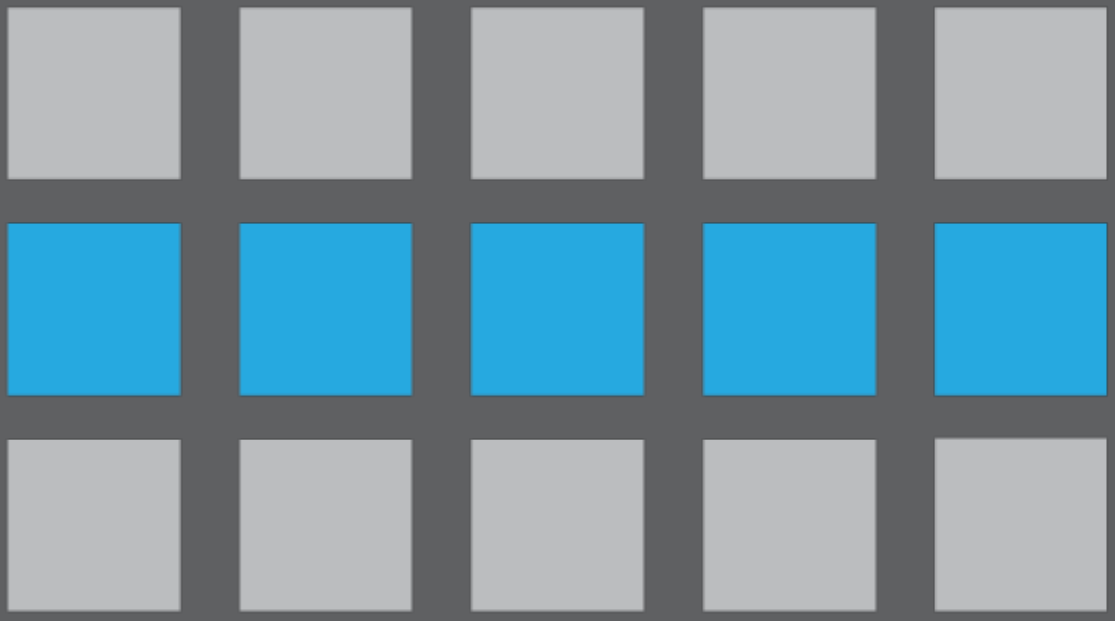
LEARN CF IN A WEEK



Week 1

www.learncfinaweek.com

Acknowledgements

This course would not have been possible without the following people:

Authors

- Emily Christiansen
- Tim Cunningham
- David Epler
- Sam Farmer
- Dave Ferguson
- Simon Free
- Paul Hastings
- Guust Nieuwenhuis
- Dan Skaggs
- Nic Tunney
- Adam Tuttle
- Dan Wilson

Editors

- Mark Esher
- Kristin Ferguson
- Tiffany Goebel

Designers

- Nick Borden
- Jim Priest

Copyright

Contents

Installation

To be able to follow along with the Hands On portion of the course you will need to install ColdFusion, MySQL and the sample files. The following instructions show you how to install the software as well as the sample files.

Installing ColdFusion

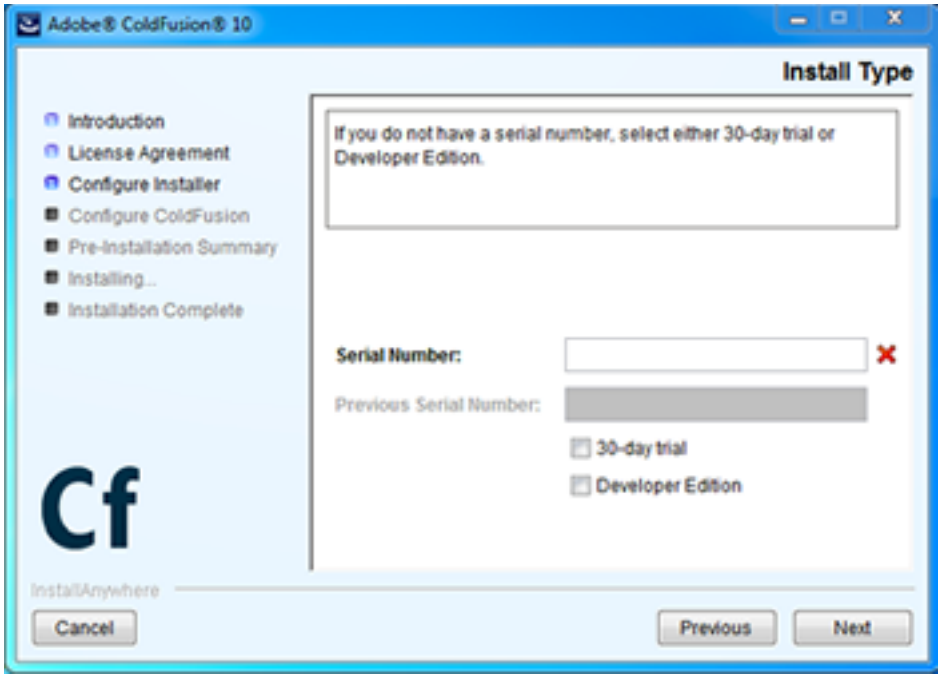
To be able to program in ColdFusion, a ColdFusion server needs to be installed. There are a couple of options available, but the one that we are going to focus on is a local development server.

A local development server is free and allows you to develop ColdFusion applications that use all of ColdFusion's available features. There are, however, a few limitations, such as not being able to use the server as an external web server. That being said, there are additional benefits to using a local ColdFusion development server, such as not needing to have IIS or Apache installed, but instead using the packaged web server.

To install ColdFusion, follow the steps below:

Windows

1. Open up a web browser and go to: http://www.adobe.com/go/trycoldfusion/
2. Sign in with your Adobe ID. If you do not have an Adobe account, click on the "Create an Adobe account" button on the left.
3. Select the appropriate version of ColdFusion you wish to download. If you have a 64 bit system select 'English | Windows 64 bit', else select 'English | Windows'.
4. Click "Download".
5. Save the file to your desktop.
6. Once downloaded, double click the file.
7. Click "Next".
8. Accept the terms and click "Next".
9. When you see the screen below, select "Developer Edition" and click "Next".



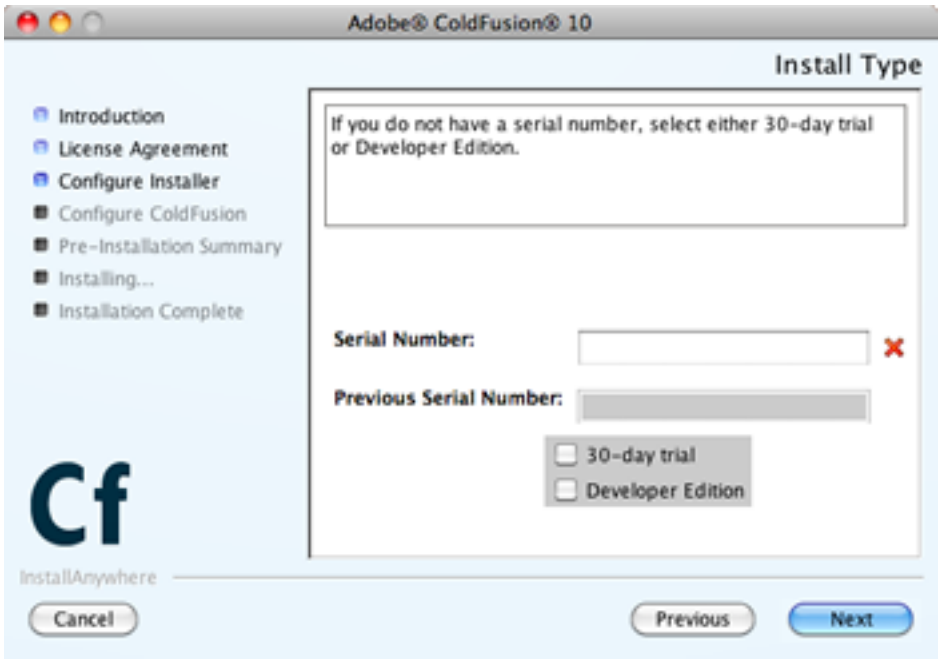
10. Select Server configuration and click "Next".
11. Leave all checkboxes selected and click "Next".
12. Leave the "Enable Secure Profile" checkbox unchecked and click "Next".
13. Provide a password and click "Next". Make sure you remember the password!
14. Leave the location at `C:\ColdFusion10\` and click "Next". If you want to install in a different location you can, but it is important to note that the install instructions will always reference `C:\ColdFusion10\`.
15. Select 'Built-in web server (Development use only)' and click "Next".
16. Provide a password and click "Next". Make sure you remember the password; you will need this later on in the install process.
17. Select 'Enable RDS', provide a password and click "Next". Make sure you remember the password!
18. Leave the 'Automatically check for server updates' selected and click "Next".
19. Click "Install".
20. If you receive any Windows firewall messages, click 'Allow'.
21. When the installation wizard is done, keep the 'Launch the Configuration Wizard in the default browser' selected and click "Done".
22. A browser window will open. Enter your ColdFusion Administrator Password and click "Login". (This is the password from step 16)
23. Once you receive the 'Setup Complete' screen, click the "OK" button.
24. You will now see the ColdFusion Administrator screen. To access this screen at a later date, you

can go to: http://localhost:8500/CFIDE/administrator/index.cfm

25. You have now successfully installed ColdFusion 10. To access the web root, you can go to: http://localhost:8500/. he web root on the file system is `C:\ColdFusion10\cfusion\wwwroot\`.

Mac

1. Open up a web browser and go to: http://www.adobe.com/go/trycoldfusion/
2. Sign in with your Adobe ID. If you do not have an Adobe account, click on the "Create an Adobe account" button on the left.
3. Select the English | MAC OS X version of ColdFusion.
4. Click "Download".
5. Save the file to your desktop.
6. Once downloaded, double click the file.
7. Click "Next".
8. Accept the terms and click "Next".
9. When you see the screen below, select "Developer Edition" and click "Next".



10. Select Server configuration and click "Next".
11. Leave all checkboxes selected and click "Next".
12. Leave the "Enable Secure Profile" checkbox unchecked and click "Next".

13. Provide a password and click "Next". Make sure you remember the password!
14. Leave the location at `/Applications/ColdFusion10/` and click "Next". If you want to install in a different location you can, but it is important to note that the install instructions will always reference `/Applications/ColdFusion10/`.
15. Select 'Built-in web server (Development use only)' and click "Next".
16. Provide a password and click "Next". Make sure you remember the password; you will need this later on in the install process.
17. Select 'Enable RDS', provide a password and click "Next". Make sure you remember the password!
18. Leave the 'Automatically check for server updates' selected and click "Next".
19. Click "Install".
20. When the installation wizard is done, keep the 'Launch the Configuration Wizard in the default browser' selected and click "Done".
21. A browser window will open. Enter your ColdFusion Administrator Password and click "Login". (This is the password from step 16)
22. Once you receive the 'Setup Complete' screen, click the "OK" button.
23. You will now see the ColdFusion Administrator screen. To access this screen at a later date, you can go to: http://localhost/CFIDE/administrator/index.cfm
24. You have now successfully installed ColdFusion 10. To access the web root, you can go to: http://localhost:8500/. The web root on the file system is `/Applications/ColdFusion10/cfusion/wwwroot/`

Installing MySQL

ColdFusion has the ability to communicate with a number of different databases, which will be covered later on in this course; for the sample application we will be working on throughout the course, we will be using MySQL. If you already have MySQL 4 or 5 already installed, you can proceed to the 'Install Sample Files' section. If not, follow the steps below:

Windows

1. Open up a browser and go to: http://dev.mysql.com/downloads/mysql/
2. Scroll to the list of available downloads. Click the 'Download' button next to the applicable download.
   - Windows 64 bit - Windows (x86, 64-bit), MSI
   - Windows 32 bit - Windows (x86, 32-bit), MSI
     If you are unsure, assume you are on a 32 bit machine.
3. To begin the download, you must either login using pre-existing credentials by clicking the 'Proceed' button under the New Users section or click the 'No thanks, just start my download' link at the bottom of the screen.
4. Save the file to your desktop.
5. Double click the file.
6. On the Welcome screen click 'Next'.
7. Accept the terms in the license agreement and click 'Next'.
8. When presented with a list of available Setup Types, Select 'Typical'.
9. When ready to install, click 'Install'.
10. If a system message pops up asking if you want the software to be installed on your machine, click 'Yes'.
11. If a MySQL enterprise pop up window appears, click 'Next' until it disappears.
12. When the Setup Wizard has completed, Click 'Finish'.
13. If you receive another pop up message asking if you want the software to be installed on your machine, click 'Yes'.
14. When the Server Instance Configuration Wizard displays, Click 'Next'.
15. When displayed the available Server Instance Configuration types select 'Standard Configuration' and click 'Next'.
16. When on the Windows Options scree, keep 'Install as a Service' selected as well as 'Launch the MySQL Server automatically'. Select the 'Include Bin Directory in Windows PATH' option and select 'Next'.

17. On the Security Options scree, specify a new root password and select 'Next'. Remember this password as it will be needed later!
18. When the Ready to Execute screen displays, click 'Execute'.
19. After all the Configuration steps have successfully run, Click 'Finish'.
20. You have successfully installed MySQL.

Mac

1. Open up a browser and go to: http://dev.mysql.com/downloads/mysql/
2. Scroll to the list of available downloads. Click the 'Download' button next to the applicable download.
    - Mac 64 bit - Mac OS X ver. 10.6 (x86, 64-bit), DMG Archive
    - Mac 32 bit - Mac OS X ver. 10.6 (x86, 32-bit), DMG Archive

      If you are unsure, assume you are on a 32 bit machine
3. To begin the download, you must either login using pre-existing credentials by clicking the 'Proceed' button under the New Users section or click the 'No thanks, just start my download' link at the bottom of the screen.
4. Save the file to your desktop.
5. Double click the file.
6. Once the DMG has mounted, double click the mysql-5.5.28-osx10.6-x86_64.pkg (or similar) file. It should be the first file in the list.
7. On the welcome screen, click 'Continue'.
8. Review the Important Information screen and click 'Continue'.
9. Review the Software License Agreement and click 'Continue'.
10. Agree to the terms.
11. Select the drive you wish to install the software to and click 'Continue'.
12. Click 'Install'.
13. Provide your system password and click 'OK'.
14. Once you receive the screen stating the Installation was Successful, click 'Close'.
15. Double click the MySQL.prefPane icon.
16. Click 'Install'.
17. If the MySQL server is not running, click the 'Start MySQL' Server button.

18. If prompted, provide your system password and click 'OK'.
19. Click the 'Automatically Start MySQL Server' on Startup check box.
20. Close the window.
21. Open up a terminal window and enter the following:
    ```
    /usr/local/mysql/bin/mysqladmin -u root password [NewPassword]
    ```
22. Make sure to replace [NewPassword] with the password you wish to use for the root user. Remember this password, it will be needed later.
23. Close the Terminal window.
24. You have successfully installed MySQL and set the root user password.

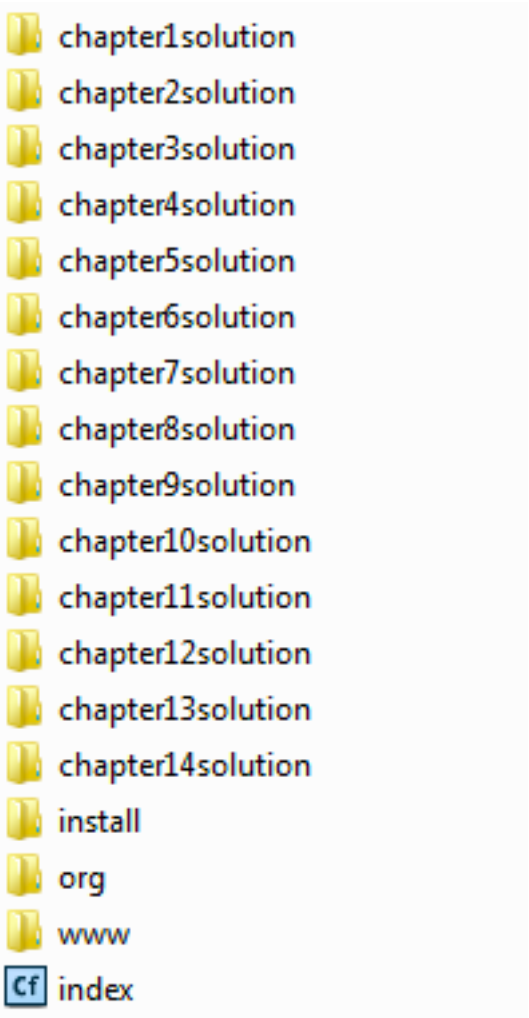
Installing Sample Files

As well as having the ability to read about ColdFusion, 'Learn CF in a Week' has a Hands On section of the course, giving you the opportunity to create your own ColdFusion web site. During the course, you will take a basic HTML website and add ColdFusion to it, creating a fully functional ColdFusion application.

To be able to take part in the Hands On, you must first install the necessary Application files. To do this, follow the steps below:

Windows

1. Download the sample application files from the download link in the header of http://www.learncfinaweek.com/.
2. Create a folder in your web root called 'learncfinaweek'. Your webroot is located at: `C:\ColdFusion10\cfusion\wwwroot\`
3. Unzip the sample application files into the `learncfinaweek` folder. The contents of the folder should look like this:



4. To confirm you have the files in the correct place, go to: http://localhost:8500/learncfinaweek/.

5. Now that you have the files in the correct place, you now need to run an install script that will populate your database and set up ColdFusion with the necessary information.
   Go to: http://localhost:8500/learncfinaweek/install/.

6. Fill out the form using the passwords from the previous installation processes and click on "Install".

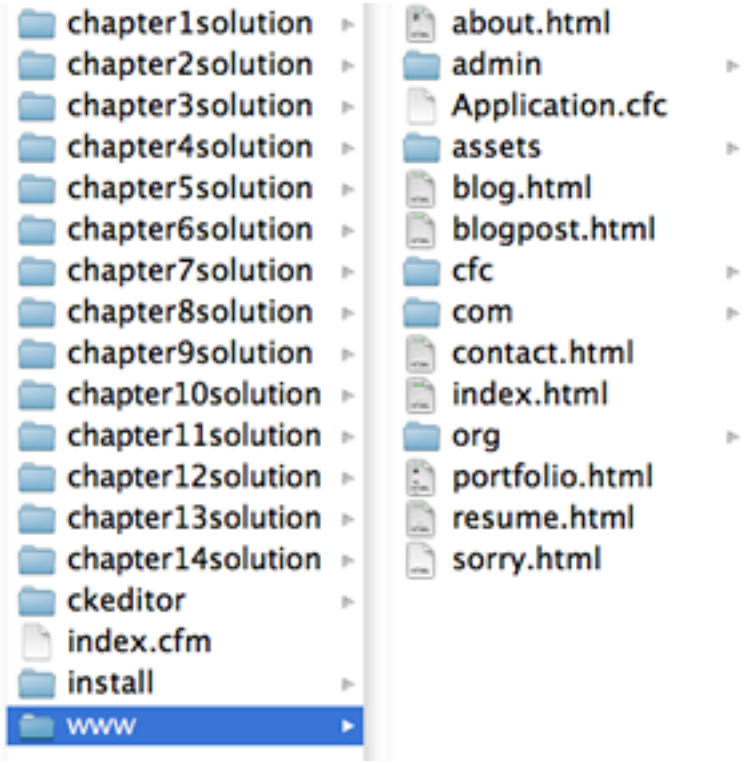
7. If any problems occur, follow the onscreen instructions on how to correct them. If no problems exist, you are good to start the course.

Mac

1. Download the sample application files from the download link in the header of http://www.learncfinaweek.com/.

2. Create a folder in your web root called learncfinaweek. Your webroot is located at `/Applications/ColdFusion10/cfusion/wwwroot/`

3. Unzip the sample application files into the learncfinaweek folder. The contents of the folder should look like this:



4. To confirm you have the files in the correct place go to: http://localhost:8500/learncfinaweek/.

5. Now that you have the files in the correct place you now need to run an install script that will populate your database and set up ColdFusion with the necessary information.
   Go to: http://localhost:8500/learncfinaweek/install/.

6. Fill out the form using the passwords from the previous installation processes and click on install.

7. If any problems occur follow the onscreen instructions on how to correct them. If no problems exist then you are good to start the course.

How the Hands On Works

Throughout this course you will see a number of hands on pages. These pages give you step by step instructions on what to do to your ColdFusion code. The root of the project is located at: `C:\ColdFusion10\cfusion\wwwroot\learncfinaweek\www\` for Windows and `/Applications/ColdFusion10/cfusion/wwwroot/learncfinaweek/www/` for Macs.

Inside the 'learncfinaweek' folder, you will see a number of folders that relate to the chapters, such as chapter1Solution, chapter2Solution, and so on. Inside these folders, you will find copies of the code which have all the hands ons completed up until and including that chapter. If at any point you get stuck with a code problem, you can look at these files to see what the solution is. If you are still unable to find what is causing the problem in your code, you can copy the files and folders from the chapter folder and place them in your 'www' folder. Once you have copied them in, you can continue with the next chapter..

Once you have completed the hands on problems, you will have a fully functioning ColdFusion web site that is ready to be launched right away. Included in this application is file manipulation, database calls, remote calls, sending emails, ORM, and a fully secured Admin area.

1

The Basics

By Dan Wilson

About Dan Wilson

As principal partner of DataCurl LLC, Dan Wilson runs both the consulting practice and ChallengeWave.com, a way to help employees start and stick with healthier lifestyles.
Before launching DataCurl, Dan held numerous senior program and development positions in such industries as Technical Consulting, Health Care, Online Publishing and Government Contracting.

Dan is an avid participant in technology communities; an Adobe Community Professional, manager of the Triangle ColdFusion User Group in Research Triangle Park, North Carolina, Managing Director of the popular Model-Glue framework and contributor to numerous open source projects based on ColdFusion, Flex and AIR platforms.

Dan presents on ColdFusion, Flex and Rapid Development Techniques at popular conferences around the world. You can find his thoughts on ColdFusion, Flex, AIR and other technology matters at http://www.nodans.com and some occasional ramblings on food at http://blog.chefdanwilson.com.

When not in front of a computer, you can find him biking, hiking, surfing, playing volleyball and helping small businesses improve their sales and marketing.

What is ColdFusion?

ColdFusion is a rapid development platform for building modern web applications. ColdFusion is designed to be expressive and powerful. The expressive characteristic allows you to perform programming tasks at a higher level than most other languages. The powerful characteristic gives you integrations with functionality important to web applications like database access, MS Exchange access, PDF form creation and more.

The ColdFusion platform is built on Java and uses the Apache Tomcat J2EE container. While you have full access to Java and Tomcat, you need not worry about these details. You'll interact with ColdFusion and the user friendly ColdFusion Mark-up Language (CFML) to write your programs. Your ColdFusion files will use the file extension '.cfc' for objects and '.cfm' for pages. CFML requires much less ceremony and infrastructure than typical java while offering a significantly faster development experience than Java.

After taking this CF in a Week series, you'll have the basics necessary to begin making dynamic web sites, building intranet applications, or even working on the next Facebook competitor!

Setting Variables

Programming is all about doing stuff to things. Generally, the stuff is the execution of a process or algorithm and the things are information or data.

An algorithm is just a fancy name for a series of steps, like tying your shoelaces. Information or data is just values. Your name is a piece of information and so is your birth date. Since it's shorter, in this article we'll use the word data to describe a piece of information.

In ColdFusion, data is held in variables. Think of a variable like a mailbox. You can stuff things like letters and packages into a mailbox and get them out later to do stuff. ColdFusion makes storing information very easy because it's a loosely typed Language. You can stick any kind of data into a ColdFusion variable without having to tell ColdFusion what kind of data it is.

Here's the simplest code to set a variable:

```
<cfset ThisIs = "fun" />
```

Now, any time I refer to **ThisIs** in my ColdFusion code, it'll hold the data "fun". You can look at the contents of a variable in ColdFusion by using the `cfdump` tag.

```
<cfdump var="#ThisIs#" />
```

See how the variable contains the string "**fun**"?

