features such as hit counters, guest books, and message boards to the ColdFusion applications they create. To provide this type of functionality, you must also provide access to some resources that you might prefer to protect. 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 do not want your customers to access the `cffile` tag on your servers. However, if you provide your customers with a hit counter, you must let them read specific, system-maintained files, in this case, the file that contains number of hits to the customer's home page. You can provide the hit-counter in a custom tag that uses 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 nontrusted user after the trusted piece of code executes.

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 on 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, CF for application level or OS for 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 you might implement user security by authenticating users and then allowing users to see or use only the resources that 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:

1 The Application.cfm page checks to see whether the current user is *authenticated*. If not, the page presents a login form and the user must submit a username and password for authentication.

   If the user passes the authentication test, ColdFusion passes the `cfauth` cookie to carry the user’s authentication state to subsequent application pages governed by this Application.cfm page.

2 Only authenticated users can 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

The following example code for an Application.cfm page checks first to see whether the current user is authenticated by checking to see whether 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, the requested page appears. The application uses the CGI variables `script_name` and `query_string` to keep track of the originally requested page so that it can display that page after the user is authenticated.

All pages governed by this Application.cfm page — those in the same directory as Application.cfm and in its subtree — automatically 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. Try authenticating --->
    <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. Show login again--->
    <cfset showlogin="Yes">
  </cfif>

<!--- Show the login form --->
  <cfif showlogin>
<!--- Recreate the url used to call the requested page --->
    <cfset url="#cgi.script_name#">
    <cfif cgi.query_string IS NOT "">
      <cfset url=url & "?#cgi.query_string#">
    </cfif>
```

```
<!--- The login form.
    Submitting the form re-requests the originally requested page
    using 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, you 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 previous Application.cfm page. It uses the `IsAuthorized` function to test whether authenticated users are allowed to update or select data from a data source.

### Example: orders.cfm

```
<!--- 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 #Customer#, #OrderID#
    </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 20

# Using cfobject to Invoke Component Objects

This chapter describes how to use the `cfobject` tag to invoke objects created by component technologies, including COM/DCOM, CORBA, and Java 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 toMicrosoft's COM site, http://www.microsoft.com/com.

## 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, http://www.omg.com, is the main Web repository for CORBA information.

## About Java objects

Java objects include any Java class available in the class path specified on the ColdFusion Administrator JVM and Java Settings page.

# Invoking Component Objects

You use the `cfobject` tag to create an instance of an object. You use other ColdFusion tags, such as `cfset` and `cfoutput`, 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 ColdFusion page execution.

The examples in the following sections assume that the `name` attribute in the `cfobject` tag specified the value “obj”, and that the object has a property called “Property”, and methods called “Method1”, “Method2”, and “Method3”.

## Using properties

Use the following coding practices to access properties.

### 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-quotation marks (") to specify reference arguments. If the object changes the value of "x", it now contains 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 must invoke a property or method on that object, use the following syntax:

```
<cfset objX = myObj.X>
        <cfset prop = objX.Property>
```

(The syntax `<cfset prop = myObj.X.Property>` fails.)

## Getting Started with COM/DCOM

ColdFusion is an automation (late-binding) COM client. As a result, the COM object must support the IDispatch interface, and arguments for methods and properties must be standard automation types. Because ColdFusion is a typeless language, it uses the object's type information to correctly set up the arguments on call invocations. Any ambiguity in the object's data types can lead to unexpected behavior.

In ColdFusion, you should only use server-side COM objects, which do not have a graphical user interface. If your ColdFusion application invokes an object with a graphical interface in a window, the component might appear on the Web server desktop, not on the user's desktop. This can take up ColdFusion Server threads and 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:

- Microsoft OLE/COM Object Viewer, available from Microsoft at http://www.microsoft.com/com/resources/oleview.asp. This tool is handy for viewing registered COM objects.
- The COM objects, which are typically DLL or EXE files, that you want to use in your ColdFusion application pages. These components should allow late binding, that is, they should implement the IDispatch interface. Object Viewer lets you view an object's class information so that you can properly define the `class` attribute for the `cfobject` tag. It also displays the object's supported interfaces, so you can discover the properties and methods (for the IDispatch interface) of the object.

## Registering the object

after you acquire the object you want to use, you must register it with Windows in order for ColdFusion (or any other program) to find it. Some objects might be deployed with their own setup programs that register objects automatically, while others might require manual registration.

You can register Inproc object servers (*.dll, *.ocx) manually by running the regsvr32.exe utility using the following form:

```
regsvr32 c:\path\servername.dll
```

You typically register local servers (*.exe) either by starting them or specifying a command line parameters, such as the following:

```
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 are ready to use the `cfobject` tag. If you do not have documentation, use the Object Viewer to view the component's interface.

### Using the OLE/COM Object Viewer

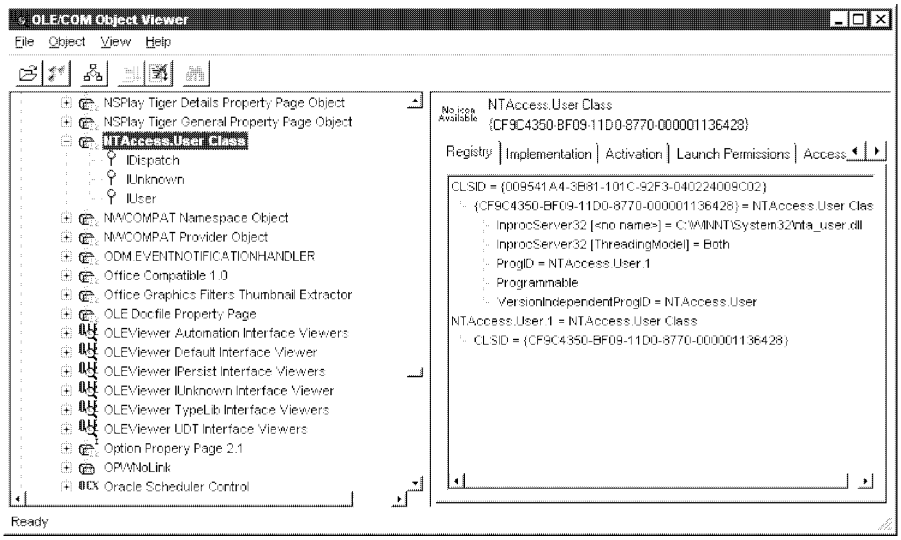
The OLE/COM Object Viewer installation installs the executable by default as \mstools\bin\oleview.exe. You use the Object Viewer to retrieve a COM object's Program ID as well as its methods and properties.