A few things to note. Examine the statement again:

```
<cfset ThisIs = "fun" />
```

The part of the statement to the left (**ThisIs**) of the equals sign is the variable name. The part of the statement to the right of the equals sign ("**fun**") is the value to assign to the variable.

Now, in the second statement, you'll note we surrounded the variable name with pound # or hash symbols. The pound sign tells ColdFusion to evaluate the contents. In the below example, the pound sign will tell ColdFusion to replace the ThisIs string with the contents of the previously defined variable ThisIs.

```
<cfdump var = "#ThisIs#" />
```

Here's another way to look at pound sign use. Examine the below three statements. Can you guess what will happen in each case?

```
<cfdump var = "1 + 2" /><br />
<cfdump var = "#1 + 2#" /><br />
<cfdump var = "1 + 2 IS #1 + 2#" /><br />
```

In the first statement, our math problem is surrounded by the Double Quote character. This tells ColdFusion to treat the statement as a string and do no evaluations on it.

In the second statement, we also surrounded the math problem with the pound sign. ColdFusion executes the statement and performs our calculation for us.

In the third statement, notice we've mixed strings and evaluations through the correct use of the pound sign and the double quotes.

The Left Side Of The Statement: <cfset ThisIs

The left side of the statement is the variable name. You can have numbers as part of the variable name, but the variable name must start with a letter. You must not have any spaces in your variable names. If you need to use more words for your variable name, you can simply write the variable with **CamelCase** letters, or use Under_Scores to separate each word if you want. The choice is one of style, there are no right answers. Also, most special characters are not allowed in variable names. A good rule to follow is variable names should be descriptive and help provide context to what is being done.

For example, the word **variable** is a bad naming choice because the name adds no context to why the variable exists:

```
<cfset variable = "12/26/1975" />
```

The variable name **UserBirthdate** is a good naming choice because it adds context to why the variable exists:

```
<cfset UserBirthdate = "12/26/1975" />
```

> **Pro Tip:** You'll understand the most about WHY your program is written a certain way as you are writing it. Take advantage of that hard-earned understanding and leave yourself (or others) as many clues and as much context as possible. Later on, after you've forgotten some of the details, it'll be much easier to piece together why the program was written a certain way and it'll be quicker to make good updates to your program.

The Right Side Of The Statement: "fun">

The right side of the statement contains the value for the variable. Simple strings are enclosed in single or double quotes, with double quotes being the most common.

The right side of the statement is an execution zone. This means ColdFusion will attempt to evaluate items on the right side of the statement.

Examples

Here are a few examples that will help you understand setting ColdFusion variables:

Print Out the Current Date:

```
<cfset DateToday = now() />
<cfdump var = "#DateToday#" />
```

See how the ColdFusion function `now()`, which gets the servers current date and time, was evaluated and the contents placed in the variable DateToday? Anything not specifically in quotes (double or single) will be evaluated. It is possible to evaluate items in quotes, if you use the # sign. Keep this in mind as you develop ColdFusion. Novices have a tendency to overuse the pound sign.

Alternate Method: Print Out the Current Date:

This code will evaluate to the same thing in the previous step.

```
<cfset DateToday = "#now()#" />
<cfdump var = "#DateToday#" />
```

However, unless there are strings in the quotes, it's generally preferred to use the previous method.

To mix execution and strings, do this:

Mix Execution and Strings

```
<cfset DateToday = "Today is: #now()#" />
<cfdump var = "#DateToday#" />
```

Concatenation

You could also concatenate the strings using the & operator. This example is functionally equal to the previous example:

```
<cfset DateToday = "Today is: " & now() />
<cfdump var = "#DateToday#" >
```

So is this one:

```
<cfset DateToday = "Today is: " />
<cfset DateToday = DateToday & now() />
<cfdump var = "#DateToday#" />
```

See how we added DateToday to another evaluation and replaced the DateToday variable name with the new contents?

Outputting a Variable

You will often need to output the contents of the variables you create. One reason is to display the contents of a variable to the user, perhaps to display the username on a web page. Another reason is to verify the contents of a variable while you are in the process of writing or debugging your program.

About cfoutput: To display the contents of a variable to a user, use `cfoutput`. The variable reference must be a simple value that can be displayed as text. This includes Strings, Numbers, Dates, Times, and so on. Complex variables, such as Structs, Arrays, Queries, Functions, and so on, can not be displayed with the `cfoutput` command because they are not displayable as text.

Example of cfoutput Usage

```
<cfset DateToday = "Today is: #now()#" />
<cfoutput>#DateToday#</cfoutput>
```

The variable will be evaluated, and the variable contents will be added to the current end of the response buffer. i.e. it will be displayed to the user.

About cfdump

To inspect or verify the contents of a variable while writing or debugging your program, use `cfdump`. ColdFusion makes it very easy to see the contents of ANY variable by using the `cfdump` command. `cfdump` will convert the variable contents to a string representation and format it for easy viewing. The `cfdump` command can be used to help debug your program.

Example of cfdump Usage

```
<cfset DateToday = "Today is: #now()#" />


<cfdump var = "#DateToday#" />


<cfset DateArray = [dateFormat(now(), "short"), dateFormat(dateadd('d',1
,now()), "short"), dateFormat(dateadd('d',2,now()), "short")] />
```

```
<cfdump var = "#DateArray#" />


<cfset DateStruct = { today=dateFormat(now(), "short"), tomorrow=dateF
ormat(dateadd('d',1,now()), "short"), later=dateFormat(dateadd('d',2,n
ow()), "short") } />


<cfdump var = "#DateStruct#" />
```

Data Types

In our last Chapter, we talked about variables, data, and how to transport data around in your application. Different data types serve different purposes.

Strings/Numbers

Is Simple: Yes

Strings and numbers are very easy to work with. To set a string or a number, use the `cfset` command. To append strings and numbers to each other, use the & operator:

```
<cfset aString = "hi" />
<cfset aNumber = 42 />
<cfset aStringAndANumber = aString & aNumber />

aString: <cfoutput>#aString#</cfoutput>
aNumber: <cfoutput>#aNumber#</cfoutput>
aStringAndANumber: <cfoutput>#aStringAndANumber#</cfoutput>
```

If you have a big block of strings to set, you can use the `cfsavecontent` command.

```
<cfsavecontent variable="EmailContent">
    Hi

    We want to send you a hoverboard.
    Let us know if you will accept this free offer.

    -Us
</cfsavecontent>

<cfoutput>#EmailContent#</cfoutput>
```

Dates

Is Simple: Kind of

Dates are also very easy to work with in ColdFusion. You can use built in functions like `now()` to make a date, or you can type the date into the variable assignment like this:

```
<cfset DateToday = now() />
<cfset NewYearDay = "1/1/2013" />
```

You can use built-in functions to work with dates. To show how many days it has been since the turn of the century:

```
<cfset DaysSinceTurnOfCentury = DateDiff("d", "1/1/2000", now() ) />
<cfoutput>#DaysSinceTurnOfCentury#</cfoutput>
```

Or suppose you want to know what the date will be 42 days from now:

```
<cfset FortyTwoDaysFromNow = DateAdd("d", now(), 42 ) />
```

ColdFusion provides many useful functions to work with dates and times. You can compare dates, find out how far two dates are apart, reformat a date, and more. See the ColdFusion Date/Time documentation for a full listing:http://help.adobe.com/en_US/ColdFusion/10.0/CFMLRef/WSc3ff6d0ea77859461172e0811cbec1a60c-7ffc.html#WSc3ff6d0ea77859461172e0811cbec22c24-6986

Arrays

Is Simple: No

Arrays are an ordered series of data. Here's an example of a one dimensional array:

Array Creation

```
<cfset ThingsILike = ["Warm Sandy Beaches", "Tropical Drinks", 42] />
<cfdump var = "#ThingsILike#" />
```

Alternate Method: Array Creation

Here's another way to create an array, along with a couple of different ways to add data to an array:

```
<cfset ThingsILike = arrayNew(1) />
```

Adding items to an Array

You can add things by a specific position. Note: Arrays in ColdFusion start at 1, not 0.

```
<cfset ThingsILike[1]  = "Warm Sandy Beaches" />
```

Alternate Method: Adding items to an Array

You can append an item to the end of the array

```
<cfset ArrayAppend( ThingsILike,  "Tropical Drinks") />
<cfset ArrayAppend( ThingsILike,  42) />
<cfdump var = "#ThingsILike#" />
```

See how I defined the strings in my array with quotes, and non-strings without? Each element in the array is an execution zone also, so if you need ColdFusion to evaluate something, just add it in:

```
<cfset ImportantDates = ["12/26/1975", now() ] />
<cfdump var = "#ImportantDates#" />
```

Displaying the Contents of an Array

You can not use the `cfoutput` command on an array because complex data types such as arrays are not displayable as a string. You can loop over the array, however, and output the strings to the page:

```
<cfset ThingsILike = ["Warm Sandy Beaches", "Tropical Drinks", 42] />
<cfloop array="#ThingsILike#" index="thing">
    <cfoutput>#thing#</cfoutput>
</cfloop>
```

ColdFusion provides many useful functions to work with arrays. You can search the contents of an array for a value, perform mathematical functions such Sum or Average, sort the contents of an array, and more. See the ColdFusion Array function documentation for a full listing: http://help.adobe.com/en_US/ColdFusion/10.0/CFMLRef/WSc3ff6d0ea77859461172e0811cbec1a60c-7ffc.html#WSc3ff6d0ea77859461172e0811cbec22c24-6a66

Structs

Is Simple: No

Structs, aka Structures, are a collection of data, stored by a key, or name. Suppose for example, you wanted to store several kinds of fruit and also whether you like it or not. Structs provide a way to organize like name/value pairs and let you refer to them as a single collection.

Struct Creation

```
<cfset FruitBasket = structNew() />
```

Alternate Method: Struct Creation

```
<cfset FruitBasket = {} />
```

Adding items to a Struct: Bracket Notation

```
<cfset FruitBasket = {} />
<cfset FruitBasket["Apple"] = "Like" />
<cfset FruitBasket["Banana"] = "Like" />
<cfset FruitBasket["Cherry"] = "Dislike" />

<cfdump var = "#FruitBasket#" />
```

Adding items to a Struct Dot Notation

```
<cfset FruitBasket = {} />
<cfset FruitBasket.Apple = "Like" />
<cfset FruitBasket.Banana = "Like" />
<cfset FruitBasket.Cherry = "Dislike" />

<cfdump var = "#FruitBasket#" />
```

Struct Creation and Population in One Statement

```
<cfset fruitBasket = {
    "Apple" = "Like",
    "Banana" = "Like",
    "Cherry" = "Dislike"
} />
<cfdump var = "#FruitBasket#" />
```

> **Pro Tip:** There are reasons to use one struct notation over another. If you ran some of these examples you would notice that the Bracket Notation preserved the case of the keys and the Dot Notation did not. Sometimes the preservation of case is important, like when passing values to Javascript or other case sensitive languages or formats. In the Struct Creation and Population in One Statement Example, the case will be preserved as long as they keys are surrounded by quotes. If the keys are not quoted, the case will be converted to upper case.
>
> Also, the bracket notation allows for a dynamic key reference. This is helpful when the name of the struct key will come from a runtime operation, such as looping over the struct. Many people find Dot Notation easier to read and use it most of the time, except in cases where Bracket Notation offers a feature Dot Notation does not.

Displaying the Contents of a Struct

See how your preference was mapped to the kind of fruit? You can't use the `cfoutput` command on structs either because, once again, they aren't displayable as a string. You can loop over the struct and output the keys and values to the page:

```
<cfloop collection="#FruitBasket#" item="fruit">
    <cfoutput>I #FruitBasket[fruit]# #fruit#</cfoutput><br />
</cfloop>
```

ColdFusion provides many functions to work with structs. You can search the contents of struct, make complete and partial copies, retrieve a list of just the keys, and more. See the ColdFusion Struct function documentation for a full listing:
http://help.adobe.com/en_US/ColdFusion/10.0/CFMLRef/WSc3ff6d0ea77859461172e0811cbec1a60c-7ffc.html#WSc3ff6d0ea77859461172e0811cbec22c24-69b8

Queries

Is Simple: No

Queries are recordsets. Recordsets contain a series of columns with 0 or more rows. You can think of a query like a single page of a spreadsheet with the columns across the top and rows down the side.

Most ColdFusion programs interact with databases. Database interaction takes the form of a query and ColdFusion makes it very easy to work with the data returned by the database. Databases will be covered more in more detail later on in the course.

```
<cfquery name="FruitQuery" datasource="fruit">
    SELECT Name, Price
    FROM FruitStore
    WHERE Price < 7
</cfquery>

<cfloop query="FruitQuery">
    #FruitQuery.Name# costs #FruitQuery.Price# <br />
</cfloop>
```

Query Objects have a few special properties. You can use these properties to get specific information about the data inside the query.

- **Queryname.recordcount** = How many rows does this query have?
- **Queryname.columnlist** = What columns does this query have?
- **Queryname.currentrow** = What row number are we currently on inside a `cfoutput` or `cfloop`?

ColdFusion provides several useful functions to work with query objects. You can create your own query objects in ColdFusion without a database call or even retrieve all the values in a specific column in a list format. See the ColdFusion Query function documentation for a full listing: http://help.adobe.com/en_US/ColdFusion/10.0/CFMLRef/WSc3ff6d0ea77859461172e0811cbec1a60c-7ffc.html#WSc3ff6d0ea77859461172e0811cbec22c24-67fe

Also, be sure to review the Accessing and Using Data portion of the ColdFusion documentation, which explains a number of key concepts and techniques for working with data, databases and query objects:http://help.adobe.com/en_US/ColdFusion/10.0/Developing/WS8f0cc78011fffa7168855f811cdb0b0cce-8000.html

Commenting

In our previous sections we talked about the importance of adding context into your programs. You'll find you will spend much more time reading programs for the purposes of debugging and enhancing, than you spend writing the program. So take care to write your programs in such a way the program can be read and understood as easily as possible.

Context is important because our programs express a problem to a computer using a kind of shorthand, in our case ColdFusion programming code. The computer doesn't much care about the amount of context in the application, as long as the instructions (code) are all properly formed. The program is good enough for the computer if it works well. But this doesn't necessarily mean the program is good enough for a human. Programs that are good for humans can be efficiently understood, debugged and enhanced by another programer.

One way to add context is to use descriptive variable names.

Not Descriptive:

```
<cfset var1 = "42" />
```

Descriptive:

```
<cfset AnswerForEverything = 42 />
```

Another way to add context to your program is by writing a descriptive comment. ColdFusion comments resemble HTML comments, only with an extra dash on each end.

Any text or programming code inside of a comment is not executed nor displayed. Only a person with access to the source code can see the content inside of a comment.

```
HTML Comment
<!-- I am an HTML Comment-->
ColdFusion Comment
<!---  I am a ColdFusion Comment--->
<cfscript>
    // I am a ColdFusion Comment in CFScript for a single line
```

```
    /*
        I am a multi-line
        ColdFusion Comment in CFScript
    */
</cfscript>
```

Example of Good Comments

You can use comments to describe your intent and give context to a section of your programming code:

```
<!--- Always load the hard coded value if specified --->
<cfif len( trim ( ChartHelperName ) )>
    <cfset ReportPeriod = ChartHelperName />
</cfif>


<!--- Load the Chart Helper as defined in the config --->
<cfset ChartHelper = ChartHelperLoader.load( ReportPeriod, filter,
ChartHelperOptionList ) />


<!--- Add the bits to the helper, shove the helper in the event then
announce the right result --->
<cfset arguments.event.setValue( ChartHelperEventValue, ChartHelper ) />
```

Programming code explains the HOW, portion of problem solving. Proper use of comments explain the WHY part of the problem solving. It's much better to make comment notations about your program when you are writing your program because that is when you best understand the problem and how you have chosen to solve it.

Later on, after the program has been written, you will be very happy to have good comments in your program explaining the WHY part of your program, so you can make updates and alterations in the same spirit as when you wrote the initial program.

ColdFusion Tags vs. ColdFusion Script

ColdFusion has been designed to allow the use of ColdFusion tags and ColdFusion script. Functionally, both will be interpreted equally and will achieve the same result. The use of ColdFusion Tags and ColdFusion Script is up to personal or team preference. One or two ColdFusion operations are currently Tag only, as of this tutorial, but these will be made available in ColdFusion script in future releases.

Some developers write all ColdFusion code in Tags. Some developers write all ColdFusion code in Script. Some write the view portions of their ColdFusion code in Tags, and the business layer portion in Script. As long as you stay consistent, all approaches are valid. The overarching rule should be legibility and consistency.

Let's look at some statements and compare the differences:

Setting a variable:

```
<cfset variable = "value" />
<cfscript>
    variable = "value";
</cfscript>
```

Looping over an array:

```
<cfset FruitArray = ["apple", "banana", "cherry"] />
<cfloop from="1" to="#arrayLen( FruitArray)#" index="i">
    <cfoutput>#FruitArray[i]#</cfoutput>
</cfloop>

<cfscript>
    FruitArray = ["apple", "banana", "cherry"];
    for( i=1; i <= arrayLen(FruitArray); i++){
        writeOutput(FruitArray[i]);
    }
</cfscript>
```

Some developers find the ColdFusion Script syntax to resemble other familiar programming languages. ColdFusion is designed to provide a welcoming and productive developer experience. Try both styles to see which you prefer. The ColdFusion documentation has several good pages on ColdFusion script. http://help.adobe.com/en_US/ColdFusion/10.0/Developing/WSc3ff6d0ea77859461172e0811cbec22c24-7feb.html

Hands On 1

By Simon Free

In the first part of this hands on we are going to convert some pages to .cfm pages. We will then add some variables and output them to the user. Finally, we will add a comment to our code.

Tags Used: <cfset>, <cfoutput>

1. Rename the `/www/index.html` file to `index.cfm`.
2. Confirm that the ColdFusion page displays by openning up `/www/index.cfm` in your browser.
3. Open up the `/www/index.cfm` file in your code editor.
4. First, let's create some variables that we will display later on in the page. On line 1, write a `<cfset>` tag that sets the variable `myName` with a value of your name. The tag should look something like this:

```
<cfset myName="Simon" />
```

5. On the next line, create another variable called `myPosition` with a value of 'Developer'. The tag should look something like:

```
<cfset myPosition="Developer" />
```

6. Now, let's output these variables. Locate the text [Name] on or around line 80. Replace [Name] with `#myName#`.
7. Locate the text '[position]' on or around line 81. Replace [position] with `#myPosition#`.
8. Refresh the page in your browser and confirm that you now see `#myName#` and `#myPosition#` displayed.
9. In your code editor, locate the `#myName#` variable on or around line 80 and place a `<cfoutput>` tag before it and a `</cfoutput>` tag after it so that the code looks similar to this:

```
<cfoutput>#myName#</cfoutput>
```

10. Do the same for the `myPosition` variable on or around line 81.
11. Refresh the browser and confirm that you now see your name and position displayed on the page.
12. Go to line 2 and update the variable `myPosition` so that it has a value of 'A Developer'. The tag should look similar to this:

```
<cfset myPosition="A Developer" />
```

13. Refresh the browser and confirm that you see your change.
14. In your code editor, locate the comment that says `Data Output`. Add a new line after this tag and write the following text: "This is where the name and position are output".
15. Refresh the browser and confirm that you see this text displayed on the page.
16. Go back to your code editor and add ColdFusion comments around the line of text so that it looks similar to this:

```
<!--- This is where the name and position are output --->
```

17. Refresh the browser and confirm that the text is no longer visible on the page.
18. In your code editor, locate the link to `index.html` on or around line 52. Change the link so that it points to `index.cfm`.
19. Now let's update some of the links to point to the new .cfm pages. Locate the link to `about.html` on or around line 53 and change it to point to `about.cfm`.
20. Rename the `/www/about.html` file to `about.cfm`.
21. Refresh your browser and click on the `about` link at the top of the page.
22. Confirm that the about page loads.

Hands On 2

By Simon Free

In this section of the hands on we will switch from tag based code to script based code and create a structure of data. We will then output that on the page.

Tags Used: <cfscript>, <cfoutput>

1. Open up the `/www/about.cfm` file in your code editor.
2. Locate line 1 and write an opening and closing `<cfscript>` block. Put the closing `</cfscript>` tag on the second line. Your code should look something like this:

```
<cfscript>
</cfscript>
```

3. In between the opening and closing `<cfscript>` tags, create a struct by write the following code:

```
personalInfo = {name='', dob='', address='', phonenumber='', email='',
website='', skype=''};
```

4. For each variable of the `personalInfo` struct, provide some information (you do not have to use real information if you prefer not to).
5. Locate the [Name] placeholder text on or around line 115.
6. Replace [Name] with `#personalInfo.name#`.
7. Replace the other placeholders with their matching variables in the `personalInfo` struct.
8. Wrap all the `<div>` tags with the class of clr in a `<cfoutput>` tag. Your code should look similar to this:

```
<cfoutput>
    <div class="clr"><div class="input-box">Name
</div><span>#personalInfo.name#</span> </div>
    <div class="clr"><div class="input-box">Date of birth </div><span>
#personalInfo.dob#</span></div>
    <div class="clr"><div class="input-box">Address</div><span>
#personalInfo.address#</span></div>
    <div class="clr"><div class="input-box">Phone</div>
```

```
<span>#personalInfo.phoneNumber#</span>  </div>
    <div class="clr"><div class="input-box">E-mail</div><span>
<a href="#">#personalInfo.email#</a></span>  </div>
    <div class="clr"><div class="input-box">Website </div> <span>
<a href="#">#personalInfo.website#</a></span> </div>
    <div class="clr"><div class="box1">Skype </div> <span>
<a href="#">#personalInfo.skype#</a></span> </div>
</cfoutput>
```

9. To test that your changes have taken place, load the `/www/about.cfm` page in your browser.
10. Notice that you have received a ColdFusion Error. The error states that it has found an Invalid CFML construct. This is due to the single # signs inside the links for email, website, and Skype. ColdFusion uses the # sign to denote variables and it is trying to evaluate everything between the first # sign and the next one. To resolve the error, we must escape the single # signs.
11. Locate the link that wraps around the email output on or around line 121.
12. Update the value of the href attribute to ## rather than #. You're `<a>` tag should look similar to this:

```
<a href="##">
```

13. Do this also for the website and Skype links.
14. Refresh the `/www/about.cfm` page in your browser and confirm that you now see your information being displayed.

Homework

- Convert all remaining .html pages to .cfm pages.
- Update all navigation links to point to the .cfm files rather than the .html files.
- Create a structure on the Contact page that stores your address, phone number, email, and Skype information; display it on the right hand side under Contact Info.

2

Decision Making and Scopes

By Nic Tunney

About Nic Tunney

Nic has been working on web applications since the late 90's spending most of his life using the Adobe product stack. He's been everything from a tech instructor to a web developer to VP of Software Architecture and Director. This experience gives him a unique perspective on everything from full stack application architecture to project and product management. Nic loves the opportunity to learn something new and solve problems using best of breed technologies, whatever that may be.

Decision Making

An important part of any program is the ability to make decisions based upon a set of conditions. This is aptly named 'conditional logic'. Unless you are writing a static set of HTML pages, you will almost certainly need to evaluate conditions. In conditional logic, all conditions tested must evaluate to a Boolean, either true or false. As you learned in the variables section of this training, ColdFusion is a dynamically typed language. This does not mean that ColdFusion is not typed, but rather that ColdFusion can switch from type to type dynamically while the application is executing. This is a great feature when using conditional logic as you do not need to explicitly define the Boolean such as `myVar == true`, but for simple variables you can allow ColdFusion to evaluate the value of the variable and convert it dynamically to a Boolean value. Keep this in mind, and we will revisit this in a bit.

ColdFusion has both `if/then/else` tags as well as `switch/case` logic. We will look at `if/then/else` logic first. There are three tags that are important when evaluating conditions: `cfif`, `cfelseif`, and `cfelse`.

`cfif` is used to start a decision block. As with any properly formed tag block, being XHTML compliant, a `cfif` block is terminated with `</cfif>` . Inside the conditional block you will place either code to execute or markup to output. A simple case looks like this:

```
<cfif expression>
    ..Markup..
</cfif>
```

The decision block above says that if the expression evaluates to `true`, then execute the tag contents. If the expression evaluates to `false`, do nothing. However, if we want it to do something if the expression evaluates to `false`, we use the `cfelse` tag like this:

```
<cfif expression>
    ..Markup..
<cfelse>
    ..More Markup..
</cfif>
```

If you need to compare several conditions in succession, use `cfelseif`:

```
<cfif expression>
    ..Markup..
<cfelseif another_expression>
```

```
        ..More Markup..
<cfelse>
        ..More Markup..
</cfif>
```

For review, an expression is any value that will evaluate to a Boolean (`true` or `false`). You can provide a string value that will evaluate to a Boolean such as a number. Any positive number will evaluate to `true`; 0 evaluates to `false`. Negative numbers evaluate to `false` as well. You may also call a function that returns a Boolean or numeric value like these:

```
len(myStringVar); // returns a number
isNull(mySimpleVar); // returns a Boolean
arrayLen(myArray); // returns a number
```

Sometimes you will need to perform a values comparison. ColdFusion includes many decision operators, all of the following being case-insensitive.

CFML and CFScript Operators	**CF8+ CFScript Only**
IS, EQUAL, EQ	==
IS NOT, NOT EQUAL, NEQ	!=
GT, GREATER THAN, LT, LESS THAN, GTE, LTE	>,<,>=,<=
CONTAINS	N/A
DOES NOT CONTAIN	N/A

These operators allow you to compare a set of values. An example in action is:

```
<cfif myValue EQ 'ColdFusion'>
        ..code…
</cfif>
```

Below is a fully formed example of a script based `if/then/else` block. Since ColdFusion is ECMAScript compliant, it should look very familiar:

```
if (myValue == 'ColdFusion') {
        ..code..
} else if (myValue != '.NET') {
        x = 277;
```

```
} else {
        ..code..
}
```

There are also times you will want to compare more than one expression. ColdFusion provides a series of Boolean operators to permit join and negation operations:

- AND / &&
- OR / ll
- NOT / !
- XOR, EQV
- IMP - implication

An example of a compound expression looks like this:

```
<cfif structKeyExists(myStruct, 'age') AND myStruct.age LTE 7>
     ..code..
</cfif>
```

The important thing to know about compound expressions is that ColdFusion will move from left to right when processing joined expressions. The example above first checks to make sure the `age` key exists in the structure called `myStruct` before performing further operations on it. If we check the value of the key and the key is not defined, ColdFusion will throw an error. Since ColdFusion works from left to right, ColdFusion would first see that age does not exist as a key in `myStruct` (`structKeyExists` will return `false`) and it will not perform the second operation. This is not the case for `OR` operations. Look at the following example:

```
<cfif myVar EQ 1 OR myVar EQ 10>
        ..code..
</cfif>
```

ColdFusion would look first at `myVar EQ 1`. If it returned `false`, it would simply move on to evaluate the second expression. In this case, if our variable was equal to 1 or 10, the code inside the `cfif` block would be evaluated.

In the example above, we are checking if a variable is equal to a set value. When programming, `if/then/else` logic can become cumbersome when you need to perform an action based on a variable's value. Programming languages use the `switch/case` construct for this type of conditional logic. ColdFusion uses 3 tags to define `switch/case` statement: `cfswitch`, `cfcase`, and `cfdefaultcase`. Look at the sample code below:

```
<cfswitch expression="#myVar#">
	<cfcase value="1">
		..code..
	</cfcase>
	<cfcase value="9,10" >
		..code..
	</cfcase>
	<cfdefaultcase>
		..finally..
	</cfdefaultcase>
</cfswitch>
```

The code above defines a switch block based on the expression of the value of `myVar`. Note that we are using hash marks (#) here as the `expression` attribute, taking a string value, and comparing it to a set of possible matching values. The possible values are each defined in their own `cfcase` block. `cfcase` takes a string attribute named `value`, and if it matches the current value, it will execute the code contained inside of the `cfcase` block. A value may only be declared once within the entire switch block or ColdFusion will throw an error. `cfswitch` will work from top to bottom to find the first matching value. If it cannot find a matching value, it will execute the code contained inside of the `cfdefaultcase` block. It is not required to have a `cfdefaultcase` block defined, but your logic will dictate if it is present.

The second `cfcase` declaration above lists more than one value. If the expression is either `9 OR 10`, the code will execute. The default delimiter is ','. If the value contains a comma, it is automatically treated as a list. If you want the value to include a comma in the string, simply specify an alternate delimiter using the `delimiters` attribute like this:

```
<cfcase value="9,10" delimiters="|">
	..code..
</cfcase>
```

This will allow the comparison value to be a string of 4 characters "9,10".

Astute programmers may have noticed we have not had to define a break for each case. This is true in tag based CFML `switch/case` statements. In `cfscript`, you will need to use the statement as with any other programming language. Here is an example of a switch case construct in `cfscript`:

```
switch (myVar) {
	case 1:
		writeOutput('Value was 1');
		break;
	case 9:
		writeOutput('Value was 9');
		break;
	default:
		writeOutput('Value was not 1 or 9');
}
```

ColdFusion also offers ternary operations. A ternary operation is a construct that acts as a shorthand assignment operation given a comparison with exactly two possible outputs. Here is an example in `cfscript`:

```
x = (myVar == 1) ? 1 : 277;
```

The above is equivalent to saying:

```
if (myVar == 1) {
	x = 1;
} else {
	x = 277;
}
```

Scopes

ColdFusion groups variables together in scopes, or a range in which a variable can be accessed. Think of a scope as a bucket of memory that can store variables. Each scope has its own purpose, and each scope has its own lifecycle.

The following table shows major scopes available in a running ColdFusion application:

- **Variables**: Default scope available in ColdFusion templates. Variables are available only during the execution of the template.
- **URL**: All variables in the query string or sent to ColdFusion via an HTTP GET request are available in the URL scope. URL variables are available for the current request.
- **Form**: All variables posted from a form (HTTP POST) are available in the Form scope. Form variables are available for the current request.
- **CGI**: CGI variables sent from the browser are placed into the CGI scope. CGI variables are available for the current request.
- **Query** (not a true scope): Upon execution of a query, the resultset is placed into a named scope as specified by the operator assignment or `cfquery` tag's `name` attribute. The data stored in this pointer is available for the current request.
- **Server**: Developers may choose to utilize the server scope to share data across application running within the context of the current ColdFusion instance or cluster. This scope persists across requests, and is available until the server shuts down.
- **Application**: Application variables are shared amongst all connected clients for the current named application. This scope is also used for objects instantiated using the singleton pattern. This scope is available across requests for the life of the application, which may terminate on server shutdown, application malfunction, or application timeout.
- **Session**: Developers use session variables to store a single visitor's data across requests. This scope is only available to the current session, and will persist until server or application termination, or session timeout.
- **Request**: The request scope contains data that is available to all functions, CFCs, templates, and custom tags executed during the context of the current request. Data in this scope is available during the current request.
- **Arguments**: The arguments scope contains data passed into a ColdFusion function. The arguments scope is mutually exclusive with the local function scope, and may not contain the same variable names as the local scope. This scope is available during the current execution of a function, and is private to the current function context.
- **Attributes**: This scope contains variables passed in as attributes to a ColdFusion custom tag. The data in this scope is available during the execution lifespan of a custom tag. Refer to the ColdFusion Livedocs for additional scopes available to custom tags, as well as how scopes are handled in nested custom tags.
- **Local (function)**: The Local scope may be referenced explicitly, or defined using the var keyword. Variables in this scope are private to the current function context. This scope is mutually exclusive with the arguments scope, and may not contain the same variable names as the arguments scope.

NOTE: The above list is not all-inclusive. Please reference the Adobe Livedocs documentation for additional and tag specific scopes.

To reference a scoped variable in ColdFusion, we preface the variable pointer with the scope in dot-notation. To access a variable named `myVar` in the `Application` scope, we would write `application.myVar`. It is considered to be a best practice to scope all variables in your ColdFusion application; however, it is not always necessary. ColdFusion will attempt to figure out what scope the variable is stored in if you do not explicitly provide one by searching each scope one at a time using scope precedence. The following list shows the order in which ColdFusion will search for a variable matching the pointer you have provided.

1. Arguments
2. Local (function-local, UDFs and CFCs only)
3. Thread local (inside threads only)
4. Query (not a true scope; variables in query loops)
5. Thread
6. Variables
7. CGI
8. CFFile
9. URL
10. Form
11. Cookie
12. Client

Note that some scopes are not searched in the precedence above. Variables within these scopes must ALWAYS be requested in dot notation, prefaced by the scope name.

- This (public scope of a ColdFusion component)
- Request
- Application
- Session
- Server

Even though ColdFusion defines scope precedence, there are a number of reasons to always provide a scope when requesting access to a variable such as readability and maintainability; we will explicitly discuss of these reasons. The first is that variable names are not exclusive in ColdFusion. We can store a value in multiple scopes with the same name, and which is especially prevalent across persistent scopes such as the application and session scopes, as well as when you are using reusable or third-party components and custom tags. A common example would be if our program has a variable in the `Variables` scope called `userID` and we posted a `form` that contains a variable called `userID`, which can result in a collision. Based on ColdFusion scope precedence, if we referred to the variable `userID` without explicitly specifying the scope, ColdFusion would return the data stored in `variables.userID` instead of `form.userID`. This can cause problems if we were trying to use the latter data and makes your code hard to read and maintain.

The second reason to scope variables is that there is a minor performance gain in providing the explicit scope because ColdFusion does not have to locate the variable across a number of scopes. This performance increase has been essentially relegated to nil in the current versions of ColdFusion, but there is still a negligible improvement when scope is explicitly defined.

There are also times when you might not want to scope variables. One example is when you are not

sure the source of the variable you will be accessing, such as when a variable could come from either the form or URL scopes. This circumstance is rare and can introduce security risks, so you should generally attempt to find an alternative solution, such as implementing logic using `cfparam`.

Hands On 3

By Simon Free

In this hands on, you are going to add some conditional logic to the site as well as create a form post.

Tags Used: <cfif>, <cfparam>

1. Open up the `/www/contact.cfm` file in your code editor.
2. Locate the comment that reads `Message Output`.
3. Below the comment, create a `<cfif>` statement that checks if the `form.submitted` variable is true. Your code should look something like this:

```
<cfif form.submitted>


</cfif>
```

4. Inside the `<cfif>` tag, place the text `<p>Your form has been submitted</p>`.
5. Navigate to the `/www/contact.cfm` page in your browser.
6. Notice that you receive an error stating that 'Element Submitted is undefined in form'. This error is thrown because the `form.submitted` variable does not exist. When dealing with forms, it is important to provide `<cfparam>` tags that will create the variable if it is not already defined.
7. Go to line 1 of `contact.cfm` and add the following line of code:

```
<cfparam name="form.submitted" default="0" />
```

8. Refresh the page in the browser and notice that the error has been resolved.
9. Locate the `<form>` tag on or around line 109.
10. Update the `action` attribute to point to contact.cfm.
11. Refresh the page in your browser and click the submit button.
12. Notice that the form was submitted but the text was not displayed. This is because the `form.submitted` value still does not exist, and the `<cfparam>` is setting that value for us.
13. Locate the closing `<form>` tag on or around line 130.
14. Before the closing `</form>` tag, add a hidden input called `submitted` and give it a value of '1'. The tag should look similar to this:

```
<input type="hidden" name="submitted" value="1" />
```

15. Refresh the page in your browser and click the submit button.
16. You should now see the text 'Your form has been submitted' displayed on the page.

Hands On 4

By Simon Free

In this hands on, you are going to validate the form data that was submitted and output any problems that might have occurred. We will access data from the form scope.

Tags Used: <cfset>, <cfif>
Functions Used: len

1. Locate the comment that says `Message Output` in the `/www/contact.cfm` file and remove all the code that is inside the `<cfif>` tag just below it.
2. Inside the `<cfif>` tag create a variable called `OK` and set it to `true`.
3. Below the `<cfset>`, create a `<cfif>` statement that checks if the length of the value of `form.contactname` is equal to 0. To do that, use a function called `len()` which will return the number of characters in the provided string. Your code should look similar to this:

```
<cfset ok = true />


<cfif len(form.contactname) eq 0>


</cfif>
```

4. Inside the `<cfif>` statement, set the `OK` variable to false.
5. You should end up with code similar to this:

```
<cfset ok = true />


<cfif len(form.contactname) eq 0>
    <cfset ok = false />
</cfif>
```

6. Create `<cfif>` statements for the email and message fields in the form.
7. After the last `<cfif>` statement, create a new `<cfif>` statement that checks if the `OK` value is equal to false. If it is equal to false, output the following:

```
<p>You did not provide all the required information!</p>
```

8. Your final block of code should look similar to this:

```
<cfif form.submitted>
     <cfset ok = true />

     <cfif len(form.contactname) eq 0>
          <cfset ok = false />
     </cfif>

     <cfif len(form.email) eq 0>
          <cfset ok = false />
     </cfif>

     <cfif len(form.message) eq 0>
          <cfset ok = false />
     </cfif>

     <cfif ok eq false>
          <p>You did not provide all the required information!</p>
     </cfif>
</cfif>
```

9. Reload the `contact.cfm` page in your browser and click 'submit'. You should see the message "You did not provide all the required information!" displayed.
10. Fill in all the fields in the form and click 'submit'. You will notice that the message does not display.

Hands On 5

By Simon Free

In this hands on, you are going to update our code to provide a better user experience to your users as well as improve the form validation.

Tags Used: <cfparam>, <cfoutout>
Functions Used: len, trim

1. Locate the `<cfparam>` tag at the top of the `/www/contact.cfm` file.
2. Below the `<cfparam>` tag, create another one for `contactname`, `email`, and `message`. Leave the `default` attribute as empty. Your code should look similar to this:

```
<cfparam name="form.contactname" default="" />
<cfparam name="form.email" default="" />
<cfparam name="form.message" default="" />
```

3. Locate the `contactname` form input on or around line 132 and add a `value` attribute with a value of `#form.contactname#`.
4. Locate the `email` form input on or around line 136 and add a `value` attribute with the value of `#form.email#`.
5. Locate the `message` textarea on or around line 140 and place the value `#form.message#` between the open and close tags.
6. Confirm your code looks similar to this:

```
<div>
    <label>Name <span class="font-11">(required)</span></label>
    <input name="contactname" type="text" class="required"
value="#form.contactname#"/>
</div>
<div>
    <label>E-mail <span class="font-11">(required)</span></label>
    <input name="email" type="text" class="required email"
value="#form.email#"/>
</div>
```

```
<div class="textarea">
    <label>Message <span class="font-11">(required)</span></label>
    <textarea name="message" rows="6" cols="60" class="required">
#form.message#</textarea>
</div>
```

7. Locate the open `<form>` tag on or around line 129 and put on open `<cfoutput>` tag before the form tag.
8. Locate the closing `</form>` tag on or around line 146 and put a closing `</cfoutput>` tag after the form tag.
9. Reload the `contact.cfm` page in your browser; fill in 2 of the 3 inputs of the form and click 'submit'.
10. Notice that the form is now pre-populated with the values you submitted, but you still receive the message "You did not provide all the required information!"
11. Reload the `contact.cfm` page in your browser.
12. Add a space in all form fields and click submit.
13. Notice that the message does not display on the page. This is due to the fact that the form fields do have a length, even though it is just a space.
14. Locate the `<cfif>` statement that checks the length of `form.contactname` on or around line 108.
15. Wrap the `form.contactname` variable in a `trim()` function. This will remove all whitespace at the beginning and end of the variable.
16. Your code should look similar to this:

```
<cfif len(trim(form.contactname)) eq 0>
```

17. Add the `trim` function to the `form.email` and `form.message <cfif>` statements.
18. Reload the `contact.cfm` page in your browser and add spaces to all 3 form fields. Notice that the message is displayed.

Hands On 6

By Simon Free

In this hands on, we are going to refactor our code and take out any unnecessary logic we might have.

Tags Used: <cfif>, <cfelse>

1. Open up the `/www/contact.cfm` file in your code editor and locate the `<cfif>` statement that checks the `form.contactname` variable. Remove the `eq 0` part of the expression; as any number greater than 0 is treated as true, and the number 0 is treated as false, we do not need this additional part of the <cfif> statement. Because we only want to run the code when there is NO value passed we must negate the value by putting the word NOT before it.
2. Remove the `eq 0` part of the `form.email` and `form.message <cfif>` statements.
3. Locate the `<cfif>` statements that checks the variable `OK` on or around line 120.
4. Remove the `eq false` part of the expression. As a `<cfif>` tag is looking for a true or false value, we do not need to check the value of the variable as it is already going to be a true or false value. As we are checking if the variable is false then we need to put the ! symbol before the variable. ! is the same as NOT.
5. Before the closing `</cfif>` tag of the expression that checks the `OK` variable, insert a `<cfelse>` tag.
6. Between the `<cfelse>` tag and the closing `</cfif>` tag add the following code:
   `<p>Form submitted successfully!</p>`
7. Your final code block should look similar to this:

```
<cfif form.submitted>
    <cfset ok = true />

    <cfif len(trim(form.contactname))>
        <cfset ok = false />
    </cfif>

    <cfif len(trim(form.email))>
        <cfset ok = false />
    </cfif>
```

```
    <cfif len(trim(form.message))>
        <cfset ok = false />
    </cfif>

    <cfif ok>
        <p>You did not provide all the required information!</p>
    <cfelse>
        <p>Form submitted successfully!</p>
    </cfif>
</cfif>
```

8. Reload the `contact.cfm` page in your browser and enter spaces in all form fields.
9. Click 'submit' and notice that the message "You did not provide all the required information!" message is displayed.
10. Provide information for all the form fields and click 'submit'.
11. Notice that the message "Form submitted successfully!" is displayed.

3

Looping

By Nic Tunney

Looping is employed by programming languages to repeat a behavior defined by a section of code either while a condition is maintained `true` (conditional looping), or a certain number of times (iterative looping). As with other programming languages, ColdFusion allows you to perform both iterative and conditional looping.

ColdFusion uses the `cfloop` tag to enable looping in tag-based syntax. `cfloop` permits you to loop over a variable such as a `structure` or `array`, loop while a condition remains `true`, or loop for a given iteration. ColdFusion permits looping in `cfscript` using `for loops` and `do while` loops.

Tag-based Looping

It is easiest to demonstrate looping by example. Consider the following code:

```
<cfloop from="1" to="5" index="i">
	#i#<br />
</cfloop>
```

The code above performs an iterative loop. The `from` attribute specifies the start value, while the `to` attribute specifies the end value. By default, `cfloop` will increment by 1 each iteration. You can override this behavior using the `step` attribute, which can be any positive or negative number. This means you can also decrement the loop's index by specifying `-1` for the `step` attribute. The `index` attribute stores the current index value of the loop. This is the value that increments on each execution. The iterative loop will continue execution until the index reaches the max value as specified by the `to` attribute. It is important to note that the loop will execute while the index is equal to the maximum value, but then exit. This means the output from the code above is:

```
1
2
3
4
5 <- notice that the loop executed while the index was equal to the maximum value
```

Iterative looping generally uses variables to specify the bounds of the iteration, unlike the example above. A great example of this is looping over an `array`. Until CF9, we could only loop over an array through the use of iterative looping as shown below. This case is still useful today, especially when you need to keep track of the current `index` as well as the array value. The important thing to keep in mind is that ColdFusion arrays are 1-indexed instead of 0-indexed as with some other programming languages. This means we have to start our loop counter at 1 when looping over an array.

```
<cfset myArray = ['Jeff', 'John', 'Steve', 'Julianne'] />


<cfloop from="1" to="#arrayLen(myArray)#" index="i">
	#i#: #myArray[i]#<br />
</cfloop>
```

For the upper limit of this iterative loop, we are passing in a variable equal to the present size of our array. This loop will execute the code within the `cfloop` block one time for each value in the array. The output when running the code above is:

```
1: Jeff
2: John
3: Steve
4: Julianne
```

CF9 and above allows us to loop over an array using shorthand. This is useful when you only need the data contained within the array, but do not need the associated index.

```
<cfloop array="#myArray#" index="item">
	#item#<br />
</cfloop>
```

Don't let the `index` attribute fool you; the index that is returned is not an index, but rather the data contained within the array at the current index. We no longer have a reference to the array index itself. The output when running the code above is:

```
Jeff
John
Steve
Julianne
```

`cfloop` permits the developer to loop over a list. The list `datatype`, as we learned earlier, is a delimited `String`. ColdFusion understands that the string is delimited and provides special functions for the list. A list loop is displayed in the code below:

```
<cfset myList = 'Jeff,John,Steve,Julliane' />
<cfloop list="#myList#" index="item">
	#item#<br />
</cfloop>
```

If you need to keep track of the index as opposed to just the value of the current item position, you can use a standard iterative loop:

```
<cfloop from="1" to="#listlen(myList)#" index="i">
	#i#: #listGetAt(myList, i)#<br />
</cfloop>
```

Another neat trick you can use in a list loop is by using the `delimiters` attribute. You can specify multiple delimiters as well. The following example allows you to break down a string into individual words:

```
<cfset myList = "This is the test sentence" />
<cfloop list="#myList#" index="word" delimiters=" ">
	#word#<br />
</cfloop>
```

The output from the code above is:

```
This
is
a
test
sentence
```

ColdFusion also allows you to loop over the items in a `collection` (structure) one iteration per key. Using associative array notation, you can get the value for each key in a structure.

```
<cfset myStruct = { name= 'Jeff Katersian', id= 12445, dob= '1/2/1994' } />


<cfloop collection="#myStruct#" item="key">
	#key#: #myStruct[key]#<br />
</cfloop>
```

The code above results in the following output:

```
NAME: Jeff Katersian
DOB: 1/2/1994
ID: 12445
```

ColdFusion allows you to loop over the contents of a `query` using `cfloop` as well. The loop will execute the code inside the `cfloop` tag one time per query row.

```
<cfscript>
	myQuery = queryNew("id,user");
	queryAddRow(myQuery);
	querySetCell(myQuery, 'id', '1');
	querySetCell(myQuery, 'user', 'Jeff');
	queryAddRow(myQuery);
	querySetCell(myQuery, 'id', '2');
	querySetCell(myQuery, 'user', 'John');
	queryAddRow(myQuery);
	querySetCell(myQuery, 'id', '3');
	querySetCell(myQuery, 'user', 'Steve');
</cfscript>


<cfloop query="myQuery">
	#myQuery.id# #myQuery.user#<br />
</cfloop>
```

Unlike the other loops, the `query` attribute takes a pointer to the query as opposed to the value itself (we specified the name of the query as opposed to supplying `#myQuery#`). The code above results in the following output:

```
1 Jeff
2 John
3 Steve
```

`cfloop` can be used to perform conditional looping as well as the item and iterative loops demonstrated above. Conditional looping will execute the code within the body of a `cfloop` as long as a condition evaluates to true. In the following example, we loop over an array as long as it contains items.

```
<cfset myArray = ['Jeff', 'John', 'Steve', 'Julianne'] />


<cfloop condition="#arrayLen(myArray)#">
	Current length = #arrayLen(myArray)#<br />
	<cfset arrayDeleteAt(myArray, 1) />
</cfloop>
```

Remember in the Decision Making chapter that ColdFusion is dynamically typed, so any positive integer will evaluate to the Boolean value of `true`. As long as the condition remains `true` (that the array has a length), the loop will continue to execute. As soon as the condition is `false`, the loop will stop executing. We will never receive an error from this loop that the array contains no items, since it will not execute if the condition is not true.

> **Important Note**: If the upper limit of the iterative loop is never reached, or a condition never becomes `false` in a conditional loop, an infinite loop condition can occur. This simply means that your loop will never finish and program execution will continue forever until the thread runs out of memory. This condition will result in Java heap space error, or a timeout error.

cfscript Looping

Almost all of the same loop constructs are permitted in `cfscript` as are available in CFML. Common `cfscript` loops are demonstrated below.