After have install a COM object, make sure you register it using the regsvr32.exe utility. Otherwise, you cannot 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.



```
NSPlay Tiger Details Property Page Object
NSPlay Tiger General Property Page Object
NTAccess.User Class
    IDispatch
    IUnknown
    IUser
NWCOMPAT Namespace Object
NWCOMPAT Provider Object
ODM.EVENTNOTIFICATIONHANDLER
Office Compatible 1.0
Office Graphics Filters Thumbnail Extractor
OLE Docfile Property Page
OLEViewer Automation Interface Viewers
OLEViewer Default Interface Viewer
OLEViewer IPersist Interface Viewers
OLEViewer IUnknown Interface Viewer
OLEViewer TypeLib Interface Viewers
OLEViewer UDT Interface Viewers
Option Propery Page 2.1
OPWNoLink
Oracle Scheduler Control
```

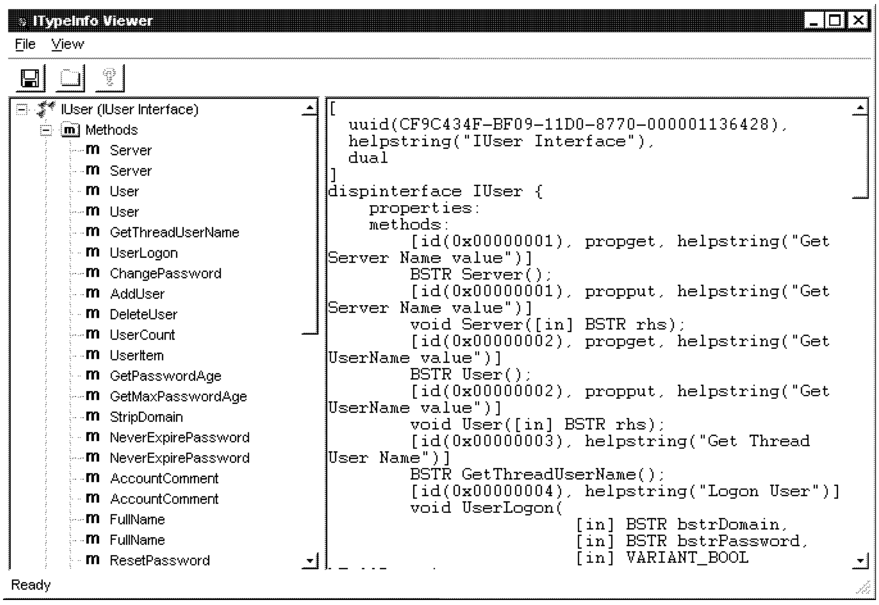
```
NTAccess.User Class
{CF9C4350-BF09-11D0-8770-000001136428}

CLSID = {009541A4-3B81-101C-92F3-040224009C02}
  {CF9C4350-BF09-11D0-8770-000001136428} = NTAccess.User Clas
    InprocServer32 [<no name>] = C:\WINNT\System32\nta_user.dll
    InprocServer32 [ThreadingModel] = Both
    ProgID = NTAccess.User.1
    Programmable
    VersionIndependentProgID = NTAccess.User
NTAccess.User.1 = NTAccess.User Class
  CLSID = {CF9C4350-BF09-11D0-8770-000001136428}
```

2 Select and expand the object in the Object Viewer.

3 Right-click the object to view it.

If you view the TypeInfo, you see the object's methods and properties. Some objects do not have access to the TypeInfo area. This is determined when an object is built and by the language used.



```
[
  uuid(CF9C434F-BF09-11D0-8770-000001136428),
  helpstring("IUser Interface"),
  dual
]
dispinterface IUser {
    properties:
    methods:
        [id(0x00000001), propget, helpstring("Get
Server Name value")]
        BSTR Server();
        [id(0x00000001), propput, helpstring("Get
Server Name value")]
        void Server([in] BSTR rhs);
        [id(0x00000002), propget, helpstring("Get
UserName value")]
        BSTR User();
        [id(0x00000002), propput, helpstring("Get
UserName value")]
        void User([in] BSTR rhs);
        [id(0x00000003), helpstring("Get Thread
User Name")]
        BSTR GetThreadUserName();
        [id(0x00000004), helpstring("Logon User")]
        void UserLogon(
                        [in] BSTR bstrDomain,
                        [in] BSTR bstrPassword,
                        [in] VARIANT_BOOL
```

## Creating and Using COM Objects

The following example uses `cfobject` to create the CDO (Collaborative Data Objects) for NTS NewMail object to send mail.

```
<cfobject type="COM"
    action="Create"
        name="Mailer"
        class="CDONTS.NewMail">
```

**Note**
CDO is installed by default on all Windows NT and 2000 operating systems that have installed the Microsoft SMTP server. In Windows NT Server environments, the SMTP server is part of the Option Pack 4 set up. In Windows 2000 Server and Workstation environments, it is bundled with the operating system. For more information on CDO for NTS, see http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/cdo/_olemsg_overview_of_cdo.htm.

You must create the component in ColdFusion before your application pages can invoke any methods or assign any properties in the component. This sample CDO for NTS NewMail component includes a number of methods and properties to perform a wide range of mail-handling tasks. (In the OLE/COM Viewer, methods and properties might be grouped together, so you might find it difficult to distinguish between them at first.)

The CDO for NTS NewMail object includes the following properties:

```
Body [ string ]
Cc  [ String ]
From  [ String ]
Importance  [ Long ]
Subject  [ String ]
To  [ String ]
```

You use these properties to define elements of the mail message you want to send. The CDO for NTS NewMail object also includes a method with a number of optional arguments to send messages:

```
Send()
```

## Connecting to COM objects

The `action` attribute of `cfobject` provides two ways to connect to COM objects:

- Create method (`cfobject action="Create"`) Takes a COM object, typically a DLL, and instantiates it prior to invoking methods and assigning properties.
- Connect method (`cfobject action="Connect"`) Links to an object that is already running on the server, typically an executable.

You can also use the `context` attribute to specify the object context. If you do not specify a context, ColdFusion uses the setting in the registry.

- `InProc` An in-process server object (typically a DLL) that is running in the same process space as the calling process, such as ColdFusion.

- `local` An out-of-process server object (typically an exe file) that is running outside the ColdFusion process space but running locally on the same server.
- `remote` An out-of-process server object (typically an exe file) that is running remotely on the network. If you specify `remote`, you must also use the `server` attribute to identify where the object resides.

## Setting properties and invoking methods

The following example, using the sample Mailer 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 provide method parameters and properties, such as the name of the recipient, the desired e-mail address, and so on.

```
<!--- First, create the object --->
<cfobject type="COM"
  action="Create"
  name="Mailer"
  class="CDONTS.NewMail">

<!--- 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.From = "#Form.fromName#">
<cfset Mailer.To = "#Form.to#">
<cfset Mailer.Subject = "#Form.subject#">
<cfset Mailer.Importance = 2>
<cfset Mailer.Body = "#Form.body#">
<cfset Mailer.Cc = "#Form.cc#">

<!--- Last, use the Send() method to send the message.
  Invoking the Send() method destroys the object.--->
<cfset Mailer.Send()>
```

**Note**
Use the `cftry` and `cfcatch` tags to handle exceptions thrown by COM objects. For more information on exception handling, see “Handling Exceptions in ColdFusion” on page 204.

## Getting Started with CORBA

The ColdFusion `cfobject` tag supports CORBA through the Dynamic Invocation Interface (DII). As with COM, the object's type information has to be available to ColdFusion. This requirement implies that an IIOP-compliant Interface Repository (IR) should be running on the network, and that the object's Interface Definition Language (IDL) specification must be registered in the IR.

ColdFusion 5 loads ORB runtime libraries dynamically using a connector. Macromedia provides connectors for some of the popular ORBs. Each of these connectors requires the ORB runtime libraries provided by the vendor. You must license the libraries from the appropriate vendor before deploying them. You manage the connectors on the CORBA Connectors page of the ColdFusion Administrator Server tab. Using this page, you can add a connector and specify the location of the ORB library. For more information, see *Installing and Configuring ColdFusion Server*.

## Calling CORBA Objects

In the `cfobject` tag, you must specify the following attributes when 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 `context` to `IOR` for a file containing the object's unique Interoperable Object Reference or to `NameService`.
- If you set the `context` attribute 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 make it local to the server or on the network in an accessible location.
- If you set the `context` attribute to `NameService`, the `class` attribute must include a period-delimited name, such as MyCompany.Department.Dev. Currently, ColdFusion can only resolve objects registered in a CORBA 3.0-compliant naming service. Make sure that the naming service (NS) is brought-up with a default naming context. 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 that your application uses to call the object's operations and attributes.

For the complete `cfobject` syntax, see the *CFML Reference*.

### Declaring structures and sequences

After you create the object, you can invoke attributes and operations on the object using the syntax outlined in the previous sections. ColdFusion also supports the use of complex types such as structures and sequences. For structures, use ColdFusion structures; for sequences, use ColdFusion arrays.

## Example

Here is 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);

};
```

Here is 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(NestedStruct)>
        <cfset temp=StructInsert(NestedStruct,"f",x)>
        <cfset temp=StructInsert(NestedStruct,"c", 'b')>
        <cfset temp=StructInsert(NestedStruct,"s", " Test")>
        </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

You can catch exceptions thrown by the CORBA object methods with the `cftry` and `cfcatch` tags. However, you cannot extract information from the exception object.

## Calling Java Objects

The `cfobject` tag can call any Java class that is available on the class path specified on the ColdFusion Administrator JVM and Java Settings page. For example:

```
<cfobject type="Java" class="MyClass" name="myObj">
```

Although this tag loads the class, it does not 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 is 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.

The following sections provide more details on calling Java objects in ColdFusion.

## Getting Started with Java

Java is a strongly typed language, unlike ColdFusion, which does not enforce data types. As a result, there are some subtle considerations when calling Java methods. The following sections create and use a Java class to illustrate how to use Java effectively in ColdFusion pages.

### Example: the Employee class

The Employee class has four data members: FirstName and LastName are public, and Salary and JobGrade are private. The Employee class has three overloaded constructors and a overloaded SetJobGrade method.

Save the following Java source code in the file Employee.java, compile it, and place the resulting Employee.class file in a directory that is specified in the class path.