Iterative Loop over an Array

```
for (i=1;i<=arrayLen(myArray);i++) {
	writeOutput('#i#: #myArray[i]#<br />');
}
```

Array Loop

```
for (item in myArray) {
	writeOutput(#item# & '<br />');
}
```

Loop over a Collection (Structure)

```
for ( key in myStruct) {
	writeOutput('#key#: #myStruct[key]#<br />');
}
```

Loop over a Query (CF9 and Below)

```
for ( i=1;i<=myQuery.recordCount;i++ ) {
		writeOutput('#myQuery.id#: #myQuery.user[i]#<br />');
}
```

Loop over a Query (CF10+)

```
for ( row in myQuery ) {
	writeOutput('#row.id#: #row.user#');
}
```

Flow Control Inside of cfloop

Sometimes it is necessary to end a loop before the condition for loop termination is met. ColdFusion allows this in CFML via the `cfbreak` tag and in `cfscript` using the `break` keyword. A simple example is provided below:

```
<cfloop from="1" to="5" index="i">
	#i#<br />
	<cfbreak />
</cfloop>
```

The output from the code above is:

```
1
```

There are also times when you want to skip processing the current iteration of a loop, perhaps based on a condition, and then continue looping as normal. As of CF9, you can use the `cfcontinue` tag to perform this logic. In `cfscript` (across versions) you can use the `continue` keyword. An example is provided below:

```
<cfloop from="1" to="5" index="i">
	<cfif i MOD 2 EQ 0>
		<cfcontinue />
	</cfif>
	#i#<br />
</cfloop>
```

The code above results in the following output:

```
1
3
5
```

Additional Looping

ColdFusion also allows you to loop over files and date/time types. For more information visit:

- http://help.adobe.com/en_US/ColdFusion/10.0/CFMLRef/WSc3ff6d0ea77859461172e0811cbec22c24-71a7.html
- http://help.adobe.com/en_US/ColdFusion/10.0/CFMLRef/WSc3ff6d0ea77859461172e0811cbec22c24-71a9.html

Hands On 7

By Simon Free

In this hands on, we will perform some simple looping using a list and output it to the page.

Tags Used: <cfset>, <cfloop>, <cfoutput>

1. Open up the `/www/resume.cfm` file in your code editor.
2. First, let's create the list. On line 1, create a `<cfset>` tag called 'mySkills' with the following value: ColdFusion,HTML5,CSS3,MySQL,jQuery. Your code should look similar to this:

```
<cfset mySkills = "ColdFusion,HTML5,CSS3,MySQL,JQuery" />
```

3. Locate the `Skills Listings` comment.
4. Next, loop over the list using a `<cfloop>`. After the opening `<ul>` tag below the comment, on or around line 154, create an opening `<cloop>` tag with the following attributes:
   - **List:** #mySkills#
   - **Index:** skill
5. On the next line, write the following line of code:

```
<li class="#skill#" id="#skill#">#skill#</li>
```

6. After this line, create a closing `</cfloop>` tag.
7. Your code should look similar to this:

```
<cfloop list="#mySkills#" index="skill">
    <li class="#skill#" id="#skill#">#skill#</li>
</cfloop>
```

8. Delete the remaining `<li>` tags.
9. Wrap the `<cfloop>` tag with an opening and closing `<cfoutput>` tag.
10. Your final code should look similar to this:

```
<div class="skills">
     <ul>
          <cfoutput>
               <cfloop list="#mySkills#" index="skill">
                    <li class="#skill#" id="#skill#">#skill#</li>
               </cfloop>
          </cfoutput>
     </ul>
</div>
```

11. Reload `resume.cfm` in your browser and confirm that output under 'My Skillset' is outputting correctly.

Hands On 8

By Simon Free

In this hands on, we will create a more complex loop that uses an array of structures.

Tags Used: <cfscript>, <cfloop>, <cfoutput>
Functions Used: arrayAppend

1. Open the `/www/portfolio.cfm` file in your code editor.
2. At the top of the file, create open and closing `<cfscript>` tags.
3. Inside the `<cfscript>` tags create a variable `myPortfolio` and instantiate an empty array.
4. Your code should look similar to this:

```
<cfscript>
    myPortfolio = [];
</cfscript>
```

5. Once you have an empty array, you need to populate it with some data. Below the variable declaration, write the following line of code:

```
arrayAppend(myPortfolio,{title='',website='',image='',description=''});
```

6. Add this line 2 more times.
7. Provide information for all 3 structures. Your code should look similar to this:

```
<cfscript>
    myPortfolio = [];
    arrayAppend(myPortfolio,{title='Title 1',website='http://www.
website1.com',image='portfolio1.png',description='Description 1'});
    arrayAppend(myPortfolio,{title='Title 2',website='http://www.
website2.com',image='portfolio2.png',description='Description 2'});
    arrayAppend(myPortfolio,{title='Title 3',website='http://www.
website3.com',image='portfolio3.png',description='Description 3'});
</cfscript>
```

8. Go to the comment `Start Portfolio`.

9. Once the array has been populated you can loop over it and output its data. Below the comment, create a <cfloop> tag with the following attributes:
    - **array**: #myPortfolio#
    - **index**: portfolio
10. Add a closing </cfloop> tag after the first closing tag.
11. Replace the value of the href attribute with #portfolio.website#.
12. Replace the value of the title attribute with #portfolio.title#.
13. Replace the src attribute of the <img> tag with /assets/images/portfolio/#portfolio.image#.
14. Replace the contents of the <h5> tag with #portfolio.title#.
15. Replace the contents of the <p> tag with #portfolio.description#.
16. Delete the remaining 2 <li> tags that are outside of the <cfloop>.
17. Wrap the <cfloop> tag with an opening and closing <cfoutput> tag.
18. Your code should look similar to this:

```
<ul id="portfolio-list">
    <!-- Start Portfolio -->
    <cfoutput>
        <cfloop array="#myPortfolio#" index="portfolio">
            <li>
                <div class="left">
                    <a href="#portfolio.website#"
title="#portfolio.title#" class="viewDetail ">
                        <img src="images/assets/small/#portfolio.
image#"   alt=" " border="0" />
                        <h5>#portfolio.title#</h5>
                    </a>
                </div>
                <div class="right">
                    <p>
                        #portfolio.description#
                    </p>
```

```
                    </div>
                </li>
            </cfloop>
        </cfoutput>
</ul>
```

19. Load the `/www/portfolio.cfm` page in your browser; notice that the contents from the struct is being displayed.

Homework

- Update the contents of the structs with more relevant information.
- Add 2 new portfolio items.

4

Data Handling

By Dave Ferguson

About Dave Ferguson

Dave has been working in information technology for many, many, years. He has spent the majority of that time specializing in large enterprise-class web applications. Dave runs his own ColdFusion blog, http://blog.dkferguson.com where he posts interesting things he comes across. He is also the founder and co-host of the CFHour Podcast http://www.cfhour.com. The podcast, just like his blog, is mostly focused on ColdFusion but dabbles in to other technologies. Dave is also a member of the Adobe Community Professional group.

Databases

One of the key parts of any application is its ability to do CRUD (Create, Read, Update, and Delete) operations. ColdFusion has the ability to read a variety of data sources with minimal effort; there is built in support for over ten different databases. Among the supported databases, you will find MS SQL, MySql, Oracle, DB2, and Sybase. There are slight sql syntax differences when interacting with the database; however, the same code can be used against different databases without any changes.

> **Note**: This chapter will not teach you how to write sql. However, it will show you how to take the sql skills you have and incorporate them into your code. Alternatively, if you are using ColdFusion Builder, you can use the query builder to assist you with query creation.

Query Basics

A data source needs to be configured in the administrator. The data source contains all the connection information that ColdFusion needs to communicate with your database. Rather than providing all that information with each database call, ColdFusion allows you to defined a data source, providing it a name, and use the name as a reference for all your queries.

If you installed the samples when you installed ColdFusion, then there are some for you to play with already. Once you have the data source created, you can start to do CRUD operations.

Creating a Data Source:

- Login to ColdFusion Administrator.
- On the left menu under "Data & Services", select "Data Sources".
- Under "Add New Data Source", enter a name (this will be the name you will use in your code to reference the data source) and select a driver type. The type must match the database type you are using.
- Click "Add" to submit the form.
- The next screen will vary depending on data source type. The following steps are based on selecting "MySQL (4/5)" as a type.
- Enter database name (name of your database on your server).
- Enter Server (IP address or domain name of server).
- Port is defaulted but change it if necessary.
- Enter username (this is the login that will be tied to all database actions from your application).
- Enter Password.
- Enter a description if necessary.
- This is not a requirement, but you can click on "Advanced Settings" and make any adjustments there.
- Click "Submit" to save.
- The data source will validate; if there are any errors they will be reported on screen.
- If necessary, correct any errors until the data source validates upon save.

Basic Query:

```
<cfquery name="myQuery" datasource="cfartgallery" >
    SELECT * FROM artists
</cfquery>
```

The query result is stored in the variable with the same name as they query name. To see the result of the query, you can just dump the result.

```
<cfdump var="#myQuery#" />
```

The dump will contain information on caching status, execution time, actual sql statement, and the entire result set. Don't let the dump confuse you; the result is not stored in a "resultset" variable. That is just how `cfdump` shows it.

There is other information about the query that `cfdump` does not display that are part of the `myquery` object, such as `columnlist` and `recordcount`. These can be referenced by using `myquery.columnlist` or `myquery.recordcount`.

> **Performance Tip**
> The query should only return the columns you need; this speeds up the query and return processing. For example, since the output in the above example only used email, firstname, and lastname, the query should be changed from "SELECT *" to "SELECT email, firstname, and lastname".

Query Output

While `cfdump` is a great way to make sure the query is working, it isn't very useful output to a user. Using `cfoutput`, we now gain display control and can display what we want how we want it.

```
<cfoutput query="myQuery">
    #myquery.CurrentRow# - #myquery.email# - #myquery.firstname# -
#myquery.lastname#<br />
</cfoutput>
```

There are times where you may wish to process the query return but not directly output it. To accomplish this, use `cfloop`. This tag allows you to loop over the result set and perform operations per row (and output them too). It is also useful in the event that you wish to loop over a query result and there is already an outer `cfoutput` in the code. The syntax looks identical to `cfoutput`.

```
<cfset artistArray = [] />
<cfloop query="myQuery">
    <cfset arrayappend(artistArray, myquery.lastname & ', ' &
myQuery.firstname) />
</cfloop>
```

Query Output Grouping

It is also possible to nest `cfoutput` tags so you can group like data. This creates a nested loop where the outer loop is iterated once per group and the inner is looped once for each item in the group. While this sounds great on the surface, it does have a drawback: the data must be pre-sorted before the `cfoutput` for the grouping to work correctly. In the example below, an order by is added to the query, which makes sure that the rows come back in the correct order. The `cfoutput` tag now has a `group` attribute. The value of this attribute must always be a column name of the query. Inside the `cfoutput` is another `cfoutput`. This is the inner loop that will output once per row in the group. The outer will only output once.

```
<cfquery name="myQuery" datasource="cfartgallery">
    SELECT * FROM art
    ORDER BY issold
</cfquery>

<cfoutput Query="myQuery" group="issold">
    <p>
    Sold ?: #YesNoFormat(myQuery.issold)#:<BR>
    <blockquote>
         <cfoutput>
             #myQuery.artname#: #DollarFormat(myQuery.price)#<br />
         </cfoutput>
    </blockquote>
    </p>
    <hr />
</cfoutput>
```

Query Param

One of the biggest attacks against a system is sql injection. However, an easy way to prevent it is to use `cfqueryparam`. This sets parameters for the query input, thus making injection nearly impossible. `cfqueryparam` should be used anywhere you have query input. `cfqueryparam` is not exclusive to select where clauses; it should also be used for inserts, updates, or any query that has dynamic input.

`cfqueryparam` is also useful in doing data length validation as well as data type. For example, if your queryparam is set as an integer and a string is passed, an error will be thrown. The same is true when a string is longer than the set max length.

Note: cfqueryparam should be used ALWAYS without exception.

Query Example:

```
<cfquery name="myQuery" datasource="cfartgallery">
    SELECT * FROM artists
    WHERE firstname = <cfqueryparam value="#form.name#" cfsqltype=
"cf_sql_varchar" />
</cfquery>
```

If you have debugging enabled, the query output in the debug info will look a little different:

```
myQuery (Datasource=cfartgallery, Time=1ms, Records=1) i

    SELECT * FROM artists
    WHERE firstname = ?

Query Parameter Value(s) -
Parameter #1(cf_sql_varchar) = Austin
```

Notice the "?" in the where clause? That is the parameter that was created using cfqueryparam. You can also use it to replace a list in an "in" clause:

```
<cfquery name="myQuery" datasource="cfartgallery">
    SELECT * FROM artists
    WHERE artistid IN (<cfqueryparam value="#form.ids#" list="true" />)
</cfquery>

myQuery (Datasource=cfartgallery, Time=2ms, Records=4) i

SELECT * FROM artists
        WHERE artistid IN (?,?,?)
Query Parameter Value(s) -
Parameter #1(CF_SQL_CHAR) = 1
Parameter #2(CF_SQL_CHAR) = 2
Parameter #3(CF_SQL_CHAR) = 3
```

Query Caching

Query caching is very easy to implement. Caching will be covered in detail in a later chapter; for now, this chapter will just touch on one way to cache here: `cachedwithin`. This type of cache states that the query result will be stored and reused for the specified duration. Using the previous query example, caching the result can be done like this:

```
<cfquery name="myQuery" datasource="cfartgallery" cachedwithin=
"#createTimespan(0,1,0,0)#">
    SELECT * FROM artists
</cfquery>
```

This would cause the query result to cache for 1 hour. Caching is based on the query syntax, not the query name. As long as the syntax doesn't change, the cache will be used. However, if you have a very dynamic query, caching can be a challenge. For example, if the below is your query, you might not want to cache it because the sql syntax can vary widely from request to request. Using `cfqueryparam` does not help in caching at the ColdFusion server level. However, it does assist with database query plans so there is some caching benefit.

```
<cfquery name="myQuery" datasource="cfartgallery" cachedwithin=
"#createTimespan(0,1,0,0)#">
    SELECT firstname, lastname, email FROM artists
    WHERE firstname = <cfqueryparam value="#form.firstname#"
cfsqltype="cf_sql_varchar" />
</cfquery>
```

Dynamic Queries

Queries are not always straightforward. There may be occurrences in which a where clause or column list needs to be decided upon at runtime. Thankfully, `cfquery` allows for wide range of dynamics to create the query. Think of it like generating browser output without any markup or `cfoutput` tags, like a paragraph of text. Knowing that, we can use a variety of ways to generate a dynamic query.

cftags inside cfquery

```
<cfquery name="myQuery" datasource="cfartgallery">
    SELECT firstname, lastname, email FROM artists
    WHERE 1 = 1
     <cfif structkeyExists(form, 'firstname') and len(form.firstname)>
        AND firstname = <cfqueryparam value="#form.firstname#"
cfsqltype="cf_sql_varchar" />
    </cfif>
    <cfif structkeyExists(form, 'lastname') and len(form.lastname)>
        AND lastname = <cfqueryparam value="#form.lastname#"
cfsqltype="cf_sql_varchar" />
    </cfif>
</cfquery>
```

Generating sql string outside query

```
<cfset query = 'SELECT firstname, lastname, email FROM artists' />

<cfquery name="myQuery" datasource="cfartgallery">
    #query#
</cfquery>
```

Limiting Result Counts

Returning thousands of rows from a query can be very inefficient. It is good practice to only return the number of rows you are going to deal with. Reducing the result set to a defined number of rows can be done in a variety of ways: You could add a `maxrows` to the output; you could add a max count to the sql statement itself; or you could add a `max` to the `cfquery` tag itself. These are all valid ways to handle it, but choosing which way to handle the result can be a challenge. Adding a limiter to the `cfquery` tag or in the sql statement limits the rows from the databases, but the max rows on the output just reduces the output. Thus, if you query returned 10,000 rows, your output could have a max of 100, and only display the first 100.

```
<cfoutput query="myQuery" maxrows="100">
    #myquery.CurrentRow# - #myquery.email# - #myquery.firstname# -
#myquery.lastname#<br>
</cfoutput>
```

The issue is that there were 9,900 rows of ignored data. The better approach may be to have the database only return 100 rows, which can be done by setting a max on the initial query.

```
<cfquery name="myQuery" datasource="cfartgallery" maxrows="100">
    SELECT firstname, lastname, email FROM artists
</cfquery>
```

This way is much more efficient, as less data is transmitted from database to server. It also helps speed things up as ColdFusion no longer has to deal with query data that is not being used.

Insert and Identity Retrieval

When inserting records, you are going to want to know the ID of the inserted item more often than not. Typically, this is achieved by doing the insert, then doing a select and getting the highest ID in the table. While this may work, it is very problematic and may not yield the expected results. Fortunately, there is an alternative that works very well and yields the correct results. Look at the example below:

```
<cfquery result="qryResult" datasource="cfartgallery" >
    INSERT INTO art
    (
        artistID, artName, description, isSold, largeImage, medaID,
price
    )
    VALUES
    (
        <cfqueryparam cfsqltype="CF_SQL_INTEGER" value="1" />,
        <cfqueryparam cfsqltype="CF_SQL_VARCHAR" value="Test Item" />,
        <cfqueryparam cfsqltype="CF_SQL_VARCHAR" value="Test Desc" />,
        <cfqueryparam cfsqltype="CF_SQL_INTEGER" value="1" />,
        <cfqueryparam cfsqltype="CF_SQL_VARCHAR" value="img.png" />,
        <cfqueryparam cfsqltype="CF_SQL_INTEGER" value="1" />,
        <cfqueryparam cfsqltype="CF_SQL_INTEGER" value="1" />
    )
</cfquery>

<cfdump var="#qryResult#" />
```

Notice in the above a `result` attribute was added and given a value of "qryResult". This variable is a dumpable structure of data. One of the parts of the data is the ID of the inserted row. It will also contain the params (if any), execution time, and the sql statement. Depending on your database, the identity variable may be different; dump the result first to see what the column is.

Queries in cfscript

There is a way to run queries if you really like writing all your code in script syntax; it is slightly more complicated, but has all the same functionality that the tag based syntax has. There are many ways to implement using queries in script. The example below is just one of those ways. The one main

difference is that with script based query, you actually have to tell ColdFusion to execute the query; the query doesn't just run by creating it. Also, it will be necessary to get the query result by using the `getResult()` function.

Basic Example

```
<cfscript>
    myQry = new Query();
    myQry.setDatasource("cfartgallery");
    myQry.setSQL("SELECT firstname, lastname, email FROM artists");
    myQuery = myQry.execute();
    writeDump(myQuery.getResult());
    writeDump(myQuery.getPrefix());
</cfscript>
```

Streamlined Example

Using chaining and adding params to the query initialization, the code can be reduced to a single line:

```
<cfscript>
    myQueryResult = new Query(sql="SELECT firstname, lastname, email FROM
artists", datasource="cfartgallery").execute().getResult();
    writeDump(myQueryResult);
</cfscript>
```

Using Query Params

Just like the tag based version, you can easily add query params to the query. For the script syntax you have to define a matching pair, which is the param name in the query; the added param must match. If they don't match, an error will be thrown.

```
<cfscript>
    myQry = new Query();
    myQry.setDatasource("cfartgallery");
    myQry.setSQL("SELECT artname, description FROM art WHERE issold =
:sold");
```

```
    myQry.addParam(name: "sold", value: "1", cfsqltype: "CF_SQL_INT");
    myQuery = myQry.execute();
    writeDump(myQuery.getResult());
    writeDump(myQuery.getPrefix());
</cfscript>
```

XML

Dealing with XML in ColdFusion is fairly straightforward. ColdFusion provides many built-in functions for doing all sorts of XML based operations. Some of those functions provide streamlined access to create, read, and search an xml document. You can also validate XML data against a DTD or schema and transform XML using XSLT.

Reading an XML Document

Reading in an existing XML document can be done a few ways. For example, you can read the file then process it as XML, or you can have the XML parser read the file.

This example reads the file in and then passes it to the parser to create an XML object:

```
<cffile action="read" file="#ExpandPath("./order.xml")#" variable=
"myxml" />
<cfset mydoc = XmlParse(myxml) />
```

Alternatively, you can just have the `xmlparse` function read in the file, thus skipping a step:

```
<cfset mydoc = XmlParse(ExpandPath("./order.xml")) />
```

The `xmlparse` function takes the XML document and converts the XML to an object. This now allows you to read the XML as a structure, or depending on XML complexity, an array of structs. You can use `cfdump` on the result of the `xmlparse` to see the object.

```
<cfdump var="#mydoc#" />
```

Dumping the XML will display a short version of the XML. You can click on the dump where it says "[short version]" and it will expand to a long version of the XML. The long version shows how ColdFusion is referencing the XML.

Creating an XML Document

Creating XML in ColdFusion is just as easy as generating HTML output. The `cfxml` tag is used to generate the XML output:

```
<!---Get some data from, in this case, the database--->
<cfquery name="getData" datasource="cfartgallery" >
    SELECT artistID, artID, artName, description, isSold, price,
largeImage FROM art
```

```
    WHERE artistid = 1
</cfquery>

<!---Process the query result and generate xml--->
<cfxml variable="artxml">
<art artistid="<cfoutput>#getdata.artistid#</cfoutput>">
    <cfoutput query="getData">
        <piece id="#getData.artid#" available="#getdata.isSOLD#">
            <artname>#getData.artname#</artname>
            <description>#getData.description#</description>
            <image>#getData.largeImage#</image>
            <price>#getData.price#</price>
        </piece>
    </cfoutput>
</art>
</cfxml>
```

Write XML Out to a File

```
<cffile action="write" file="#expandpath('./out.xml')#" output=
"#artxml#" />
```

Dump Output to the Browser

```
<cfdump var="#artxml#" />
```

Parsing XML

ColdFusion provides a single, yet powerful, function to parse XML. This function can read a file, URL, or a string containing XML. It can also validate the XML based on a DTD or schema.

Basic Parse using an XML Document

```
<cfset mydoc = xmlParse(ExpandPath("./order.xml", true)) />
```

When parsing the XML, you can have the parsing function maintain case of the XML nodes or ignore case altogether. If it is necessary to maintain case, then referencing the XML content will change slightly. Assuming that the above XML parse (with case sensitivity turned on) returned the XML below, let's go over some of the nuances.

```
<myArt>
    <art>
        <itemName Type="1">art name</item>
    </art>
</myArt>
```

In the above example, the root note is "myArt". Typically you can reference the root like this:

```
mydoc.myart
```

But with case sensitivity on this will cause an error, even if the root node is lowercase. The correct way to reference it:

```
mydoc.xmlRoot
```

The same is also true with nodes and attributes. With case sensitivity off, this is perfectly valid:

```
mydoc.myart.art[1].itemname.xmltext
```

But with case sensitivity on, the same is accomplished using associative array (bracket) notation.

```
mydoc.XMLRoot.xmlChildren[1]["itemName"]
```

The bracket syntax is not exclusive to case sensitivity. You can use it even if case sensitivity is off.

XML Datatype and Strings

At its core, XML is just a fancy string. The `xmlparse` function takes the string and converts it to something useful. The XML string then becomes an XML datatype in ColdFusion. This datatype is an array of structures and arrays. When saving XML to a document you need to save the XML as a string, not as the XML datatype. Converting the XML to a string is as simple as parsing the XML to begin with; all it takes is a single function call.

```
<cfset xmlString = toString(myDoc) />
```

Looping over XML

Looping over data in ColdFusion is not very complicated; however, looping over XML can be a little troublesome. This is due to nesting of nodes, attributes, and case sensitivity. Below are two examples of looping over XML. The XML source for this was the same generated XML from the create XML example previously reviewed.