```
public class Employee {

public String FirstName;
public String LastName;
private float Salary;
private int   JobGrade;

public Employee() {
  FirstName ="";
  LastName ="";
  Salary   = 0.0f;
  JobGrade = 0;
}

public Employee(String First, String Last) {
  FirstName = First;
  LastName = Last;
  Salary   = 0.0f;
  JobGrade = 0;
}

public Employee(String First, String Last, float salary, int grade) {
  FirstName = First;
  LastName = Last;
  Salary   = salary;
  JobGrade = grade;
}

public void SetSalary(float Dollars) {
   Salary = Dollars;
}

public float GetSalary() {
  return Salary;
}

public void SetJobGrade(int grade) {
  JobGrade = grade;
}

public void SetJobGrade(String Grade) {
  if (Grade.equals("CEO")) {
    JobGrade = 3;
  }
  if (Grade.equals("MANAGER")) {
    JobGrade = 2;
  }
  if (Grade.equals("DEVELOPER")) {
    JobGrade = 1;
  }
}

public int GetJobGrade() {
  return JobGrade;
}

}
```

## Example: CFML page that uses the Employee class

Save the following text as JEmployee.cfm:

```
<html>
<body>
<cfobject action=create type=java class=Employee name=emp>
<!-- <cfset void = emp.init()> -->
<cfset emp.firstname="john">
<cfset emp.lastname="doe">
<cfset firstname=emp.firstname>
<cfset lastname=emp.lastname>
</body>

<cfoutput>
  Employee name is  #firstname# #lastname#
</cfoutput>
</html>
```

When you view the page in your browser, you should get the following output:

Employee name is john doe

### Reviewing the code

The following table describes the CFML code and its function:

| Code | Description |
|---|---|
| `<cfobject action=create type=java class=Employee name=emp>` | Load an instance of the Employee Java class named emp. |
| `<!--- <cfset void=emp.init()> --->` | Do not call a constructor. ColdFusion invokes the default constructor when it first uses the class; in this case, when it processes the next line. |
| `<cfset emp.firstname="john">`<br>`<cfset emp.lastname="doe">` | Set the public fields in the emp object to your values. |
| `<cfset firstname=emp.firstname>`<br>`<cfset lastname=emp.lastname>` | Get the field values back from emp object. |
| `<cfoutput>`<br>`Employee name is  #firstname# #lastname#`<br>`</cfoutput>` | Display the retrieved values. |

## Java considerations

Keep the following points in mind when you write a ColdFusion page that uses a Java class object:

- The Java class name is case sensitive. You must make sure that both the Java code and the CFML code use Employee as the class name.

- Java method and field names are not case sensitive. Similarly, ColdFusion variables are not case sensitive. As a result, the sample code works even though the CFML uses emp.firstname and emp.lastname, while the Java source code uses FirstName and LastName for these fields.
- If you omit a call to the constructor (or, as in this example, comment it out) ColdFusion automatically invokes the default constructor when it first uses the class.

## Using an alternate constructor

The following CFML page explicitly calls one of the alternate constructors for the Employee object:

```
<html>
<body>

<cfobject action=create type=java class=Employee name=emp>
<cfset emp.init("John", "Doe",100000.00, 10 )>
<cfset firstname=emp.firstname>
<cfset lastname=emp.lastname>
<cfset salary=emp.GetSalary()>
<cfset grade=emp.GetJobGrade()>
</body>

<cfoutput>
  Employee name is  #firstname# #lastname#
  Employee salary and Job Grade #Salary# #grade#
</cfoutput>
</html>
```

In this example, the constructor takes four arguments; the first two are strings, and third is a float, and the fourth is an integer.

## Java and Cold Fusion Data Type Conversions

Cold Fusion is a typeless scripting language (that is, it does not use explicit data types) while Java is strongly typed.

Under most situations, when the method names are not ambiguous, ColdFusion can determine the required types. For example, ColdFusion strings are implicitly converted to the Java String type. Similarly, if a Java object contains a doIt method that expects a parameter of type int, and CFML is issuing a doIt call with a CFML variable x, ColdFusion will converts the variable x to Java int type. However, ambiguous situations can result from Java method overloading, where a class has multiple implementations of the same method that differ only in their parameter types.

The following sections describe how ColdFusion handles the unambiguous situations, and how it provides you with the tools to handle ambiguous ones.

## Default data type conversion

Whenever possible, ColdFusion matches Java types to ColdFusion types as listed in the following table. ColdFusion does not support direct conversion of Date/Time variables and structures.

| CFML | Java type |
|---|---|
| Character | String |
| Numeric | Int/long/float/double (depending on JavaCast) |
| Boolean | Bool |
| Array Of Character | Array of String |
| Array Of Numeric | Array of Java (int/long/float/double)<br>The conversion rule depends on the Java Method Signature (JavaCast does not help). |
| Array Of Java Objects | Array of Java Objects |

## Resolving ambiguous data types with the JavaCast function

You can overload Java methods so a class can have several identically named methods. At runtime, the VM resolves the specific method to use based on the parameters passed in the call and their types.

In the section "Example: the Employee class," on page 378, the Employee class has two implementations for the SetJobGrade method. One method takes a string variable, the other an integer. If you write code such as the following, which implementation to use is ambiguous:

```
<cfset  emp.SetJobGrade(“1”)>
```

The "1" could be interpreted as string or as number, so there is no way to know which method implementation to use. When a ColdFusion encounters such an ambiguity, it throws a user exception.

The ColdFusion `JavaCast` function helps you resolve such issues by specifying the Java type of a variable, as in the following line:

```
<cfset  emp.SetJobGrade(JavaCast(“int”, “1”)>
```

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

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

## Exception Handling

Use the `cftry` and `cfcatch` tags to can catch exceptions thrown by Java objects. Use the CFML `GetException` function to retrieve the Java exception object. The following example demonstrates the `GetException` function.

## Example: exception-throwing class

The following Java code defines the foo class that throws a sample exception. It also defines a fooException class that extends the Java built-in Exception class and includes a method for getting an error message.

```
public class foo {
  public foo() {
  }
  public void doException() throws fooException {
    throw new fooException("I am throwing a throw Exception ");
  }
}

Class fooException

public class fooException extends Exception {
  public String GetErrorMessage() {
    return "Error Message from fooException";
  }
}
```

## Example: CFML Java exception handling code

The following CFML code calls the foo class doException method. The `cfcatch` block handles the resulting exception by calling the CFML `GetException` function to retrieve the Java exception object and then calling the object's GetErrorMessage method to get the error information.

```
<cfobject action=create type=java class=foo name=Obj>
<cftry>
  <cfset VOID = Obj.doException() >
  <cfcatch type="Any">
    <cfset exception=GetException(Obj)>
    <cfset message=exception.GetErrorMessage()>
    <cfoutput>
      <br>The exception message is: #message#<br>
    </cfoutput>
  </cfcatch>
</cftry>
```

### Reviewing the code

The following table describes the code and its function:

| Code | Description |
|---|---|
| `<cfobject action=create type=java class=foo name=Obj>` | Load an instance of the Java foo class named Obj. |
| `<cftry>` `<cfset VOID = Obj.DoException() >` | Inside a cftry block. call the doException method of the Obj object. This method throws an exception of the fooException type. |

| Code | Description |
|---|---|
| `<cfcatch type="Any">` | Catch any exceptions and handle them in this block. |
| `<cfset exception=GetException(Obj)>`<br>`<cfset message=exception.GetERrorMessage()>` | Get the exception data by calling the CFML `GetException` function and passing it the Obj object. Set the message variable to the string returned by the exception object's `GetErrorMessage` method. |
| `<cfoutput>`<br>`  <br>The exception message is:`<br>`     #message#<br>`<br>`</cfoutput>` | Output the message string. |

Note that after you call GetException, the exception object is just like any other Java component object, and you can call any methods on it.

## The class loading mechanism

In ColdFusion prior to version 5, Java classes were loaded on demand and they were not unloaded until the server restarted. This is the way a typical Java application works and is appropriate for production systems. However, if you change a Java method implementation, you must shut down the server and restart it before ColdFusion can use the new class implementation.

In version 5, ColdFusion Server uses a custom class loader to load Java classes on the fly, similar to Java Servlet engines. This enables you to modify Java method implementations and use the new code from ColdFusion without restarting the server.

ColdFusion 5 introduces the concept of **dynamic load path**, a directory that you can specify on Cold Fusion Administrator JVM and Java Settings page. You deposit your Java class files in this directory when you update the class implementation. ColdFusion checks the class file time stamp when an object of the class is created. If ColdFusion detects a new class file, it loads the class from that directory.

To use this feature, make sure that the Java implementation classes that you modify are not in the general JVM class path. In addition, do not package the classes into jar files or zip files. In all other ways, ColdFusion Class loader follows Java conventions including those for package names and directory name mapping.

Suppose the directory “C:\classes” is the designated “hot” dynamic class path and you have a class com.Allaire.Employee. You put the Employee.class file in the following location:

C:\ classes\com\Allaire\Employee.class

The dynamic class-loading feature is meant for development uses, and incurs a slight performance penalty associated with checking time stamps during disk IO operations. For that reason, you do not use this feature in a production environment where there might be high volume of traffic.

To disable automatic class loading, put all classes in the normal Java class path. Classes located on the Java class path are loaded once per server lifetime and can only be reloaded by stopping and restarting ColdFusion Server.

## A more complex Java example

The following code provides a more complete example of using Java with `cfobject`. The Example class manipulates integer, float, array, Boolean, and Example object types.

### The Example class

The following Java code defines the Example class. The Java class Example has one public integer member, `mPublicInt`. Its constructor initializes `mPublicInt` to 0 or an integer argument. The class has the following public methods:

| Method | Description |
|---|---|
| ReverseString | Reverses the order of a string. |
| ReverseStringArray | Reverses the order of elements in an array of strings. |
| Add | Overloaded: Adds and returns two integers or floats or adds the `mPublicInt` members of two Example class objects and returns an Example class object. |
| SumArray | Returns the sum of the elements in an integer array. |
| SumObjArray | Adds the values of the `mPublicInt` members of an array of Example class objects and returns an Example class object. |
| ReverseArray | Reverses the order of an array of integers. |
| Flip | Switches a boolean value. |

```
public class Example {
  public   int      mPublicInt;

  public Example() {
      mPublicInt = 0;
  }

  public Example(int IntVal) {
      mPublicInt = IntVal;
  }

  public String ReverseString(String s) {
      StringBuffer buffer = new StringBuffer(s);
      return new String(buffer.reverse());
  }

  public String[] ReverseStringArray(String [] arr) {
    String[] ret = new String[arr.length];
    for (int i=0; i < arr.length; i++) {
      ret[arr.length-i-1]=arr[i];
    }
    return ret;
  }

  public int Add(int a, int b) {
      return (a+b);
  }

  public float Add(float a, float b) {
      return (a+b);
  }

  public Example Add(Example a, Example b) {
      return new Example(a.mPublicInt + b.mPublicInt);
  }

 static  public int SumArray(int[] arr) {
    int sum=0;
    for (int i=0; i < arr.length; i++) {
      sum += arr[i];
    }
    return sum;
  }

  static  public Example SumObjArray(Example[] arr) {
    Example sum= new Example();
    for (int i=0; i < arr.length; i++) {
      sum.mPublicInt += arr[i].mPublicInt;
    }
    return sum;
  }

  static public int[] ReverseArray(int[] arr) {
    int[] ret = new int[arr.length];
    for (int i=0; i < arr.length; i++) {
      ret[arr.length-i-1]=arr[i];
    }
    return ret;
  }


  static public boolean Flip(boolean val) {
      System.out.println("calling flipboolean");
      return val?false:true;
  }
}
```

## The useExample CFML Page

The following useExample.cfm page uses the `Example` class to manipulate numbers, strings, Booleans, and Example objects. Note the use of the `JavaCast` CFML function to ensure that CFML variables convert into the appropriate Java data types.