The first example takes into account that we know the XML structure. This makes it fairly easy to deal with as we can directly reference the structures in the code:

```
<cfloop array="#artxml.art.xmlchildren#" index="i" >
    <cfoutput>
        name: #i.artname.xmlText#<br/>
        price: #i.price.xmlText#<br/>
        Available? #yesNoFormat(i.xmlAttributes["available"])#
    </cfoutput>
    <br />
</cfloop>
```

The above example loops over the `xmlChildren` of the art node in the XML. By doing it this way, we can treat all the children as an array. This then allows us to isolate each node and reference each item in the node. We can also reference the attributes and output them as well.

However, what if we didn't know the XML structure or wanted to process it as if we didn't? Using cfdump would allow us to see the structure and write code accordingly, but that is not always an option. Also, what if you want to do something with the XML node names and/or content? The example below is written as though nothing is known about the XML structure. While it is not a complete example, you should get the idea.

```
<cfloop array="#artxml.xmlRoot.xmlchildren#" index="i" >
    <cfoutput>
        <cfif structKeyExists(i, "xmlattributes")>
            <cfloop collection="#i.xmlattributes#" item="a" >
                #a#: #i.xmlAttributes[a]#<br/>
            </cfloop>
        </cfif>
        <cfloop array="#i.xmlchildren#" index="x">
```

```
        #x.xmlName#: #x.xmlText#<br/>
        </cfloop>
    </cfoutput>
    <br />
</cfloop>
```

This example uses the underlying XML structure to get attributes, node names, and note text. It works just like the previous example, but it outputs all nodes for each item based on their node name. This is done by looking at each node and treating its children as an array. It also checks the root node for any attributes and outputs them as necessary.

Setting XML Node Value

Setting the value of a node is just about as easy as outputting it. Taking the example XML created previously as a base, let's update the prices.

```
<cfloop array="#artxml.art.xmlchildren#" index="i">
    <!---price increase by 20% --->
    <cfset i.price.xmlText = i.price.xmlText*.2 />
</cfloop>
```

Setting an attribute works about the same. However, we look at the attributes of the node using an associative array and change the available value instead.

```
<cfloop array="#artxml.art.xmlchildren#" index="i" >
    <!--- make all items available --->
    <cfset i.xmlAttributes['available'] = 1 />
</cfloop>
```

Convert XML to Query Result Set

Sometimes dealing with large XML files can be cumbersome. Depending on the content of the XML, you can convert it to a query result type structure that will make it a little easier to reference. This is done by using a combination of query, XML, and array functions. Using the previously generated XML, we will take it and convert it to a query:

```
artQuery = QueryNew("artistid, artid, name, description, image, price,
sold");
queryAddRow(artQuery, arraylen(myDoc.art.piece));


for (i = 1; i lte arraylen(myDoc.art.piece); i++){
    thisItem = mydoc.art.piece[i];
    QuerySetCell(artQuery, "artistid", mydoc.art.xmlAttributes.artistid,
i);
    QuerySetCell(artQuery, "artid", thisItem.xmlAttributes.id, i);
    QuerySetCell(artQuery, "sold", thisItem.xmlAttributes.available, i);
    QuerySetCell(artQuery, "name", thisItem.artname.xmltext, i);
    QuerySetCell(artQuery, "description", thisItem.description.xmltext,
i);
    QuerySetCell(artQuery, "image", thisItem.image.xmltext, i);
    QuerySetCell(artQuery, "price", thisItem.price.xmltext, i);


}


writedump(artQuery);
```

The above example will dump out a query object. You can then use the same query object in a `cfoutput` or in a `cfloop`.

Searching XML

ColdFusion 10 supports XPath 2.0 for `XMLSearch()` and `XMLTransform()`. This allows for using paramaters when searching XML you can build complicated searches with ease. The example below is just one of many ways to search XML. In the example below the first step is to load and parse the XML (using xml created above). Then a param is created for the search. Next the search is performed using the param. The search is looking for a node value of "Mary" where the value resides in the piece/artname path. The "//" before piece means that the node piece can have any parent. The result of the search is an array of results. Also, the result is just the node found. So, in the dump the return var must be treated as an array. It is also necessary to get the parent of the found node so we get the entire art node for the item.

```
<cffile action="read" file="#expandPath('./out.xml')#" variable="myxml" />


<cfset mydoc = XmlParse(myxml) />
<cfscript>
    params = structNew();
    params["artname"] = "Mary";
</cfscript>


<cfset searchRes = xmlSearch(myxml, '//piece/artname[. = $artname]',
params) />


<cfdump var="#searchRes[1].xmlParent#" />
```

JSON

JSON or, JavaScript Object Notation, is a simple way for applications to share data. It is commonly used to send data from a server to a browser client, but it can be used for much more. It closely resembles XML without the markup or, more accurately, a struct of structs. Creating JSON data in ColdFusion is just about as easy as setting a variable.

Create JSON String

To create JSON, you first need to start with a structure. For example:

Structure

```
<cfset myStruct = {
    items: {
        item1:  {name: 'something', price: 1000},
        item2:  {name: 'something else', price: 12.50},
        item3:  {name: 'something more', price: "1,240.10"}
    },
    users: {
        user1:  {id: 1, email: 'none@none.net'},
        user2:  {id: 2, email: 'none@none.net'},
        user3:  {id: 3, email: 'none@none.net'}
    }
} />
```

Then convert it to JSON:

```
<cfset myJsonVar = serializeJSON(myStruct) />
```

End Result:

```
{"USERS":{"USER3":{"ID":3,"EMAIL":"none@none.net"},"USER2":{"ID":2,"EMAI
L":"none@none.net"},"USER1":{"ID":1,"EMAIL":"none@none.net"}},"ITEMS":{"
ITEM2":{"PRICE":12.50,"NAME":"something else"},"ITEM3":{"PRICE":"1,240.1
0","NAME":"something more"},"ITEM1":{"PRICE":1000,"NAME":"something"}}}
```

Notice, however, that all the names in the name/value pairs have been converted to uppercase. This

is default behavior, but it can be easily remedied. It is caused by how the structure is created. If we alter the structure creation to the following, the names will not change case:

```
"items": {
    "item1":    {name: 'something', price: 1000},
    "item2":    {name: 'something else', price: 12.50},
    "item3":    {name: 'something more', price: "1,240.10"}
},
```

Any name that needs to remain in lower cause would need to be quoted. Leaving it uppercase is just fine, but remember that if the JSON data is being used by JavaScript, then be careful as JS is case sensitive.

Convert JSON to Structure

Converting a JSON string to a ColdFusion structure requires one single function:

```
<cfset myJsonStruct = deserializeJSON(myJsonVar) />
```

Validate String as JSON

Checking to see if a string is in a valid JSON format is simple as well. All it takes is a call to a validator function:

```
<cfdump var="#isJson(myJsonVar)#" />
```

Hands On 9

By Simon Free

In this hands on, we will do a simple database call and output the data.

Tags Used: <cfset>, <cfquery>, <cfloop>

1. Open up the `/www/resume.cfm` file in your code editor.
2. First, replace the list with a call to the database. On line 1, replace the `<cfset>` tag with an open `<cfquery>` tag with the following attributes:
   - **name**: mySkillSet
   - **datasource**: learncfinaweek
3. On the next line enter the following SQL code:

```
SELECT
     name
FROM
     skillset
ORDER BY
     name DESC
```

4. On the next line, write a closing `</cfquery>` tag.
5. Your code should look similar to this:

```
<cfquery name="mySkillset" datasource="learncfinaweek">
     SELECT
          name
     FROM
          skillset
     ORDER BY
          name DESC
</cfquery>
```

6. Please note that the data source you are using was created during the initial setup process for you.

7. Now that you have a query, you need to update the loop to accept the query rather than the list. Locate the opening `<cfloop>` tag on or around line 120 and replace its attributes with:
    - **query**: mySkillset
8. Inside the `<cfloop>` tag, replace all references to skill with `#mySkillset.name#`.
9. Your code should look similar to this:

```
<cfloop query="mySkillset">
    <li class="#mySkillset.name#" id="#mySkillset.name#">
#mySkillset.name#</li>
</cfloop>
```

10. Open up your browser and navigate to the `/www/resume.cfm` page. Confirm that the Skills are still displaying.

Hands On 10

By Simon Free

In this hands on, we are going to create a more complex query and output that to the page.

Tags Used: <cfquery>, <cfoutput>, <cfelse>
Functions Used: dateFormat, len

1. Open up the `/www/resume.cfm` file in your code editor.
2. Below the `mySkillset` query, create another open `<cfquery>` tag with the following attributes:
   - **name**: myResume
   - **datasource**: learncfinaweek
3. After the opening `<cfquery>` tag, provide the following SQL text:

```
SELECT
    title,
    startDate,
    endDate,
    details,
    type
FROM
    resume
ORDER BY
     type,
     endDate,
     startDate
```

4. After the SQL text, add a closing `</cfquery>` tag.
5. Your code should look similar to this:

```
<cfquery name="myResume" datasource="learncfinaweek">
     SELECT
         title,
```

```
        startDate,
        endDate,
        details,
        type
    FROM
        resume
    ORDER BY
        type DESC,
        endDate,
        startDate
</cfquery>
```

6. Locate the `Resume Listings` comment tag on or around line 90.
7. After the comment, add a `<cfoutput>` tag with the following attributes:
   - **query**: myResume
   - **group**: type
8. Add a closing `</cfoutput>` tag after the `End Resume Listing` comment tag.
9. Delete the lines of code from right after the closing `</cfoutput>` tag to the closing `</div>` tag.
10. Your code should look similar to this:

```
<div class="left">
    <!-- Resume Listings -->
    <cfoutput query="myResume" group="type">
        <h2>Work Experience <span> </span></h2>
        <h5>Senior Developer - Google Inc <span>2010 to present </span>
</h5>
        <p>Lorem ipsum dolor sit amet, habitasse pretium dolor sociis.
Nulla et facilisis interdum elit amet erat, consectetuer condimentum
eaque, ante maecenas Suspendisse libero diam.</p>
    <!-- End Resume Listing -->
    </cfoutput>
</div>
```

11. Replace the text "Work Experience" with `#myResume.type#`.
12. After that line, place an opening `<cfoutput>` tag.
13. Note that we are using the group functionality within ColdFusion; this allows sub loops to be performed inside of a `<cfoutput>`. This is the only time that a `<cfoutput>` tag should be nested inside of another `<cfoutput>` tag.
14. Place a closing `</cfoutput>` tag just before the other `</cfoutput>` closing tag.
15. Your code should look similar to this:

```
<div class="left">
    <!-- Resume Listings -->
    <cfoutput query="myResume" group="type">
        <h2>#type# <span> </span></h2>
        <cfoutput>
            <h5>Senior Developer - Google Inc <span>2010 to present
</span> </h5>
            <p>Lorem ipsum dolor sit amet, habitasse pretium dolor
sociis. Nulla et facilisis interdum elit amet erat, consectetuer
condimentum eaque, ante maecenas Suspendisse libero diam.</p>
            <!-- End Resume Listing -->
        </cfoutput>
    </cfoutput>
</div>
```

16. Replace the "Senior Developer –Google Inc" text with `#myResume.title#`.
17. Replace the "2010 to present" text with `#myResume.startDate# to #myResume.endDate#`.
18. Replace the contents of the `<p>` tag with `#myResume.details#`.
19. Your code should look similar to this:

```
<div class="left">
    <!-- Resume Listings -->
    <cfoutput query="myResume" group="type">
        <h2>#type# <span> </span></h2>
```

```
        <cfoutput>
            <h5>#myResume.title# <span>#myResume.startDate# to
#myResume.endDate# </span> </h5>
            <p>#myResume.details#</p>
            <!-- End Resume Listing -->
        </cfoutput>
    </cfoutput>
</div>
```

20. Open up a browser and navigate to the `/www/resume.cfm` page.
21. Note that there are new Work Experience and Education information displaying. Also note that the dates are not displaying in a nice, readable format.
22. Go to the `myResume.startDate` output on or around line 123 and change it to read:

```
#dateFormat(myResume.startDate,"mm/dd/yyyy")#
```

23. Reload the `/www/resume.cfm` page in the browser and note that the start dates are now in a nice, readable format.
24. Go to the `myResume.endDate` output on or around line 123 and make the same update. The code should read:

```
#dateFormat(myResume.endDate,"mm/dd/yyyy")#
```

25. Reload the `/www/resume.cfm` page in the browser. Notice that the date range does not show an end date for the Senior Developer position. This is because there is no end date in the database. Let's make that a bit more readable.
26. Go back to the reference to `myResume.endDate` on or around line 123.
27. After the word "to" and before the start of the `dateformat` function, add the following code:

```
<cfif len(myResume.endDate)>
```

28. After the end of the `dateFormat` function, add the following code:

```
<cfelse>present</cfif>
```

29. Your code should look similar to the following:

```
<div class="left">
    <!-- Resume Listings -->
    <cfoutput query="myResume" group="type">
        <h2>#type# <span> </span></h2>
        <cfoutput>
            <h5>#myResume.title# <span>#dateFormat(myResume.
startDate,"mm/dd/yyyy")# to <cfif len(myResume.
endDate)>#dateFormat(myResume.endDate,"mm/dd/yyyy")#<cfelse>present
</cfif> </span> </h5>
            <p>#myResume.details#</p>
            <!-- End Resume Listing -->
        </cfoutput>
    </cfoutput>
</div>
```

30. Reload the `/www/resume.cfm` page in your browser.
31. You should now see that the Senior Developer entry has a more readable output.

Homework

Update the Blog and Portfolio sections to pull the data from the database.

Hands On 11

By Simon Free

In this hands on, we are going to generate an XML sitemap that is formatted for use with Google Sitemaps.

Tags Used: <cfxml>, <cfoutput>, <cffile>
Functions Used: toString

1. Create a new file in the `/www/` called `generateSitemap.cfm`.
2. On line 1, create an open `<cfxml>` tag and provide the following attribute:
   - **variable**: xmlSiteMap
3. After the opening `<cfxml>` tag enter the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.google.com/schemas/sitemap/0.90">
  <url>
    <loc>http://www.myWebsite.com/</loc>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.myWebsite.com/about.cfm</loc>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.myWebsite.com/resume.cfm</loc>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.myWebsite.com/portfolio.cfm</loc>
    <priority>0.5</priority>
  </url>
```

```
  <url>
    <loc>http://www.myWebsite.com/contact.cfm</loc>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.myWebsite.com/blog.cfm</loc>
    <priority>0.5</priority>
  </url>
</urlset>
```

4. After this code, enter a closing `</cfxml>` tag.
5. Your code should look similar to this:

```
<cfxml variable="xmlSitemap">
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.google.com/schemas/sitemap/0.90">
  <url>
    <loc>http://www.myWebsite.com/</loc>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.myWebsite.com/about.cfm</loc>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.myWebsite.com/resume.cfm</loc>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.myWebsite.com/portfolio.cfm</loc>
    <priority>0.5</priority>
```

```
  </url>
  <url>
    <loc>http://www.myWebsite.com/contact.cfm</loc>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.myWebsite.com/blog.cfm</loc>
    <priority>0.5</priority>
  </url>
</urlset>
</cfxml>
```

6. After the closing `</cfxml>` tag, create an open `<cfoutput>` tag.
7. On the line after the `<cfoutput>` tag, write `#toString(xmlSiteMap)#`.
8. After the `toString` line of code, write a closing `</cfoutput>` tag.
9. Open up a browser and navigate to `/www/generateSiteMap.cfm`.
10. View the source of the page and you will see the generated XML.
11. Return to your code editor and remove the `<cfoutput>` tags and their contents.
12. After the closing `</cfxml>` tag, create a `<cffile>` tag with the following attributes:
    - **action**: write
    - **file**: #expandPath('./sitemap.xml')#
    - **output**: #toString(xmlSiteMap)#
13. Your code should look similar to this:

```
<cffile action="write" file="#expandpath('./sitemap.xml')#"
output="#toString(xmlSitemap)#" />
```

14. After the `<cffile>` tag, create a `<p>` tag. Provide the text "File created!" and place a closing `</p>` tag after the line of text.
15. In your browser, reload the `/www/generateSiteMap.cfm` page and confirm that you see the message "File Created!"
16. Navigate to `/www/sitemap.xml` and confirm that the XML file was created and that all your XML nodes are displayed.

5

Code Reuse

By Emily Christiansen

About Emily Christiansen

Emily Christiansen is a software engineer at American Public Media. She works on the Public Insight Network, a platform for newsrooms to engage their sources in a meaningful way. Emily loves learning new programming languages as well as teaching others. She began working with ColdFusion in 2009 and has been a frequent speaker at conferences ever since. Her main interests are code reuse and refactoring. In addition to programming, Emily also enjoys singing, reading, zumba, and video games.

One acronym that is thrown around in programming circles is DRY: Don't Repeat Yourself. Does anyone enjoy reading a repetitive book? Listening to the same five songs on the radio station? How about writing the same piece of code, over and over again? Performing the same task over and over again is mind-numbing and soul-crushing work. When working in a large system, it's much more fun to write a piece of code in such a way that it can be used multiple times.

Now that some of the basics of programming with ColdFusion have been covered, we can take a moment to discuss code reuse. Writing code that can be used again is an important skill for developers to master; not only does it save time, but it forces the developer to structure their code such that individual routines are isolated and labelled properly. Doing this makes the code easier to read and interpret. In this chapter, simple examples will be used; you can extrapolate from these examples and imagine the effects on a larger scale.

Let's pretend I have a ColdFusion page. The function of this page is to display a greeting. The code would look something like this:

```
<cfparam name="firstName" default="Emily" />
<cfparam name="lastName" default="Christiansen" />

<cfset fullName = firstName & " " & lastName />

<cfoutput>
    Hello, #fullName#!
</cfoutput>
```

This code doesn't do much; it takes the first name and last name, concatenates them to make the full name, then outputs it to the screen. What if we wanted to isolate that piece of code that makes the full name and use it over and over again throughout the system? ColdFusion offers several ways to structure your code for reuse. As the language has grown and evolved over the years, it has gone from simple inclusion of files, to making individual functions, and finally embracing object-oriented programming principles. It also offers the ability to create your own custom tags. When discussing code reuse, functions are the simplest place to start.

Functions

Functions are reusable blocks of code that have a name, may take in arguments, perform some operations on those arguments, and may return a new value. Breaking complex tasks into smaller, clearly labelled functions will also make them easier to read and comprehend. Functions can be thought of as building blocks for a program.

In ColdFusion, functions are created using the cffunction tag. A simple function stub would look like this:

```
<cffunction name="myFunction" output="false" access="public"
returnType="void">

</cffunction>
```

Obviously, this function does nothing right now. Before moving on let's take a moment and go through the tag's attributes.

Name: The name of the function. This should be short, but as descriptive as possible.

Output: In some cases you may want your function to be processed as if it were in a tag, which would display any whitespace and output. If so, set this attribute to true. Most functions are used for processing, not display, so you'll probably want to use false most of the time.

Access: This controls who can see your function. If it's public, it can be called from anywhere else in the system. If it is private, only functions within the same component can call the function. Package visibility is similar to private, but allows components that are within the same package to use those functions. Remote grants webservices the ability to call that function. Components will be discussed later in the chapter.

ReturnType: This describes what type of value is returned. If the function returns nothing, this is set to void.

Now that the stub of the function is created, it's time to make it do something. Let's say we have the name of a person stored in our database. We want to display the person's first and last name in the interface, but their name is stored in two different fields, first_name, and last_name.

If we already have the first and last names stored in variables we can pass them into the function as an argument. We can tell our function which arguments to expect by using the cfargument tag.

For example, the first name would look like this:

```
<cfargument name="firstName" type="string" required="false" default="" />
```

As you can see the tag takes several attributes of its own.

Name: Clear and concise names of your argument.

Type: Denotes the type for the argument. In our example above, it is a string.

Required: Denotes whether or not the argument is required. If you set this to false, make sure you perform checks to ensure that it exists before using it!

Default: If the argument is not required and not passed in, this value will be used. This eliminates the problem of using an argument that doesn't exist. It will not be used if the argument is required.

The function stub now looks like the example below once we add the arguments:

```
<cffunction name="getFullName" output="false" access="public"
returnType="void">
     <cfargument name="firstName" type="string" required="false"
default="" />
     <cfargument name="lastName" type="string" required="false"
default="" />


</cffunction>
```

Take note that the name of the function was changed. The name is much more descriptive of what it actually does; this is an example of refactoring, the process of refining code.

If we want to access the firstName and lastName variables, we will need to look in the arguments scope. This is one of the many scopes provided by ColdFusion. Variables in the arguments scope exist only for the function in which they are declared. You can access them by using arguments.yourVariableHere.

Let's put this to use in the function:

```
<cffunction name="getFullName" output="false" access="public"
returnType="void">
    <cfargument name="firstName" type="string" required="false"
default="" />
    <cfargument name="lastName" type="string" required="false"
default="" />


    <cfset fullName = arguments.firstName & " " & arguments.lastname />


</cffunction>
```

Even though the string is constructed, the fullName variable does not belong to a scope. We need to make sure that only this function can access fullName, so we will var scope it like so:

```
<cfset var fullName = arguments.firstName & " " & arguments.lastName />
```

Now that the new variable is scoped, we can now return the variable. To do so, use the cfreturn tag. Since the function is now returning something, we also need to change the returnType attribute in the cffunction tag. Because we are only returning a string, we will just set it to string. Functions can return pretty much anything you want them to, including structs and objects.

In the end, the function ends up looking like this:

```
<cffunction name="getFullName" output="false" access="public"
returnType="string">
    <cfargument name="firstName" type="string" required="false"
default="" />
    <cfargument name="lastName" type="string" required="false"
default="" />

    <cfset var fullName = arguments.firstName & " " & arguments.lastname
/>

    <cfreturn fullName />
</cffunction>
```

The function is now built, albeit a very simple one. It is concise and clearly labelled, meaning that it is easy to read and reuse elsewhere in the system.

Now, the question is how to allow people to use it. Fortunately, calling functions in ColdFusion is only a little different than setting up any other variable. We use the cfset tag.

```
<cfset fullName=getFullName(firstName="Emily", lastName="Christiansen")/>
```

When it comes to passing in arguments, there are a few ways you can do it. Above, the arguments are explicitly named going into the function. This is particularly useful for functions that have a large number of arguments, as it is far easier to keep track of what actually got passed in. Alternatively, something like this could have been done:

```
<cfset fullName = getFullName("Emily", "Christiansen") />
```

This approach relies on the arguments being passed in the same order in which they are specified in the method signature above. It can get very confusing for functions with large numbers of arguments.

Another option is available if working within one function and sending the same arguments to a helper function. For example, the getFullName function could be used by another, larger function. Let's call it getGreeting:

```
<cffunction name="getGreeting" output="false" access="public"
returnType="string">
    <cfargument name="firstName" type="string" required="false"
default="" />
    <cfargument name="lastName" type="string" required="false"
default="" />

    <cfset var fullName = getFullName(argumentCollection=arguments) />

    <!--- We can do more stuff with the fullName variable here --->

</cffunction>
```

In the above example, argumentCollection = arguments was used to pass the same arguments from one function to another. This is very handy when passing a bunch of arguments to another functions with the same argument names. This is much faster than typing out the names of each and every argument.

In order to call a function, it needs to be put somewhere. There are a number of ways to accomplish this. The simplest way is to put it in a .cfm page and call it.