```
<html>
<head>
  <title>CFOBJECT and Java Example</title>
</head>
<body>

<!--- Create a reference to an Example object --->
<cfobject action=create type=java class=Example name=obj>
<!--- Create the object and initialize its public member to 5 --->
<cfset x=obj.init(JavaCast("int",5))>

<!--- Create an array and populate it with string values,
     then use the Java object to reverse them. --->
<cfset myarray=ArrayNew(1)>
<cfset myarray[1]="First">
<cfset myarray[2]="Second">
<cfset myarray[3]="Third">
<cfset ra=obj.ReverseStringArray(myarray)>

<!--- Display the results --->
<cfoutput>
  <br>
  original array element 1: #myarray[1]#<br>
  original array element 2: #myarray[2]#<br>
  original array element 3: #myarray[3]#<br>
  after reverse  element 1: #ra[1]#<br>
  after reverse  element 2: #ra[2]#<br>
  after reverse  element 3: #ra[3]#<br>
  <br>
</cfoutput>


<!--- Use the Java object to flip a Boolean value, reverse a string,
     add two integers, and add two float numbers --->
<cfset c=obj.Flip(true)>
<cfset StringVal=obj.ReverseString("This is a test")>
<cfset IntVal=obj.Add(JavaCast("int",20),JavaCast("int",30))>
<cfset FloatVal=obj.Add(JavaCast("float",2.56),JavaCast("float",3.51))>

<!--- Display the results --->
<cfoutput>
  <br>
  StringVal: #StringVal#<br>
  IntVal: #IntVal#<br>
  FloatVal: #FloatVal#<br>
  <br>
</cfoutput>
```

```
<!--- Create a two-element array, sum its values,
    and reverse its elements --->
<cfset intarray=ArrayNew(1)>
<cfset intarray[1]=1>
<cfset intarray[2]=2>
<cfset IntVal=obj.sumarray(intarray)>
<cfset reversedarray=obj.ReverseArray(intarray)>

<!--- Display the results --->
<cfoutput>
  <br>
  IntVal1 :#IntVal#<br>
  array1: #reversedarray[1]#<br>
  array2: #reversedarray[2]#<br>
  <br>
</cfoutput><br>


<!--- Create a ColdFusion array containing two Example objects.
    Use the SumObjArray method to add the objects in the array
    Get the public member of the resulting object--->
<cfset oa=ArrayNew(1)>
<cfobject action=create type=java class=Example name=obj1>
<cfset VOID=obj1.init(JavaCast("int",5))>
<cfobject action=create type=java class=Example name=obj2>
<cfset VOID=obj2.init(JavaCast("int",10))>
<cfset oa[1] = obj1>
<cfset oa[2] = obj2>
<cfset result = obj.SumObjArray(oa)>
<cfset intval = result.mPublicInt>

<!--- Display the results --->
<cfoutput>
  <br>
  intval1: #intval#<br>
  <br>
</cfoutput><br>
</body>
</html>
```

Chapter 21

# Building Custom CFXAPI Tags

Sometimes, the best approach is to develop elements of your application by building executables to run with ColdFusion. Perhaps the application requirements go beyond what is currently feasible in CFML. Perhaps you can improve application performance for certain types of processing.

To meet these types of requirements, you can use the ColdFusion Extension Application Programming Interface (CFXAPI) to access ColdFusion functions.

This chapter documents custom tag development using Java or C++, however, it is also possible to develop tags in Borland's Delphi.

## Contents

## What Are CFX Tags?

CFX tags are custom tags written against the ColdFusion Extension Application Programming Interface. Generally, you create a CFX if you want to do something that is not possible in CFML, or if you want to improve performance of a repetitive task. CFXs can do the following:

- Handle any number of custom attributes.
- Use and manipulate ColdFusion queries for custom formatting.
- Generate ColdFusion queries for interfacing with non-ODBC based information sources.
- Dynamically generate HTML to be returned to the client.
- Set variables within the ColdFusion application page from which they are called.
- Throw exceptions that result in standard ColdFusion error messages.

You can build CFXs using C++ or Java.

**Note**

To use a CFX, you must register it in the ColdFusion Administrator, as described in "Registering CFXs" on page 404.

# Before You Begin Developing CFX Tags in Java

While implementing CFX tags in Java is easy, you should consider the information in this section before you begin developing them.

## Sample Java CFXs

Before you begin developing a CFX tag in Java, you might want to study sample CFX tags. You can find the Java source files for the examples on Windows in the cfx\java\distrib\examples subdirectory of the main installation directory. On UNIX systems, the files are located in the cfx/java/examples directory. The example tags are as follows:

- `HelloColdFusion` Prints a personalized greeting. Demonstrates the minimal implementation required to create a CFX.
- `ZipBrowser` Retrieves the contents of a zip archive. Demonstrates how to generate a ColdFusion query and return 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 how to handle a ColdFusion query as input, throw exceptions, and generate 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, including the Java Development Kit (JDK) from Sun to build Java CFXs. You can download the JDK from Sun http://java.sun.com/j2se.

Although you can use the basic JDK, the recommended approach is to use one of the commercial Java IDEs that provide an integrated environment for development, debugging, project management, and access to documentation.

### Configuring the class path

To configure your development environment to build Java CFXs, you must ensure that the supporting classes are visible to your Java compiler. These classes are located in the cfx.jar archive, located in the Java/classes subdirectory of your ColdFusion installation directory. 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:

- 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.
- It supports a feature that allows Java CFXs located in the directory to be reloaded every time they are changed. Although this is not the default behavior for other Java classes, this behavior is very useful during an iterative development and testing cycle.

When you create new Java CFXs, you should develop and deploy them in 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 "Customizing and Configuring Java" for more details on customizing the class path.

## Customizing and Configuring Java

You use the JVM and Java Settings page on the ColdFusion Administrator Server tab to customize your Java development environment, such as by customizing the class path and Java system properties, or specifying an alternate Java Virtual Machine (JVM).

## Writing a Java CFX

To create a Java CFX, you create a class that implements the `CustomTag` interface. This interface contains one method, `processRequest`, which is passed `Request` and `Response` objects that are then used to do the work of the tag.

### To create a Java CFX:

1 Create a new source file in your editor.

2 Enter your code. The following code shows how t o create 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, use the following command line, which you execute from within the `classes` directory:

```
javac -classpath cfx.jar HelloColdFusion.java
```

**Note**
The previous command works only if the Java compiler (`javac.exe`) is in your path. If it is not in your path, specify the fully qualified path; for example, c:\jdk12\bin\javac on Windows NT or /usr/java/bin/javac on UNIX.

If you receive errors during compilation, check the source code to make sure you entered it correctly. If no errors occur, you just successfully wrote your first Java CFX.

As you can see, implementing the basic `CustomTag` interface is straightforward. This is all a Java class must 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 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.

This section provides an overview of these object types. For a complete example Java CFX that uses `Request`, `Response`, and `Query` objects, see the “ZipBrowser Example” on page 397.

## Request object

The `Request` object is passed to the `processRequest` method of the `CustomTag` interface. It provides methods for retrieving attributes, including queries, passed to the tag and for reading global tag settings.

| Method | Description |
|---|---|
| attributeExists | Checks whether the attribute was passed to this tag. |
| 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 whether the tag contains the `debug` attribute. |

## Response object

The `Response` object is passed to the `processRequest` method of the `CustomTag` interface. It provides methods for writing output, generating queries, and setting variables within the calling page.

| 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

The `Query` object provides an interface for working with ColdFusion queries. It includes methods for retrieving name, row count, and column names and methods for getting and setting data elements.

| Method | Description |
|---|---|
| `getName` | Retrieves the name of the query. |
| `getRowCount` | Retrieves the number of rows in the query. |
| `getColumns` | Retrieves the names of the query columns. |
| `getData` | Retrieves a data element from the query. |

| Method | Description |
|---|---|
| `addRows` | Adds a new row to the query. |
| `setData` | Sets a data element within the query. |

For detailed reference information on each of these interfaces, see the *CFML Reference*.

## Loading Java CFX classes

Each Java CFX class has its own associated `ClassLoader` that loads it and any dependent classes also located in the `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 you are attempting to perform casts on instances of classes loaded from a different `ClassLoader`. The cast fails even though the objects are apparently of the same type. This is because the object was created from a different `ClassLoader` and is therefore technically not the same type.

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

## Automatic class reloading

You can determine how the server treats changed Java CFX class files by specifying the `reload` attribute when you use a CFX tag in your CFML page. The following table describes the allowable values for the `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 but the CFX class file does not change. |
| `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 does not change. |
| `Never` | Never reload Java CFX classes. Load them once per server lifetime. |

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

**Note**

The `reload` attribute applies only to class files located in the `classes` directory. The ColdFusion server loads classes located on the Java class path once per server lifetime. You must stop and restart ColdFusion Server to reload these classes. For information on loading Java class files, see "The class loading mechanism" on page 384.

## 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. To store data and/or objects that are accessible to all instances of your `CustomTag`, use `static` data members. If you do so, you must ensure that all accesses to the data are thread-safe.

## Calling the CFX from a ColdFusion page

You call Java CFXs from within ColdFusion pages by using the name of the CFX that is registered on the ColdFusion Administrator CFX tags page. This name should be the prefix `cfx_` followed by the class name (without the .class extension). The following CFML page 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 preceding CFML code.

2 Save the file in a directory configured to serve ColdFusion pages. For example, you can save the file as C:\inetpub\wwwroot\cfdocs\testjavacfx.cfm on Windows NT or /home/docroot/cfdocs/testjavacfx.cfm on UNIX.

3 If you have not already done so, register the CFS in the ColdFusion Administrator, as described in "Registering CFXs" on page 404.

4 Request the page from your Web browser using the appropriate URL; for example:

```
http://localhost/cfdocs/testjavacfx.cfm
```

ColdFusion processes the page and returns a page that displays the text “Hello, 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 tag’s `archive` attribute specifies the fully qualified path of the zip archive to browse. The tag’s `name` attribute must specify the query to return to the calling page. The returned query contains three columns: Name, Size, and Compressed.

For example, to query an archive at the path C:\logfiles.zip for its contents and output the results, you use the following CFML code:

```
<cfx_ZipBrowser
      archive="C:\logfiles.zip"
      name="LogFiles" >

<cfoutput query="LogFiles">
#Name#,  #Size#, #Compressed# <BR>
</cfoutput>
```

The Java implementation of `ZipBrowser` is as follows:

```
import com.allaire.cfx.* ;
import java.util.Hashtable ;
import java.io.FileInputStream ;
import java.util.zip.* ;

public class ZipBrowser implements CustomTag
{
   public void processRequest( Request request, Response response )
      throws Exception
   {
      // validate that required attributes were passed
      if (  !request.attributeExists( "ARCHIVE" ) ||
            !request.attributeExists( "NAME" ) )
      {
         throw new Exception(
            "Missing attribute (ARCHIVE and NAME are both " +
            "required attributes for this tag)" ) ;
      }
   // get attribute values
      String strArchive = request.getAttribute( "ARCHIVE" ) ;
      String strName = request.getAttribute( "NAME" ) ;

  // create a query to use for returning the list of files
      String[] columns = { "Name", "Size", "Compressed" } ;
      int iName = 1, iSize = 2, iCompressed = 3 ;
      Query files = response.addQuery( strName, columns ) ;
```

```
         // read the zip file and build a query from its contents
            ZipInputStream zin =
               new ZipInputStream( new FileInputStream(strArchive) ) ;
            ZipEntry entry ;
            while ( ( entry = zin.getNextEntry()) != null )
            {
               // add a row to the results
               int iRow = files.addRow() ;

               // populate the row with data
               files.setData( iRow, iName,
                  entry.getName() ) ;
               files.setData( iRow, iSize,
                  String.valueOf(entry.getSize()) ) ;
               files.setData( iRow, iCompressed,
                  String.valueOf(entry.getCompressedSize()) ) ;

               // finish up with entry
               zin.closeEntry() ;
            }

            // close the archive
            zin.close() ;
         }
      }
```

## Approaches to Debugging Java 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 you cannot use an interactive debugger to debug Java classes that have been loaded by another process.

To overcome this limitation, you can use one of two techniques:

- Debug the CFX while it is running within ColdFusion Server by outputting debug information as needed.
- Debug the request in an interactive debugger offline from ColdFusion Server using the special `com.allaire.cfx` debugging classes.

### 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 develop for 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 outputting debug text using the `Response.write` method, you can also call your Java CFX tag with the `debug="On"` attribute. 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, use the following CFML code:

```
<cfx_HelloColdFusion" name="Robert" debug="On">
```

To determine whether a CFX is invoked with the `debug` attribute, use the `Request.debug` method. To write debug output in a special debug block after the tag finishes executing, use the `Response.writeDebug` method. For details on using these methods, see the *CFML Reference*.

## 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. You use this method 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.

After you implement a `main` method as described previously, 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...
   }
}
```

# Developing CFX Tags in C++

The following sections provide some background to help you develop CFX tags in C++.

## Sample C++ CFXs

Before you begin development of a CFX tag in C++, you might want to study the two CFX tags that are included to give you additional insight into working with the CFXAPI. The two example tags are as follows:

- `CFX_DIRECTORYLIST` Queries a directory for the list of files it contains.
- `CFX_NTUSERDB` (Windows NT only) Allows addition and deletion of NT users.

On Windows NT, these tags are located in the \cfusion\cfx\examples directory. On UNIX, look in /<installdirectory>/coldfusion/cfx/examples.

## Setting up your C++ development environment

The following compliers generate valid CFX code for UNIX platforms:

| **Platform** | **Compiler** |
|---|---|
| Solaris | Sun C++ compiler 5.0 or higher (gcc does not work) |
| Linux | RedHat 6.2 gcc/egcs 1.1.2 compiler |
| HPUX 11 | HP aCC C++ compiler |

Before you can use your C++ compiler to build custom tags, you must enable the compiler to locate the CFXAPI header file, `cfx.h`. On Windows NT, you do this by adding the CFXAPI Include directory to your list of global include paths. On Windows, this directory is \cfusion\cfx\include. On UNIX it is /opt/coldfusion/cfx/include. On UNIX, you will need `-I` *`<includepath>`* on your compile line (see the Makefile for the directory list example in the cfx/examples directory).

## 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, you must install the CFXAPI Tag Development Kit (it is installed 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**, and 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 (Optional) 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 404.

## Compiling C++ CFXs

CFX tags built on Windows NT and UNIX must be thread safe. Compile CFXs for Solaris with the `-mt` switch on the Sun compiler.

## 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. For a detailed description of the CFXAPI classes and members, see the see *CFML Reference*.

## Debugging C++ CFXs

After you configure a debug session, you can run your custom tag from within the debugger, set breakpoints, single-step, and so on.

### On Windows NT

You can easily debug custom tags within the Visual C++ environment. To debug a tag, open the Build Settings dialog box 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

Use the following debuggers and settings for the supported UNIX operating systems:

| OS | Debugger | Other OS-specific requirements |
|---|---|---|
| Solaris | dbx | The environment variables must include `LD_LIBRARY_PATH` and `CFHOME`.<br>Use “stop in main” to set a breakpoint in main. |
| Linux | gdb | The environment variables must include `LD_LIBRARY_PATH` and `CFHOME`.<br>Use “break main” to set a breakpoint in main. |
| HP-UX | DDE | The environment variables must include `SHLIB_PATH` and `CFHOME`.<br>Use “stop in main” to set a breakpoint in main. |

Shut down ColdFusion using the stop script. Set the environment variables as they are set in the start script. You can then run the cfserver executable under the dbx debugger and set breakpoints in your CFX code. You might need to set a breakpoint in `main` so the debugger loads the symbols for your CFX before you can set breakpoints in your code.

## 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 a Java CFX:

1. On the ColdFusion Administrator Server tab, select **Extensions > CFX Tags** to opent the CFX Tags page.
2. Click the Register Java CFX button.
3. Enter the tag name (for example `cfx_MyNewTag`).
4. Enter the class name (without the .class extension).
5. (Optional) Enter a description.
6. Click Submit Changes.

You can now call the tag from a ColdFusion page.

### To register a C++ CFX:

1. On the ColdFusion Administrator Server tab, select **Extensions > CFX Tags** to opent the CFX Tags page.
2. Click the Register C++ CFX button.
3. Enter the Tag name (for example `cfx_MyNewTag`).
4. If the Server Library .dll field is empty, enter the file path.
5. Accept the default Procedure entry.
6. Clear the Keep library loaded box while developing the tag.

   When the tag is ready for production use, you can select this option to keep the DLL in memory for improved performance.
7. (Optional), Enter a description.
8. Click Submit Changes.

You can now call the tag from a ColdFusion page.

**On Windows NT only**, the Visual C++ Custom Tag Wizard automatically registers custom tags so that they can be tested and debugged.

### To change a CFX tag:

1. Click the tag that you want to change in the Registered CFX Tags list.
2. Make changes as needed on the Edit CFX Tag page.
3. Click Submit Changes.

### To delete a CFX tag:

- Click the Delete Applet (right-most) icon in the Controls column of the Registered CFX Tags list for the tag you want to delete.

On **Windows NT only,** the Visual C++ Custom Tag Wizard automatically registers custom tags so that they can be tested and debugged.

## Distributing CFX Tags

If you are distributing a custom tag, you can automatically register it during the setup process by writing the registration entries directly into the registry. The following table lists the registry entries.

The following table lists the registry entries for Java:

| 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. |

The following table lists the registry entries for C++:

| 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 (UNIX) that implements the custom tag. |
| `ProcedureName` | The name of the procedure to call for processing tag requests. |
| `Description` | A description of the tag's functionality for browsing by end users. |
| `CacheLibrary` | Indicates whether to keep the DLL or shared object loaded in RAM (1 or 0). |

You can create a file containing this information by using the Windows 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. On UNIX you must edit the registry data file, located in /opt/coldfusion/registry/cf.registry.

### 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\CurrentVer
             sion\ 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
   ```

### 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\CurrentVer
             sion\ 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
   ```

# Index