```
<cfset fullName = getFullName(firstName="Emily",lastName="Christiansen")
/>

<cfoutput>
    Hello, #fullName#!
</cfoutput>

<cffunction name="getFullName" output="false" access="public"
returnType="string">
    <cfargument name="firstName" type="string" required="false"
default="" />
    <cfargument name="lastName" type="string" required="false"
default="" />

    <cfset var fullName = arguments.firstName & " " & arguments.lastname
```

```
/>


    <cfreturn fullName />
</cffunction>
```

Putting the function into the very same .cfm page being used to display our greeting is not very conducive to code reuse.

Including

You can also make a new .cfm file and use it as a function library. You can then use the cfinclude tag to include it on any pages that might need it.

cfinclude only has one attribute, template, that takes the path to the function library file. If the function is placed in a file called greetingCustomizer.cfm, to give our page access to the function we just include our greeting customizer on the page like so:

```
<cfinclude template="path/to/libraries/greetingCustomizer.cfm" />
```

The rest of the page remains unchanged. What it actually does is treat the code that is in the included file as if it were on the page itself. Doing so grants the included template access to all of the variables on the including page. On the surface, it might seem like a good idea, but we will get into why it can actually make code very messy in a little while. In the meantime, once we've included greetingCustomizer.cfm on our page, it will look like this:

```
<cfinclude template="path/to/libraries/greetingCustomizer.cfm" />


<cfset fullName = getFullName(firstName="Emily", lastName="Christiansen")
/>


<cfoutput>
    Hello, #fullName#!
</cfoutput>
```

While cfinclude is an extremely simple approach, it does have some drawbacks. As mentioned above, the included file now has access to every variable on the including page, creating the possibility of introducing unnecessary dependencies between the two files. For example, with cfinclude, declaring arguments isn't really necessary with the function if the values already exist on the page, and less experienced developers might be tempted to leave them out altogether.

displayPage.cfm:

```
<cfset firstName = "Emily" />
<cfset lastName = "Christiansen" />


<cfinclude template="path/to/libraries/greetingCustomizer.cfm" />
```

```
<cfset fullName = getFullName() />

<cfoutput>
    Hello, #fullName#!
</cfoutput>
```

greetingCustomizer.cfm:

```
<cffunction name="getFullName" output="false" access="public"
returnType="string">

    <cfset var fullName = firstName & " " & lastName />

    <cfreturn fullName />
</cffunction>
```

Remember that cfinclude embeds the CFML code in greetingCustomizer.cfm into displayPage.cfm, so writing our code this way is perfectly valid. Unfortunately, it opens the door to a lot of confusion. A developer looking at the function by itself has no idea where the firstName and lastName variables came from. If this function is used anywhere else, those variables will need to be declared on the page or passed into the function. What was done in the example above introduced a dependency; our function no longer stands on its own as an encapsulated block of code. Instead, it can only work in the context of the including page, and the benefits of code reuse are lost. It is very important, then, to make sure that we follow the first example, in which arguments are declared and use the arguments scope to communicate what the function need to our fellow developers.

Custom Tags

For displaying code on the front end, another option is to use a custom tag. Custom tags live in their own directory and are prefixed with "cf_". Putting a custom tag on a cfml page will execute the code contained in the tag as if it had been included on the page itself. Start by creating a new .cfm file to house our custom tag. Let's call it greeting.cfm. First, check the executionMode of the tag. If the executionMode is "start," that means that the tag is being opened. If executionMode is "end," that means that the tag is being closed. This can be checked by setting up an 'if' statement like so:

```
<cfif thisTag.executionMode EQ "start">

<!--- Code to execute when the tag is being opened --->

<cfelse>

<!--- Code to execute when the tag is being closed --->

</cfif>
```

Because our example is relatively simple, we will focus on the code to be executed when executionMode is "start".

As before, we will want to build the user's full name, which means we will probably want to pass those into the tag. Setting attributes on the tag as if it were any other ColdFusion tag will accomplish this; these are then stored in the attributes scope. Access to them is done by using attributes.ourVariableName, as in the example below:

```
<cfif thisTag.executionMode EQ "start">

    <cfset fullName = attributes.firstName & " " & attributes.lastName />

    <cfoutput>
          Hello, #fullName#
    </cfoutput>

</cfif>
```

Once again, a greeting has been constructed and we're ready to call the tag on the display page. Back in displayPage.cfm, do the following:

```
<cf_greeting firstName="Emily" lastName="Christiansen" />
```

Which will output "Hello, Emily Christiansen" in the browser.

You can also use cfimport to bring multiple tags into your page, and put them in their own namespace. The tag takes only two arguments:

prefix: This argument contains the prefix that the developer will use to reference the tag. This will replace the standard "cf_"

taglib: This is the path to the tag library, relative to the web root.

Now we can use the import tag to import several tags, and give them a more descriptive prefix.

```
<cfimport prefix="display" taglib="/path/to/tags" />


<display:greeting firstName="Emily" lastName="Christiansen" />
```

Now it's easy to see that the greeting tag belongs in the package called display, which gives us a clue as to what the tag does. It displays the greeting.

Components

A more modern alternative to cfinclude is to create components. Components in ColdFusion behave similarly to objects in many other programming languages. First, start by creating a new file, called Greeting.cfc. ColdFusion uses .cfc to denote files that are components.

Inside of Greeting.cfc, we will add a cfcomponent tag:

```
<cfcomponent>


</cfcomponent>
```

It doesn't do much, but there it is. If you prefer script over tags, you can write a script-only component, too:

```
component {
}
```

The only caveat to script components is that you can not use any tags inside them, and unfortunately, the cfscript language is currently incomplete. There are some tags that you can not use in script. Adobe is chipping away at the list of unavailable tags with each release of ColdFusion, and there is even a community project that adds additional script functionality by simply adding a few CFCs to the right folder of your ColdFusion installation. One popular alternative is to use a tag-based component, but immediately inside it, switch to script. This gives you the opportunity to write a few tag-based functions that wrap up what you need from CF's built in tags so that you can call those tags from your script:

```
<cfcomponent>
    <cfscript>
    </cfscript>
</cfcomponent>
```

Think of this component like a bucket of code that you can carry around. You can hand it to a friend, and she can execute it, or she can hand it to her friend, or give it back to you. She can put more code in the bucket, or read values from it too. Components have properties, both public and private, and methods (functions). Properties are like labels on your bucket. You can read the label, and you can change its value.

For now, let's add our functions to the component.

```
component {
        public string function getFullName (String firstName, String
lastName) {
            var fullName = arguments.firstName && arguments.lastName;
            return fullName;
        }

        public string function getGreeting (String firstName, String
lastName) {

            var fullName = getFullName(argumentCollection=arguments);
            var greeting = "Hello, " & fullName;

            return greeting;
        }
}
```

Suppose we wanted make different types of greetings. Right now, the getGreeting function only allows us to say Hello. What we can do is create a baseGreeting that will be available to all of the functions in the component.

```
component {
     this.baseGreeting = "Hello, ";

     public string function getFullName (String firstName, String lastName) {
           var fullName = arguments.firstName & " " & arguments.lastName;
           return fullName;
     }
     public string function getGreeting (String firstName, String lastName) {
           var fullName = getFullName(argumentCollection=arguments);
           var greeting = this.baseGreeting & fullName;
           return greeting;
     }
}
```

Now the base greeting is available to all of the functions in the component. Additionally, because we have put baseGreeting into the scope, it will also be available to whatever pages or other components call on the Greeting component.

If we want to make baseGreeting unavailable to other pages and components, we can put baseGreeting into the variables scope like so:

```
component {
        variables.baseGreeting = "Hello, ";


        public string function getFullName (String firstName, String
lastName) {
            var fullName = arguments.firstName & " " & arguments.
lastName;
            return fullName;
         }


        public string function getGreeting (String firstName, String
lastName) {
            var fullName = getFullName(argumentCollection=arguments);
            var greeting = variables.baseGreeting & fullName;


                return greeting;
        }


}
```

Now that baseGreeting is in the variables scope, it can only be accessed from within the Greeting component. In looking at how to call the functions using the component, there are a few options. The first is using cfinvoke.

```
<cfobject
    component = "Greeting.cfc"
    name = "greeting" />

<cfinvoke component="greeting" method="getGreeting"
returnVariable="myGreeting">
    <cfinvokeargument name="firstName" value="Emily" />
    <cfinvokeargument name="lastName" value="Christiansen" />
</cfinvoke>

<cfoutput>
    #greeting#
</cfoutput>
```

First, cfobject was used to create an instance of our Greeting component. Then, the cfinvoke tag was used to call the getGreeting function. Using cfobject allows us to reuse the greeting component as many times as needed instead of re-instantiating it every time we want to call a function. The first thing you probably noticed about the above section is that it is an awful lot of code to do three things: 1) Instantiate the object 2) Invoke the function 3) Display the result.

Fortunately, there is a way to simplify this process. Let's start by instantiating our object:

```
<cfset Greeting = CreateObject("Component", "path.to.component.
Greeting") />
```

What we've done is use ColdFusion's built in CreateObject function to make an instance of our greeting component. CreateObject takes in two parameters:

Type: The type of object you want ColdFusion to create. In the example above we are making a component, so we pass in the string "Component". As of ColdFusion 9, this is no longer a required parameter.

Component-Name: The path to the component. Unlike the template argument for cfinclude, this argument takes the path to the component with each directory separated by dots instead of slashes. Because of this, it is not necessary to add the .cfc extension on to your file name. If you do, it will interpret the above as file ".cfc" in the folder "Greeting".

Now that we have the object, we can invoke the function:

```
<cfset myGreeting = Greeting.getGreeting(firstName="Emily",
lastName="Christiansen") />
```

Notice how we have switched to a dot notation, similar to our examples of helper functions above. Now we get the sense of the object (the Greeting) doing something (the function). This is far more readable and descriptive than the alternative.

Finally, we can output the results:

```
<cfoutput>
    #myGreeting#
</cfoutput>
```

Ultimately, our finished, refactored example looks like this:

```
<cfset Greeting = CreateObject("Component", "Greeting.cfc") />
<cfset myGreeting = Greeting.getGreeting(firstName="Emily",
lastName="Christiansen") />
<cfoutput>
    #myGreeting#
</cfoutput>
```

Conclusion

ColdFusion offers a wide variety of ways to accomplish code reuse. By leveraging these techniques, you will find your code to be less repetitive and more fun to write. Code reuse is not an end in and of itself; rather, it is a means to another end: maintainability. Having code that is broken up into concise, self-documenting parts enables you to read and interpret code much more quickly.

Hands On 12

By Simon Free

In this hands on, let's create a function that will take a string and output a string of ASCII characters. Some people think that this will stop email addresses from being spidered. This may or may not be the case, but the logic for creating a function is still the same.

Tags Used: <cfscript>, <cffunction>, <cfargument>, <cfset>, <cfloop>, <cfreturn>
Functions Used: asc, mid

1. Open up the `/www/about.cfm` file in your code editor.
2. Below the closing `</cfscript>` tag, create a `<cffunction>` tag with the following attributes:
   - **name**: convertStringToASCII
   - **output**: false
   - **returntype**: string
   - **hint**: Converts String to ASCII String
3. Directly after the open `<cffunction>` tag, create a `<cfargument>` tag with the following attributes:
   - **name**: stringToBeConverted
   - **type**: string
   - **required**: true
4. Your code should look similar to this:

```
<cffunction name="convertStringToASCII" output="false"
returntype="String" hint="Converts string to asccii string" >
    <cfargument name="stringToBeConverted" type="string" required=
"true" />
```

5. After the `<cfargument>` tag, create a `<cfset>` that sets an empty string to a variable called `convertedString`.
6. Before the variable name in the `<cfset>` tag, enter the word `Var`. This will scope the variable to the scope that is local to the function and will not interfere with any other variables on the page.
7. Your code should look similar to this:

```
<cfset var convertedString = '' />
```

8. After the <cfset> tag, create a <cfloop> tag with the following attributes:
    - **from**: 1
    - **to**: #len(arguments.stringToBeConverted)#
    - **index**: i
9. After the opening <cfloop> tag, create a <cfset>. This <cfset> tag will use string concatenation and should look similar to this:

```
<cfset convertedString &= '&##' & asc(mid(arguments.
StringTobeConverted,i,1)) & ';' />
```

10. After the <cfset> tag, create a closing </cfloop> tag.
11. Below the closing </cfloop> tag, create a <cfreturn> tag that returns the convertedString variable.
12. Below the <cfreturn> tag, create a closing </cffunction> tag.
13. Your function should look similar to this:

```
<cffunction name="convertStringToASCII" output="false"
returntype="String" hint="Converts string to asccii string" >
    <cfargument name="stringToBeConverted" type="string" required=
"true" />
    <cfset var convertedString = '' />

    <cfloop from="1" to="#len(arguments.StringToBeConverted)#" index="i">
        <cfset convertedString &= '&##' & asc(mid(arguments.
StringTobeConverted,i,1)) & ';' />
    </cfloop>

    <cfreturn convertedString />
</cffunction>
```

14. Locate the personalInfo.email output which should be on or around line 129.
15. Wrap the personalInfo.email output with a call to the convertStringToASCII function call. Your code should look similar to this:

```
#convertStringToASCII(personalInfo.email)#
```

16. Browse to the `/www/about.cfm` page in your browser.
17. You should see no change to your page.
18. View the source of the page and locate where the email address is being output. You should no longer see the email address but a stream of ASCII characters.

Hands On 13

By Simon Free

In this hands on, we are going to separate out the header and footer and put the code into separate files that will be included.

Tags Used: <cfinclude>, <cfparam>, <cfif>, <cfset>

1. Create a folder called `includes` in the `/www/` folder.
2. Create a new file in `/www/includes/` called `header.cfm`.
3. Create a new file in `/www/includes/` called `footer.cfm`.
4. Open up the `/www/about.cfm` file in your code editor.
5. Copy all lines from `<!DOCTYPE`, on line 2, down to and including the `<!--header end -->` line, on line 76.
6. Paste these lines of code into the `header.cfm` file.
7. Delete these lines from `/www/about.cfm`.
8. Copy all lines from below the `about end` comment.
9. Paste these lines of code into `footer.cfm`.
10. Delete these lines from `/www/about.cfm`.
11. In a browser, navigate to the `/www/about.cfm` page. Notice that it no longer looks correct.
12. Go back to the `/www/about.cfm` file in your code editor.
13. At the top of the file, just below your `<cffunction>` tag, block on or around line 11 and create a new tag called `<cfinclude>` with the following attribute:
    - **template**: includes/header.cfm
14. Go to the bottom of the file and create a `<cfinclude>` tag with the following attribute:
    - **template**: includes/footer.cfm
15. Go back to your browser and refresh the `about.cfm` file. You should now see the page as it used to be.
16. Open up the `/www/includes/header.cfm` file in your code editor.
17. At the top of the page, create a `<cfparam>` tag with the following attributes:
    - **name**: section
    - **default**: home
18. Locate the link to the homepage in the navigation, which should be on or around line 51.

19. Inside the `<li>` tag, add the following line of code:

```
<cfif section eq "home">id="selected"</cfif>
```

20. Do the same for the other links in the navigation, but instead of checking if `section eq "home"`, replace 'home' with the value of the `<li>` class.

21. Once completed your code should look similar to this:

```
<ul class="arrowunderline" id="nav">

    <li class="home" <cfif section eq "home">id="selected"</cfif>>
<a href="index.cfm">Home</a></li>

    <li class="about" <cfif section eq "about">id="selected"</cfif>>
<a href="about.cfm">About</a></li>

    <li class="resume" <cfif section eq "resume">id="selected"</cfif>>
<a href="resume.cfm">Resume</a></li>

    <li class="blog" <cfif section eq "blog">id="selected"</cfif>>
<a href="blog.cfm">Blog</a></li>

    <li class="portfolio" <cfif section eq "portfolio">id="selected"
</cfif>><a href="portfolio.cfm">Portfolio</a></li>

    <li class="contact" <cfif section eq "contact">id="selected"</cfif>>
<a href="contact.cfm">Contact</a></li>

</ul>
```

22. Refresh the `/www/about.cfm` page in your browser. You will notice that the navigation is currently highlighting the 'Home' navigation item rather than the 'About' navigation item. This is because we have not set the Section variable on the `about.cfm` page.

23. Open up the `/www/about.cfm` in your code editor and locate the `<cfinclude>` tag which includes the header. This should be on or around line 12.

24. Before the `<cfinclude>` tag, create a `<cfset>` tag and set the value of `About` to a variable called `Section`. Your code should look similar to this:

```
<cfset section = "About" />
```

25. In your browser, refresh the `/www/about.cfm` page and notice that the 'About' navigation item is now highlighted.

Home Work

Update all other pages in the site so that they use the included header and footer rather than the inline code.

Hands On 14

By Simon Free

In this hands on, we will create a custom tag that will handle the display of the header and footer.

Tags Used: <cfif>, <cfelse>, <cfset>, <cfimport>

1. Create a folder called `customTags` in the `/www/` folder.
2. Create a file called `page.cfm` in the `/www/customTags/` folder.
3. Open the `/www/customTags/page.cfm` file in your code editor.
4. Create a `<cfif>` tag that checks if `thisTag.executionMode` is equal to `"Start"`.
5. Create a `<cfelse>` tag after the `<cfif>` tag.
6. Create a closing `</cfif>` tag after the `<cfelse>` tag.
7. Your code should look similar to this:

```
<cfif thisTag.executionMode eq "start">

<cfelse>

</cfif>
```

8. Open up the `/www/includes/header.cfm` file in your code editor.
9. Copy the contents of `header.cfm` and paste them into the first part of the `<cfif>` tag block in `page.cfm`.
10. Open up the `/www/includes/footer.cfm` file in your code editor.
11. Copy the contents of `footer.cfm` and paste them into the `<cfelse>` part of the `<cfif>` statement in `page.cfm`.
12. Open up the `/www/about.cfm` file in your code editor and navigate to the `<cfset>` tag located on or around line 12.
13. Replace the `<cfset>` tag with a `<cfimport>` tag with the following attributes:
    - **taglib**: customTags/
    - **prefix**: layout
14. Your code should look similar to this:

```
<cfmport taglib="customTags/" prefix="layout" />
```

15. Replace the <cfinclude> tag, which includes the `header.cfm` file, with a call to your custom tag. The code should look similar to this:

```
<layout:page>
```

16. Replace the <cfinclude> tag, which includes the `footer.cfm` file, with a closing tag to your custom tag. Your code should look similar to this:

```
</layout:page>
```

17. Open up your browser and navigate to the `/www/about.cfm` page. Notice that the page loads as it did before. The page is now using your custom tag rather than the include files. The only exception is that the 'Home' navigation item is being selected rather than the 'About' navigation item.

18. Open up the `/www/customtags/page.cfm` file in your code editor.

19. Find the <cfparam> tag that should be on or around line 2.

20. Change the `name` attribute from `section` to `attributes.section`.

21. Locate the Navigation <ul> block on or around line 51. For each reference to section in the <cfif> tags in the <li> tags, update them to read `attributes.section` rather than `section`. Your code should look similar to this:

```
<ul class="arrowunderline" id="nav">
  <li class="home" <cfif attributes.section eq "home">id="selected"
</cfif>><a href="index.cfm">Home</a></li>
  <li class="about" <cfif attributes.section eq "about">id="selected"
</cfif>><a href="about.cfm">About</a></li>
  <li class="resume" <cfif attributes.section eq "resume">id="selected"
</cfif>><a href="resume.cfm">Resume</a></li>
  <li class="blog" <cfif attributes.section eq "blog">id="selected"
</cfif>><a href="blog.cfm">Blog</a></li>
  <li class="portfolio" <cfif attributes.section eq
"portfolio">id="selected"</cfif>><a href="portfolio.cfm">Portfolio</a>
</li>
  <li class="contact" <cfif attributes.section eq
"contact">id="selected"</cfif>><a href="contact.cfm">Contact</a></li>
</ul>
```

22. Refresh the `/www/about.cfm` page in your browser. You will notice that the navigation is currently highlighting the 'Home' navigation item rather than the 'About' navigation item. This is because we have not set the Section variable on the `about.cfm` page.
23. Open up the `/www/about.cfm` file in your code editor.
24. Locate the `<layout:page>` tag and add an attribute called `section` with a value of "about". Your code should look similar to this:

```
<layout:page section="about">
```

25. Refresh the `about.cfm` page in your browser and confirm that the 'About' navigation item is now highlighted.

Homework

Convert all other pages to use the page custom tag

Hands On 15

By Simon Free

In this hands on, let's create a component where we will place the `convertStringToASCII` function inside of a component and instantiate it.

Tags Used: <cfcomponent>, <cffunction>
Functions Used: createObject

1. Create a folder called `cfc` inside the `/www/` folder.
2. Create a file called `utilities.cfc` inside of the `/www/cfc/` folder.
3. Open up the `/www/cfc/utilities.cfc` file in your code editor.
4. Create an open and closing `<cfcomponent>` tag. Your code should look something like this:

```
<cfcomponent>


</cfcomponent>
```

5. Open up the `/www/about.cfm` file in your code editor.
6. Copy the `<cffunction>` block of code and paste it between the `<cfcomponent>` tags in `/www/cfc/utilities.cfc`.
7. Inside the `<cffunction>` tag, add an attribute called `access` and set it to "public".
8. Your code should look similar to this:

```
<cfcomponent>
    <cffunction name="convertStringToASCII" output="false"
returntype="String" hint="Converts string to asccii string"
access="public">
        <cfargument name="stringToBeConverted" type="string"
required="true" />
        <cfset var convertedString = '' />


        <cfloop from="1" to="#len(arguments.StringToBeConverted)#"
index="i">
            <cfset convertedString &= '&##' & asc(mid(arguments.
```

```
StringTobeConverted,i,1)) & ';' />
          </cfloop>

          <cfreturn convertedString />
     </cffunction>
</cfcomponent>
```

9. In `/www/about.cfm`, replace the `<cffunction>` tag block with a `<cfset>` tag. The `<cfset>` tag should set a variable called `utilities` and should instantiate the utilities component by using the following code:

```
<cfset utilities = CreateObject('cfc.utilities') />
```

10. Locate the call to the `convertStringToASCII` function, which should be on or around line 58, and change the function call to `utilities.convertStringToASCII`. The line of code should look similar to this:

```
#utilities.convertStringToASCII(personalInfo.email)#
```

11. Go to the `/www/about.cfm` page in your browser and notice that the email address displays as it did before. The `convertStringToASCII` function is now in a cfc that can be instantiated from any page.

Homework

Convert the email address in the file `/www/contact.cfm` to use the `convertStringToASCII` function.

6

Application.cfc

By Adam Tuttle

About Adam Tuttle

Adam Tuttle has been developing web applications since the year 2000, starting with ColdFusion 4.5. In that time he has worked for manufacturing companies, in academia, and as a consultant; but ColdFusion was a constant. When it comes to work-focus Adam is a jack of all trades, but he especially enjoys in REST, SOA, Security, and Mobile-optimized design.

In addition to working with and speaking about ColdFusion, he is also the Co-manager of the Philadelphia ColdFusion User Group, an Adobe Community Professional for ColdFusion, and maintains a growing library of open source projects, including Taffy. In addition to his own projects, he also contributes to the Mango Blog core, and runs CFCommunity, the GitHub organization working on filling in the gaps in CFScript for CF9+.

Adam and his wife have two sons and together they enjoy gaming, camping, hiking, fishing, snowboarding, and cheering for the Phillies and the Eagles.

By now you've learned the basics of ColdFusion, script vs. tag syntax, scopes, how to deal with data, and even some code-reuse techniques. You're now able to write something useful, so it's time we introduce you to the Request Lifecycle.

You see, when someone requests a ColdFusion page, CF doesn't just start executing your code. There are several events that first take place, which you can be waiting for, and to which you can react. This is not strictly necessary, but you'll find that any complex application will eventually want to make use of some or all of these features, so it's best that you know about them. In order to react to these events, you need to have an `Application.cfc` file. ColdFusion has designated `Application.cfc` as a special component that it will automatically look for, and in which we can put our event listeners for request lifecycle events.

> **A note on Terminology**
>
> I'm going to use some technical terminology here for the sake of correctness, so let's explain it quickly. When an event "happens" we call that "broadcasting" the event. When you react to an event, that's called "listening" for it. A function that "listens" for or "handles" an event is often referred to as either an "event listener" or "event handler".

Request Lifecycle Events

Lifecycle events add a lot of power and flexibility to your ColdFusion applications, so it's important to understand what events are available and why you might want to use each.

- **onApplicationStart:** The very first time your application is used (think immediately after a reboot), the `onApplicationStart` event is broadcast. This gives you an opportunity to define some application-specific configuration, prime caches, and do a lot more. The important part to remember is that, as a rule of thumb, it only happens once, until your application times out (hasn't been used in a while), the process is restarted, or the computer is restarted.
- **onSessionStart:** Any time that a user makes a request to your application and either they haven't used it before, or it's been long enough that their session has expired -- as identified by their cookies, usually -- the `onSessionStart` event is broadcast. This allows you to set session defaults, like a flag indicating that the user is not currently logged in; or to redirect to the login page.
- **onRequestStart:** The two previous events have been specific to the first time something happens. `onRequestStart` is broadcast before every request, giving you an opportunity to set variables that should be accessible on every page in the site, or to validate that the user is allowed access to the requested page, for instance.
- **onRequest:** The `onRequestStart` event has this weird cousin, `onRequest`, which for now we'll just say does almost the same thing, but differently. You can effectively use them both (`onRequestStart` is broadcast before `onRequest`), but they operate a little differently. Don't worry, we'll cover this in more depth shortly.
- **onRequestEnd:** Similarly to `onRequestStart`, `onRequestEnd` is broadcast after your template returns control to CF, giving you the opportunity to add code to every request after the requested template has executed. You might use this to add logging, for example.
- **onSessionEnd:** You may have guessed it, and if so, you were right. There is also an `onSessionEnd` event, which you can use to do things like empty a shopping cart that was

never paid for (thus returning that reserved stock back to availability for other customers to buy).

- **onApplicationEnd:** And of course, there's an `onApplicationEnd` event, too. Lovely symmetry, isn't it? This event is broadcast when your application times out (hasn't been used in a while), or if ColdFusion is shutting down.
- **onError:** Lastly, there is an `onError` event broadcast in the event of an un-caught exception, including any you might manually throw. This gives you a last line of defense to deal with any errors that might have slipped through the cracks in your application.

Time for the Code!

If you're reading this course sequentially, you've already been exposed to ColdFusion Components, or CFC's. If not, please go back and read the previous chapter.

> For the sake of keeping the mid-chapter code samples brief and readable, I'll only show one function at a time; but any combination of them, all of them, or none of them could be in your component, and the order they are in has no effect.

```
component {

    function onApplicationStart(){
        application.datasource = "my_database";
        return true;
    }

}
```

This method listens for the `onApplicationStart` event and sets the value "my_database" into the Application Scoped variable named `datasource`. This function, `onApplicationStart`, will only be called on the relatively rare occasion that your Application has timed out or is otherwise not already running. The benefit of this should be clear: You can put a lot of code in here, and it will only be executed once in a while, instead of for every request.

This is the logical place to set most Application-scoped variables; since the variables live as long as the application, and when the application is just starting up this method will re-initialize them for you. For that reason, it makes perfect sense to set your application configuration as application-scoped variables in the `onApplicationStart` method.

Of course, every single one of the events broadcast for `Application.cfc` are optional. You don't have to include listeners for all of them; most people don't.

onRequest

Do you remember when I said that `onRequest` was a little bit different? Here's how. The requested template, e.g. `index.cfm` will be passed to `onRequest` as an argument, and it's up to you to **include** it.

This allows you more control over what happens before and after template execution, and allows you to wrap the include; for example in a `transaction`, or in a `try/catch` block. Here's how it might look:

```
component {

    function onRequest( string targetPage ) {

        try {
            include arguments.targetPage;
        } catch (any e) {
            //handle the exception
        }

    }
}
```

onSessionEnd and onApplicationEnd

`onSessionEnd` is passed two arguments, `SessionScope`, and `ApplicationScope`. Neither is direct access to the actual Session or Application scopes, and instead, are copies of what those scopes were as the session was ending. `onApplicationEnd` is similarly passed a copy of the former Application scope.

They are called when the session expires and the application stops, respectively, whether or not a request is made. For that reason, there may not be a "page" to render any output to. Instead, use alternative output options like logging, writing to files, and email, if you need any output from either of them.

onError

This method is passed two arguments: `Exception` and `EventName`. `Exception` is functionally equivalent to a `cfcatch` result, and `EventName` is the name of the event handler in which the error occurred. If the error occurred in a requested template (e.g. not in `Application.cfc`), and you have not implemented `onRequest`, it will be an empty string.

A common practice is to use this method to catch application exceptions and send a notification to the developer(s) via email. While this is a perfectly functional approach, it tends not to scale well as your application grows and errors become more frequent. If you start to notice yourself spending too much time dealing with error emails, look into community projects like BugLogHQ and Hoth.

onSessionStart, onRequestStart, and onRequestEnd

These methods are very similar to `onApplicationStart` in their usage, with the one exception

that `onRequestStart` is passed the `targetPage` as an argument. None of them need to return anything or are required to operate in any specific way. The above samples pretty well summarize how you would write each of the Request Lifecycle Event methods, so we won't bore you by writing the same code sample over and over. You'll find a complete empty template for an `Application.cfc` at the end of this chapter. Once you grasp the basics, it should be pretty self explanatory as a reference.

More than Just Events

As if that wasn't enough, `Application.cfc` has more parts!

You are free to define as many other functions in `Application.cfc` as you would like, if they would be of use. These functions will be available for use anywhere in `Application.cfc`, and anywhere in a template included from `onRequest`. However, if a template from `onRequest` is not included, the functions will not be available to every template in your application.

It also accepts setting certain configuration settings in the imlicit constructor, discussed in the Components section of the Code Reuse chapter. To review, anything inside a component (between the `<cfcomponent></cfcomponent>` tags or the `component {}` braces) but not inside a function, is part of the implicit constructor, and as such will be executed during the instantiation of the component.

ColdFusion defines more than 20 settings that can be set and updated from the implicit constructor in `Application.cfc`, however, a full discussion of each is outside the scope of this course. We'll cover the basics, and for the rest, you should read the official documentation. Inexplicably, the primary documentation for `Application.cfc` settings makes no mention of it, but there are also numerous ORM-related settings that can be configured here, which is documented in ORM Settings.

Basic Settings

Every application needs a name. It must be more or less alphanumeric and as a best practice should not include spaces or any symbols or punctuation. This name is used to create different contexts for each application running on the server, so that they don't share any memory. Without a unique name, your application's code could potentially overwrite memory values for another application.

```
this.name = "my_application3";
```

While a default application timeout setting is set server-wide in the CF Administrator, you can override this setting, up to a CF Administrator defined limit, with `this.applicationTimeout`.

```
this.applicationTimeout = CreateTimeSpan(10, 0, 0, 0); //10 days
```

Custom tags, covered in the Code Reuse chapter, can be sourced from numerous directories. You can add to the global list in the CF Administrator, and your `Application.cfc` can also define some of its own additions to the list; making your code more portable.

```
this.customTagPaths = [ expandPath('/myAppCustomTags') ];
```

Similarly, you can set per-application mappings that supercede any mappings of the same name set in the CF Administrator for requests in your application.

```
this.mappings = {
    "/foo" = expandPath('/com/myCompany/foo')
};
```

Each application is allowed one app-global default datasource, which will be used if none is defined for a query or stored procedure.

```
this.datasource = "my_datasource";
```

Session management -- whether or not sessions are enabled and tracked for your application -- can be enabled or disabled via the `sessionManagement` setting. You can also specify the session timeout.

```
this.sessionManagement = true;
this.sessionTimeout = CreateTimeSpan(0, 0, 30, 0); //30 minutes
```

Just a reminder: Do yourself a favor and check out the primary list of other available settings, as well as the ORM settings.

What is an Application?

ColdFusion applications differ from traditional desktop applications in that they are not always "running" when enabled. CF Applications only ever do anything when an http request is made; with the exception of special cases like `onApplicationStop` and `onSessionStop`.

When a request is made during the execution of the `Application.cfc` implicit constructor code, the applicable application name is determined, and the applicable memory scopes for that application are associated with the request. Then, any necessary events are broadcast and responded to, in serial (`onSessionStart` will not fire before `onApplicationStart` returns; `onRequestStart` waits for `onSessionStart`, and so on...). Control is then passed to the requested template. After the template completes, control shifts back to `Application.cfc` where `onRequestEnd`, if defined, will be executed.

Where does Application.cfc go?

Technically speaking, and assuming the default settings from a vanilla ColdFusion install, you can have an `Application.cfc` anywhere within your application structure, and even a few places outside of it. Doing so implies that everything in that folder and all of its subfolders (and theirs, and theirs...) are part of that application.

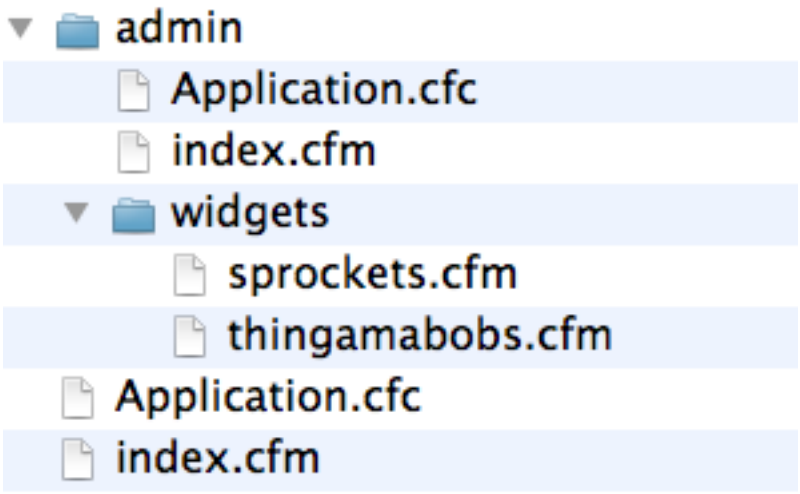
If a request is made for `you.com/foo/bar/widgets/index.cfm`, CF looks inside that widgets folder for an `Application.cfc`. If found, then it uses the `application name` set therein. If not found CF then checks the parent folder, `bar`. This process is repeated over and over until it reaches the system root. This is the default setting as of ColdFusion 10.

You can change this setting in the CF Administrator, under Server **Settings > Settings**, near the bottom of the page, under the heading "Application.cfc/Application.cfm lookup order". Other options are "until webroot", where the process described above stops at the web root instead of the system root, and "in webroot", which specifies that CF should only ever look for `you.com/Application.cfc`.

Applications inside Applications

The obvious next question is, if we can have `Application.cfc` files anywhere, can we nest one application inside another? The answer is **Yes**.

Consider the following directory structure:



Here we have two applications: One for the main website, and one for the admin section of the site. Since there is no `Application.cfc` in the widgets folder, any requests to `widgets/sprockets.cfm` will use the `Application.cfc` from the parent folder.

The two applications can have completely different settings and different code for their event handlers, as long as they have distinct names.

What's Application.cfm?

A relic of a bygone time! Back in the dark ages, before ColdFusion had components, `Application.cfm` was a similar approach to solve some of the same problems. The primary difference is that everything in `Application.cfm` is executed at the start of every request, more or less like the `onRequestStart` method does now. The other major difference is that instead of the component implicit constructor to set configuration settings, you set them as arguments to the `cfapplication` tag.

You can "fake" similar functionality to `onApplicationStart` in `Application.cfm` by using a flag in the application scope:

```
<cfif not structKeyExists(application, "initialized")>
    <cfset application.datasource = "my_datasource" />
    <cfset application.initialized = now() />
</cfif>
```

Depending on what you are doing inside that block, you may want to make use of locking to prevent multiple simultaneous requests from all executing the application initialization code at the same time.

Fortunately, we have evolved. These days, `Application.cfm` is mostly just a sign of old code.

Appendix: Application.cfc Template

For your reference, here is an empty `Application.cfc`, with all of the bells and whistles waiting to be filled in.

```
component {

    this.name = "myApplication";
    this.applicationTimeout = CreateTimeSpan(10, 0, 0, 0); //10 days
    this.datasource = "my_datasource";
    this.sessionManagement = true;
    this.sessionTimeout = CreateTimeSpan(0, 0, 30, 0); //30 minutes
    this.customTagPaths = [ expandPath('/myAppCustomTags') ];
    this.mappings = {
        "/foo" = expandPath('/com/myCompany/foo')
    };

    // see also: http://help.adobe.com/en_US/ColdFusion/10.0/CFMLRef/WSc
3ff6d0ea77859461172e0811cbec22c24-750b.html
    // see also: http://help.adobe.com/en_US/ColdFusion/10.0/Developing/
WSED380324-6CBE-47cb-9E5E-26B66ACA9E81.html

    function onApplicationStart() {
        return true;
    }

    function onSessionStart() {}

    // the target page is passed in for reference,
    // but you are not required to include it
    function onRequestStart( string targetPage ) {}
```

```
    function onRequest( string targetPage ) {
        include arguments.targetPage;
    }

    function onRequestEnd() {}

    function onSessionEnd( struct SessionScope, struct ApplicationScope
) {}

    function onApplicationEnd( struct ApplicationScope ) {}

    function onError( any Exception, string EventName ) {}

}
```

Hands On 16

By Simon Free

In this hands on we will create an `Application.cfc` file, create some application wide variables, and implement some of the `Application.cfc` event handlers.

Tags Used: <cfset>, <cfquery>
Functions Used: createTimeSpan, structKeyExists, createObject

1. Create a new file called `Application.cfc` in `/www/`.
2. Open up the `/www/Application.cfc` file in your code editor. Note: for this hands on we will be doing all our coding in script format.
3. Create a `component` declaration. Your code should look similar to this:

```
component{
}
```

4. Create two variables in the this scope called `name` and `datasource`. Set both variables to 'learncfinaweek'.
5. Create another variable in the this scope called `applicationTimeout` and give it a value of:

```
CreateTimeSpan(10, 0, 0, 0);
```

6. Create another variable in the `this` scope called `sessionManagement` and set it to true.
7. Create another variable in the `this` scope called `sessionTimeout` and give it a value of:

```
CreateTimeSpan(0, 0, 30, 0)
```

8. Your code should look similar to this:

```
component{
    this.name='learncfinaweek';
    this.datasource='learncfinaweek';
    this.applicationTimeout = CreateTimeSpan(10, 0, 0, 0);
    this.sessionManagement = true;
    this.sessionTimeout = CreateTimeSpan(0, 0, 30, 0);
}
```

9. Create a function called `onApplicationStart` that accepts no arguments and returns true.
10. Inside the `onApplicationStart` function, create two variables set in the application scope. The first variable should be called `myName` and should have the value of your name. The second variable should be called `myPosition` and should have the value of 'A Developer'.
11. Your code should look similar to this:

```
component{
     this.name='learncfinaweek';
     this.datasource='learncfinaweek';
     this.applicationTimeout = CreateTimeSpan(10, 0, 0, 0);
     this.sessionManagement = true;
     this.sessionTimeout = CreateTimeSpan(0, 0, 30, 0);

     function onApplicationStart() {
          application.myName = 'Simon';
          application.myPosition = 'A Developer';
          return true;
     }
}
```

12. Below the `onApplicationStart` function,create a new function called `onRequestStart` which accepts one string parameter called 'targetPage'.
13. Inside the function, create an `if` statement that calls the `StructKeyExists` function. The `structKeyExists` function should be passed the URL scope as its first parameter and the string 'reload' as its second parameter.
14. Inside the `if` statement, a call should be made to the `onApplicationStart` function.
15. Your code should look similar to this:

```
function onRequestStart(string targetPage){
     if(structKeyExists(url,'reload')){
          onApplicationStart();
     }
 }
```

16. Open up the `/www/index.cfm` file in your code editor.
17. Remove the `<cfset>` tags at the top of the page.
18. Locate the `myName` variable output on or around line 15 and change it to `application.myName`.
19. Locate the `myPosition` variable output on or around line 16 and change it to `application.myPosition`.
20. In your browser, navigate to the `/www/index.cfm` page. The index page should display as normal.
21. Go to the `/www/application.cfc` file in your code editor and change the `application.myPosition` variable to have the value of 'A Great Developer'.
22. In your browser, refresh the `index.cfm` page. Notice that nothing has changed. This is because the application has already started, so the `onApplicationStart` method does not get called.
23. In your browser, append `?reload=1` to the `index.cfm` in the location bar. Hit return to load the page. Notice that the position has now updated to the new value.
24. Open up the `/www/resume.cfm` file in your code editor. Locate both `<cfquery>` tags and remove the `datasource` attributes.
25. In a browser, navigate to the `/www/resume.cfm` page. Notice that the page loads normally. Both queries are now using the application wide data source.
26. In the `onApplicationStart` function in the `/www/Application.cfc` file, create a new application variable called `utilities`. Set the value of `application.utilities` to:

```
CreateObject('cfc.utilities');
```

27. Your code should look similar to:

```
function onApplicationStart() {
    application.myName = 'Simon';
    application.myPosition = 'A Great Developer';
    application.utilities = CreateObject('cfc.utilities');
    return true;
}
```

28. Open up the `/www/about.cfm` file in your code editor and remove the `<cfset>` tag on or around line 4.
29. Locate the `utilities.convertStringToASCII` function call on or around line 58 and

change it to `application.utilities.convertStringToASCII`.

30. In a browser, navigate to the `/www/about.cfm` page, remembering to include `?reload=1` in the URL as we have made an `application.cfc` change.
31. Notice that the page loads normally. The `convertStringToASCII` function is now stored in the application scope and can be accessed from any page within the application.

Homework

- Remove the datasource attribute from all query tags.
- Update the email address in the `/www/contact.cfm` file to use the `convertStringToASCII` function.

7

OOP

By Adam Tuttle

One could write a book specifically about Object Oriented Programming (OOP). This chapter is an OOP primer to get you started, but for a more in-depth explanation, check out Matt Gifford's Object Oriented Programming in ColdFusion.

What is OOP?

Object Oriented Programming is a set of concepts and techniques that make use of the "object" language construct, to write more reusable, maintainable, and organized code. Objects are implemented differently in every language; in ColdFusion, we have ColdFusion Components (CFCs). Using objects doesn't require OOP, and not every use of objects is OOP. They are simply the building blocks for writing OOP code.

When you write a lot of OOP code, you'll quickly find yourself writing repetitive code to wire together everything necessary to respond to a given request; if you take some time to write a single path through the code that analyzes the request and automatically wires together the things necessary to respond, then you've essentially written your own front-controller framework.

Frameworks are not an essential part of OOP, but they do solve a common set of problems, and it’s best not to write code without one. Some people choose to write their own frameworks, but if you're just getting started, make use of one of the many that are already available, as they have been established and have already had time to work out the kinks and bugs.

So what exactly does OOP do to make your code more organized and maintainable? You decouple unrelated sections of code, you encapsulate related functionality into the same object, and different but related object types can inherit functionality from one another or from a base object. This is all possible through the understanding and use of classes, instances, methods, and abstraction.

By combining some or all of the concepts below, you’ll find that your code is very DRY (Don’t Repeat Yourself) and well organized, enabling you to have a good separation of logic from presentation.

Classes, Instances, Methods, and Abstraction

In ColdFusion, a Class is simply a CFC. The component defines a set of public methods (functions) and can have both public and private data. It may also have private methods that are used by other methods in the class to improve code reuse and abstract complex jobs into small maintainable chunks.

Anything that's both complex and discrete is a good candidate for abstraction. For example, say you have an array of structures, each with a key named "foo,” and you want to get an array of the values of "foo" from each struct. Doing so probably only takes about 10 lines of code, but it's a fairly complex chunk of code; anyone reading your code later will get distracted from reading the entire method to figure out what this 10 lines of code does. Instead, you could take that same 10 lines of code and wrap it in its own method -- let’s call it `reduceArrayToFoos()` -- and then call it instead of writing that code inline. Then when your coworkers, or even yourself, are reading your code in 6 months, you'll be able to scan past that line because it's obvious what it does. The same concept can be applied at a higher level to abstract related functionalities for the same data or job into a utility or model class.

Inside a CFC, the `this` scope is where you can place public variable, and the `variables` scope is where you can place private variables that are globally available to the entire class. Methods can also have private variables that are not shared to other methods, which you put in the `local` scope (or use the `var` keyword, which does the same thing).

Instances tend to be the concept of OOP that people have the most trouble with. Your CFC is the class; it's the blueprint for an object, but you can create as many Instances of that class as you like. When you call `CreateObject()`, you're creating an instance of the class you specify. You can use instances in two different ways: The first are Transient instances. A transient instance is one where you create, use, and throw away when you're done. If you don't specify that it should be saved, ColdFusion will throw it away for you at the end of the request. The second instance are Singleton instances.This is an instance that lives beyond the request that created it, and future requests can use it. Typically with a singleton, you create only one single instance of it and everything that uses it uses that same instance, hence the name **single**ton. You tell ColdFusion to persist it by storing it in one of the persistent scopes: `Server`, `Application`, and `Session` would be the most common.

Now that you understand these basic concepts, let's dive into the details of what makes OOP tick.

Encapsulation, Decoupling, Inheritance, and Polymorphism

Encapsulation is the concept of bundling bits of data and some methods related to that data into an object. It often will include a distinction between public and private bits of data. For example, we may have a User object that contains the user's name, birthday, and salary. The object also has accessors and mutators. Accessors are methods that allow you to read or write a bit of data on the object -- public or private, it makes no difference; Mutators change the object and may or may not require any input. For example, a shopping cart object may have a mutator `cart.calculateSalesTax('PA')` that would calculate the sales tax for the items in the cart based on the customer's residence in Pennsylvania.

Using the objects created with Encapsulation, we can apply the rest of the techniques to create more maintainable and testable code.

Decoupling is the separation of chunks of code that shouldn't need to know about the details of each other. The obvious case of this is separating the presentation of data from the logic that retrieves it from the database and manipulates it. However, decoupling also applies to unrelated groups of related code. For example, all of the code for `widget` management should be together, but it should not be mixed with the code for `user` management; and both groups should have their own class: `WidgetService` and `UserService`. By decoupling your code, you can change the logic of the `UserService` with confidence that you're not messing with anything widget-related.

Inheritance is a way to reuse existing code, or a way to write code in one location that many objects can make use of. When an object of class B inherits from class A, object B contains all of the code -- that is, methods and data -- from class A, plus all of the code from object B. Importantly, for any methods and data that exist in both classes A and B, the value or implementation from object B takes precedence; allowing you to override the behavior or value of an object when you extend (that is, inherit from) it. Lastly, your implementation of methods in object B can also call the implementation from object A, more or less as a "wrapper" for the implementation in class A.

Polymorphism is a concept that is more evident in strictly typed languages, where it indicates that one type is somehow derived from another or implements a specific interface. In ColdFusion, as a dynamically typed language, data is implicitly polymorphic. Polymorphism allows objects of different types to have the same, or similar, APIs. For example, a `user` object and an `administrator` object are very similar. An `administrator` might inherit from the `user` class, then add its own additional properties and methods for special abilities it has permission to use. However, for everything that both `Users` and `Administrators` interact with, the code can assume the object implements everything in the `User` class. Consider blog comments and the blog author is its administrator. When an administrator leaves a comment, the `administrator` object will have the same `getUsername()` and `getEmailAddress()` methods that a `user` object would. You can think of polymorphism as a strict use of inheritance to make related but slightly different classes.

Why Should I use OOP?

OOP has survived the test of time by proving that it solves a certain set of problems well. When well written, it is DRY, Maintainable, and Testable.

DRY is an acronym for Don't Repeat Yourself, and means that you should write your code in a way that promotes reusing existing implementations. Writing a utility class with commonly used user defined functions allows you to address a bug found in the functionality of one of those functions in one place. If the same code was not a UDF and instead had been copied and pasted all over your codebase, you would have to do a lot of searching and replacing. Not only would that be tedious, but you would have more opportunities to make a mistake while applying the fix. Similarly, the DRY approach would say that there should only be one class, your `UserService`, capable of reading and writing `User` data from and to the database. With this approach, if you notice a bug when data is stored to the database, you know exactly where to find it, because there's only one place that writes that data to the database.

Maintainability is a sort of nebulous, almost subjective topic, but a majority of veteran developers agree that writing OOP code that is DRY makes it vastly more maintainable than the typical copy & paste "spaghetti code" approach.

OOP also makes your code much more unit-testable. OOP is not a requirement for integration tests, as with using a tool like Selenium, because it doesn't look at the code. Instead, it looks at "screens" or "views" and interacts with them, allowing you to assert that certain results and behaviors occur. For example, while integration testing is making sure that the beach looks the way you expect, unit testing is making sure that each grain of sand on the beach looks and behaves as it should. Unit tests know the names of the methods in each class and verify that each does what it should. If you're interested in Unit testing for ColdFusion, the most popular tool for the job is MXUnit. Both have their place, but if you want to do unit testing, you're better off with an OOP approach.

How to Write Object Oriented ColdFusion

There are very few strict requirements for writing OOP code in ColdFusion. You'll be using Objects (CFCs), but what you name them, what folders you put them in, if any, and so on, are all entirely up to you. What it boils down to is that you use Objects (Classes, Instances of those classes, and Abstraction) along with the concepts discussed above: Encapsulation, Inheritance, Polymorphism,

and Decoupling.

Let's say your application has users. You need a `UserService`, so you create an object named `UserService`. It has a `getUser()` method that returns an object of your User class. It also has a `saveUser(user)` method to save any changes you might make to that user `object` while the application is running, such as if the user updates their password. You only need one `UserService`, so you make it a Singleton, but every `user` gets its own user object, so that should be a transient object. We've described two types of objects here: data objects, sometimes referred to as **Beans**, and **Services**. Collectively, these should be considered your application's **Model**. (If you choose to use ORM, the ORM objects would also be part of your Model.)

Your code should take the new password from the user, put it into the `user` object, and pass the `user` object to the `saveUser(user)` method of the `UserService`. Then, barring any errors, it should report to the user that their password has been updated. This type of process is a **Controller**.

The last important piece are the templates that display data to the user, collect data from them, and allow them to navigate around the data; these are known as **Views**. But where do views get the data they display, and what do they do with it after they collect it? They get it from, and give it to, the controller.

In a nutshell, that is Object Oriented Programming. If the way you write your code satisfies these requirements, then your code is Object Oriented. Obviously, there are still a lot of details whose implementation is up to you; and that is why there are frameworks -- and many of them.

People have differing preferences for how they want their Models, Views, and Controllers organized, what behaviors the framework should add or hide, and how the framework provides "extension" points where you can modify the way the framework works or affect your code at a future point; thus we have different frameworks to choose from.

In fact, if you choose not to use an existing framework, instead choosing to write your own OOP code, you will do one of two things: you'll find yourself rewriting similar code over and over to wire together the `request` and `response` for each type of request, or you'll try to write something that automates that for all requests based on part of the request. If you choose the latter, you've just written your own MVC framework. Unless you've experienced at least a few of the existing frameworks, know what they have to offer, and know that you can do better, it is recommended that you use what currently exists . There's always room for improvement, but existing frameworks have the advantage that they've been available for a while and have large existing userbases that have reported and helped work out most, if not all, of the bugs. When you start from scratch, you will have to go through all of that as well. A hybrid approach would be to fork, or contribute, to an existing framework that provides most of what you want or does it mostly the way you want, but to add to it or update it as you see fit. If the wider community appreciates your changes, there's a good chance they'll be accepted into the framework and distributed to the rest of the community.

Common OOP Pitfalls

This list is not exhaustive, but hopefully will help you avoid some of the more common problems today's developers cause for themselves.

Accessing Global Scopes Directly

Red flags go up during a code review when there is code that is accessing global scopes (`Server`, `Application`, `Session`, `Request`, `Client`, `Cookie`) in model objects. One of the core tenets is obviously decoupling, and in this case it requires that everything that the object needs should be passed in either when the object is instantiated or when it is used. It should not reference anything outside itself.

Going back to the shopping cart tax calculation example, the state whose tax is being calculated is a requirement for the calculation, and it is provided to the function during use. Let's extend this example and say that we want to track statistics about our customers; specifically, we want to see a log of every time they update their cart: additions, removals, and updates. To do so, we're going to use the application's existing logging service. We happen to know that the logging service is persisted as `Application.LogService`, so we could just reference it from the `ShoppingCart` object and add our logging:

```
Application.LogService.log('user 1234 added item f3489j to their cart');
```

However, this breaks encapsulation and is considered bad practice. Instead, it should be set into the `ShoppingCart` object during instantiation:

```
Session.shoppingCart = new model.ShoppingCart(
    logService = Application.logService
);
```

Did you notice that the above code sample still references a shared scope (Session)? That's because this is not a sample from inside the `ShoppingCart` class, this is inside a controller.

The 5-for-1 Problem

Another common mistake people make is to create 5 different objects for every table in their database: Gateway, DAO (Data Access Object), Bean, Service, and Controller. While it is conceivable that some or all of these classes would be necessary for any given table in your application, it's not necessary to split them up. Generally speaking, a Gateway and a DAO do the same things, except a DAO is meant to read or write one record while a Gateway is meant to read or write many records.

Right away, it seems obvious to combine them. Then again, with the Gateway and DAO outside of the service, there's not much left in the service. The actual preference is to combine all three. If you're using ORM you'll still have that external to your service, but aside from that, there is no need to split data access out from the service at all.

In looking at Beans and Controllers, not every table in your database needs its own beans and controllers -- or services for that matter. Consider a person that has multiple addresses, email addresses, and phone numbers. In the 3rd Normal Form of database serialization, you'll have

separate tables for people, addresses, emails, and phone numbers. But this does not necessitate a `PhoneNumberService`, `PhoneNumberController`, and `PhoneNumber` bean, or the same collection of objects for addresses or email addresses. Instead, these things should all be handled within the `person` service, and you can skip the controllers. Whether or not you create beans for them would depend on how you use your beans, but they are not strictly necessary -- they could easily be a property in the `person` object.

If the 5-for-1 problem arises from a "top down" approach where the database is at the top and dictates the object model, then the majority of veterans these days would advocate for thinking with your user interface as the top, and dictate the object model from that, keeping it as simple as possible, but as complex as necessary.

Hands On 17

By Simon Free

In this Hands On we are going to create an encapsulated cfc that holds error data.

Functions Used: ArrayAppend, ArrayLen

1. Create a new file called `errorBean` in the `/www/admin/cfc/` folder.
2. Open up the `/www/admin/cfc/errorBean.cfc` file in your code editor.
3. This cfc will be written entirely in `cfscript` so create a `cfscript` based component declaration. Your code should look similar to this:

```
component{
}
```

4. The first function we are going to create is the `init` function. This function will instantiate the internal variables and return itself. Create a function called `init` that is `public` and returns a variable type of `errorBean`.
5. Inside the function create a variable declaration called `variables.errors` and instantiate an empty array.
6. Below this return the `this` scope. Your function should look similar to this:

```
public errorBean function init(){
    variables.errors = [];
    return this;
}
```

7. Next create a function called `addError` that is `public` and returns `void`. Create 2 required string arguments called `message` and `field`.
8. Inside the function append a struct to the `variables.error` variable. The structure should consist of 2 keys, `message` and `field`. Set each key to their related `argument` value. This function will be used to add errors to the object. Your function should look similar to this:

```
public void function addError(required string message, required string
field){
    arrayAppend(variables.errors,{message=arguments.
message,field=arguments.field});
}
```

9. Next we will create a function called `getErrors` which will return all errors stored in the object. The function will be `public`, return an `array`, and accept no arguments.

10. Inside the function return the `variables.errors` variable. Your function should look similar to this:

```
public array function getErrors(){
     return variables.errors;
}
```

11. We now need to create a function that will tell us if there are any errors stored in the object. Create a `public` function called `hasErrors`. The function will return a `boolean` value and will accept no arguments.

12. Inside the function create an `if` statement that checks the array length of the `variables.error` array. If there is a length, return `true`. If there is not a length return `false`. Your function should look similar to this:

```
public boolean function hasErrors(){
     if(arrayLen(variables.errors)){
          return true;
     }
     else{
          return false;
     }
}
```

13. Finally we will create a function that removes all errors from the object. Create a function called `clearErrors` that is `public`, does not return anything and accepts no arguments.

14. Inside the function assign an empty array to the `variables.errors` variable.

15. You have now created an encapsulated object. We will test that this object works in one of the other Hands On's but for now confirm that your cfc looks similar to the following:

```
component{
     public errorBean function init(){
          variables.errors = [];
          return this;
     }

     public void function addError(required string message, required
string field){
          arrayAppend(variables.errors,{message=arguments.
message,field=arguments.field});
     }



     public array function getErrors(){
          return variables.errors;
     }

     public boolean function hasErrors(){
          if(arrayLen(variables.errors)){
               return true;
          }
          else{
               return false;
          }
     }

     public void function clearErrors(){
          variables.errors = [];
     }
}
```

8

Intro to ORM

By Sam Farmer

About Sam Farmer

Growing up I never imagined I would play bass guitar for the Dave Matthews Band. And indeed it never happened.

But I have become a passionate and pretty good web developer. I have developed applications for 15 years mostly using ColdFusion, JavaScript, Oracle, SQLServer and MySQL. I am a happily married father of two daughters living in Washington, DC. Currently I am a Senior Applications Architect at FirstComp.

Introduction

Object-Relational Mapping (ORM) allows you to work with objects and have them saved to the database automatically. It can greatly simplify create-read-update-delete (CRUD) operations and make your code more object-oriented. Under the hood, ColdFusion uses the industry leading ORM framework called Hibernate.

Configuration

Application.cfc

Two settings are needed in the `Application.cfc` file to make an application use ORM: a `data source` and `ormEnabled` set to true.

The `cfartgallery` data source, which uses Derby DB, is installed with the developer version of ColdFusion. ORM will work with any modern database.

Other settings for ORM are set in a structure called `ormSettings`. In the example below we are setting `logsql` to true. When set to true, the SQL created by Hibernate will display in the ColdFusion debugging information. There are settings for `cfclocation` to specify where the application's persistent cfc's are placed, and `dbcreate` will automatically keep changes made to your model and the database.

The final setting, `invokeImplicitAccessor`, will be discussed later.

```
component {

    this.datasource = "cfartgallery";
    this.ormEnabled = true;
    this.ormSettings = { logsql : true };
    this.invokeImplicitAccessor = true;

}
```

Entities

An entity is a class that (in basic operations) maps to a single database table. In this first example, we create an entity that will be referenced as "art" in our code and will map to the art table in the artgallery database. The properties map to the columns in the table and can be a subset of all columns as well as having other properties that are not persistent and marked with the attribute `persistent=false`.

Each persistent cfc must have a primary key; this is denoted with `fieldtype="id"` (for composite primary keys use multiple properties with `fieldtype="id"`). Most of the time, the primary key will have a database auto-generated primary key; in this case, we are using the Hibernate powered generator.

Each property will have getters and setters automatically generated. Both getters and setters can be overwritten. The client's preference is that the art name always be in uppercase, rather than adjusting it everywhere it is displayed; the example below overwrites the getter to do so. Other functions can be added to the entity as well. The example below has a `getProfit` function that when a piece is sold, it calculates the amount of profit.

The property name maps to a database column via the column attribute. When the column attribute is not provided, ColdFusion uses the value in the name as the column.

art.cfc

```
component persistent="true" {
    property name="id" column="artid" fieldtype="id"
generator="increment";
    property name="name" column="artName" ormtype="string";
    property name="description" ormtype="text";
    property name="price" ormtype="double";
    property name="isSold" ormtype="boolean";

    property name="artist" fieldtype="many-to-one" cfc="artist";

    function getName() {
      return uCase( variables.name );
    }

    function getProfit() {
      if ( getIsSold() ) {
        return getPrice() * 0.2;
      } else {
        return 0;
      }
    }
}
```

Relationships

The final property establishes a relationship between art and artist, in this example, a `many-to-one`. It is similar to a foreign-key relationship in a database between two tables, except now it's between two objects. The artist entity also has a `one-to-many` relationship to art. Other relationship types include `one-to-one` and `many-to-many`, although it's generally better to use two `one-to-many` relationships than a `many-to-many`.

ColdFusion automatically adds functions to objects for controlling relationships. For the art `many-to-one` these are `getArtist()`, `hasArtist()`, and `setArtist()`. For other relationships, replace "artist" with the relationship name. `getArtist` will return an array of objects of the artist, `hasArtist` is Boolean in determining whether there is a artist associated with the art record, and `setArtist` takes an object and sets it to the piece of art.

The artist `one-to-many` will create `addArt()`, `getArt()`, `hasArt()`, `removeArt()`, `setArt()`. The two new functions are due to a `one-to-many` having the options for many objects. `addArt` takes an art object and adds it to the relationship; likewise, `removeArt` will remove an art object from the relationship.

The art relationship below also has a `cascade` property with the value of "delete". When an artist is deleted, it will find all art entities related to it and delete them. While useful, this can cause unintended consequences, so use carefully, look at the other options for `cascade`,and don't forget you can leave it off altogether.

artist.cfc

For artist, we specify the table with the table attribute. One advantage of ORM is the ability to change database tables and column names to more programmer friendly terms.

```
component persistent="true" table="artists" {


    property name="id" column="artistID" fieldtype="id"
generator="increment";
    property name="firstname" ormtype="string";
    property name="lastname" ormtype="string";


    property name="art" fieldtype="one-to-many" cfc="art"
fkcolumn="artistID" cascade="delete";


}
```

Retrieving Data

To get data, there are two functions we can use: `entityLoad` and `ormExecuteQuery`. First, let's use `entityLoad()`, which has many signatures. In the example here, the first argument is the `entity`, the second a `structure` of values to match exactly (to get all pass an empty structure), the third what to order data by.

```
<cfset artists = entityLoad( "artist", { firstname: "Jeff" },
"lastname" )>
```

This will return an array of objects where the artists first name is Jeff.

Outputting

```
<cfoutput>

    <cfloop array="#artists#" index="artist">
        <h4>#artist.firstName# #artist.lastname# #artist.id#</h4>
        <cfif artist.hasArt()>
            <ul>
                <cfloop array="#artist.getArt()#" index="art">
                    <li>#art.name# #dollarFormat( art.price )#</li>
                </cfloop>
            </ul>
        </cfif>
    </cfloop>

</cfoutput>
```

Within the loop, artist represents a single object. We can reference the data using the automatic getters in either the `getProperty()` format or, when `invokeImplicitAccessor` is set to true in `Application.cfc`, by just the property. The same concept will be used for setters later.

The output will look like this:

```
Jeff Baclawski
- Bowl of Flowers $11,800.00
- 60 Vibe $25,000.00
- Naked $30,000.00
- Sky $15,000.00
- Slices of Life $20,000.00
- sda $10.00
```

More Retrieving Data with HQL

Hibernate Query Language (HQL) is a language similar to SQL used for more complex data retrieval methods. In the following example, we use “like” to get all records in which the first name begins with A. `ormExecuteQuery` will return an array of objects the same way as `entityLoad`.

```
<cfset artists = ormExecuteQuery( "FROM artist WHERE firstname like
:firstname ORDER BY lastname", { firstname: "A%"}) />
```

Serializing as JSON

The `serializeJSON` function can serialize entities in JavaScript Object Notation (JSON) format.

```
serializeJSON( artists );
```

will produce:

```
[{"firstname":"Jeff","id":4,"lastname":"Baclawski"}]
```

Working with Data

Add Record

To add a record, use the `entityNew` function with the first argument being the entity name and an optional second argument being a structure (a map in other languages) of data. After adding data, call the `entitySave` function.

As we set up a relationship between `artist` and `art`, keep in mind that when adding an `artist`, you need to set their `art` as well. To do so, use the `addXXX()` function. In the example below, this is achieved with the `addArt` function.

```
transaction {
    art = entityNew( "art", { name : "Painting of TV", price : 200,
isSold : false } );
    entitySave( art );

    artist = entityNew( "artist", { firstname : "John", lastname : "Doe"
} );
    artist.addArt( art );

    entitySave( artist );
}
```

Update Record

To update a record, retrieve a single object. To do so, use a different signature with the third argument as a boolean which needs to be true to retrieve a single record. Once we have a single record, values can be changed with the implicit setters. Any value changed will be saved to the database at the end of the transaction. If you want to retrieve a value saved in a transaction, do so after the transaction has ended. To rollback any changes, use `transactionRollback()`.

```
transaction {
    artist = entityLoad( "artist", 100, true );
    artist.firstname = "Fred";
}
```

Delete Record

To delete a record, retrieve a single object and then pass it to the `entityDelete` function.

```
transaction {
    artist = entityLoad( "artist", 100, true );
    entityDelete( artist );
}
```

Vital Tips

ormReload()

After any changes to entity configuration (adding/removing a property, etc.), it is necessary to call `ormReload()` for it to take affect. During development, it's common to place this in the `onRequestStart` function of `Application.cfc`. Depending on the size of application, it can take 100 milliseconds to 5 seconds to run.

Null Values

There are times when ColdFusion cannot return an entity; instead, the value assigned will be null. Consider the following code:

```
art = entityLoad( "art", 9999999, true );
```

It is asking for one entity where the id is ridiculously high. Instead of erroring, it will set the art variable to null. The `isNull` variable can test for this:

```
art = entityLoad( "art", 9999999, true );
if ( isNull( art ) ) {
  // no art entity provided. write code to deal with it.
}
```

Dumping Objects

When using `writedump`, ColdFusion will try and display the data from relationships. Sometimes this can be an attempt to display way more data than desired. To prevent this, provide some of the other attributes:

```
writeDump( var=artists, top=2, showudfs=false );
```

Hands On 18

By Simon Free

In this hands on example, you are going to create the ORM entities for the Blog Section.

Functions Used: ormReload

1. Open up the `/www/Application.cfc` file in your code editor.
2. Locate the `this.sessionTimeout` variable declaration on or around line 6.
3. Below this, create the following new variables in the `this` scope:
   - this.ormEnabled = true
   - this.ormSettings = { logsql = true, dbcreate="update", cfclocation="com/entity" }
   - this.invokeImplicitAccessor = true;
4. Your code should look similar to this:

```
this.sessionTimeout = CreateTimeSpan(0, 0, 30, 0);
this.ormEnabled = true;
this.ormSettings = {
     logsql = true,
     dbcreate="update",
     cfclocation="com/entity"
     };
this.invokeImplicitAccessor = true;
```

5. Go to the `onRequestStart` Function and add an `ormReload()` function call inside of the `if` statement. Your code should look similar to this:

```
function onRequestStart(string targetPage){
     if(structKeyExists(url,'reload')){
          onApplicationStart();
          ormReload();
     }
 }
```

6. Create a new file called `blogPost.cfc` in the `/www/com/entity/` folder.

7. Open up the `/www/com/entity/blogPost.cfc` file in your code editor.
8. Declare the file as a component by adding the component script tags:

```
component{
}
```

9. Add an additional property to the component declaration, `persistent`, and set that value to true. Your code should look similar to this:

```
component persistent="true"{
}
```

10. Inside the component definition, enter the following code:

```
property name="id" column="blogpostid" fieldtype="id"
generator="increment";
property name="title" ormtype="text";
property name="summary" ormtype="text";
property name="body" ormtype="text";
property name="dateposted" ormtype="timestamp";
property name="createdDateTime" ormtype="timestamp";
property name="modifiedDateTime" ormtype="timestamp";
property name="deleted" ormtype="boolean";


property name="comments" singularname="comment" fieldtype="one-to-many"
cfc="blogComment" fkcolumn="blogpostid" cascade="all";
```

11. Create a new file called `blogCategory.cfc` in the `/www/com/entity/` folder.
12. Add the following code to the file:

```
component persistent="true"{
    property name="id" column="blogCategoryid" fieldtype="id"
generator="increment";
    property name="name" ormtype="string";
}
```

13. Create a new file called `blogComment.cfc` in the `/www/com/entity/` folder.

14. Add the following code to the file:

```
component persistent="true"{
    property name="id" column="blogCommentid" fieldtype="id"
generator="increment";
    property name="author" ormtype="string";
    property name="comment" ormtype="text";
    property name="createdDateTime" ormtype="timestamp";
    property name="deleted" ormtype="boolean";

    property name="blog" fieldtype="many-to-one" cfc="blogPost";

}
```

15. Open up the `/www/blog.cfm` page in your browser and append `?reload=1` to the url. The `ORMReload` function will now have been called and the database tables will have been created.

Hands On 19

By Simon Free

In this hands on example, you are going to add the list and create functionality in the Admin for the blog posts and categories.

Tags Used: <cfset>
Functions Used: EntityLoad, EntityNew, now

1. Open up the `/www/admin/content/blog/editBlogPost.cfm` file in your code editor.
2. Locate the `Edit Entity` comment tag on or around line 39.
3. Load the entity into a `blogPost` variable by adding the following code:

```
<cfset blogPost = EntityLoad('BlogPost',form.id,true) />
```

4. Set the blog post properties from the `form` scope by adding the following lines of code below the `<cfset>`:

```
<cfset blogPost.title = form.title />
<cfset blogPost.summary = form.summary />
<cfset blogPost.body = form.body />
<cfset blogPost.datePosted = form.datePosted />
<cfset blogPost.modifiedDateTime = now() />
```

5. Next, add the logic to create a new blog post. Locate the comment tag that reads `Create Entity` on or around line 47.
6. Create a new blog post object and set it to the `blogPost` variable by adding the following line of code:

```
<cfset blogPost = EntityNew('BlogPost') />
```

7. Set the object properties from the form scope by adding the following lines of code below the `<cfset>`:

```
<cfset blogPost.title = form.title />
<cfset blogPost.summary = form.summary />
<cfset blogPost.body = form.body />
<cfset blogPost.datePosted = form.datePosted />
<cfset blogPost.createdDateTime = now() />
```

8. At this point, the `blogPost` object has all the information it needs and will be saved successfully. The next step is pull the blog post information for the edit form. To do this, find the `Get Entity Data` comment tag.
9. Create a blog Post object using the `URL` variable `ID` and set it to the `blogPost` variable by adding the following line of code:

```
<cfset blogPost = EntityLoad('BlogPost',url.id, true ) />
```

10. Now set the `blogPost` properties into the `form` scope so that they will display in the form. To do this, add the following lines of code below the `<cfset>` tag:

```
<cfset form.id = blogPost.id />
<cfset form.title = blogPost.title />
<cfset form.summary = blogPost.Summary />
<cfset form.body = blogPost.body />
<cfset form.datePosted = blogPost.datePosted />
```

11. The add/edit process is now complete, but before you test it, update the listing page to pull the entities. Open up the `/www/admin/content/blog/listblogpost.cfm` file in your code editor.
12. Find the comment `Pull Blog Posts` and add a `<cfset>` below, that calls `EntityLoad` on `blogPost` and sets the value to a variable called `blogPosts`. Do this by adding the following line of code:

```
<cfset blogPosts = EntityLoad('BlogPost') />
```

13. Next, output the information to the user. Find the `Title` comment tag and output the title property of the `blogPost` object. Do this by adding the following code below the `Title` comment tag:

```
#blogPost.title#
```

14. Then, output the `datePosted` value; add the following line of code below the `Date Posted` comment tag:

```
#dateFormat(blogPost.datePosted,"mm/dd/yyyy")#
```

15. The final stop to complete the listing page is to output the blog post ID in the edit link so that the edit page knows which blog post is being edited. To do this, append the following line of code to the `ID` URL attribute in the `<a>` tag under the `Edit Post` comment:

```
#blogPost.id#
```

16. The admin is now able to create and edit `blogPosts`. Navigate to `/www/admin/` in your browser and login with the Email Address of `admin@myWebsite.com` and password of `admin`. Once logged in, click on 'Blog'.
17. Click on 'Add Blog Post' and create a new blog.
18. Go back to the listing page and edit your blog.

Homework

Update the `blogCategory` files in the admin so that blog categories can be created and edited.

Hands On 20

By Simon Free

In this hands on example, you are going to update the front end of the web site to pull in the blog information you have entered through the admin.

Tags Used: <cfloop>, <cfoutput>, <cfparam>, <cfset>, <cfif>
Functions Used: EntityLoad, dateFormat, arrayLen, timeFormat, EntityNew, EntitySave

1. Open up the `/www/blog.cfm` file in your code editor.
2. First, pull all blog posts (in real life, you would only want to pull 10 or so entries per page). To pull all entries, call the `EntityLoad` function passing in the string 'blogPost' and assigning that to a `blogPosts` variable. Do this by replacing the <cfquery> block with the following line of code:

```
<cfset blogPosts = EntityLoad('blogPost') />
```

3. Next, loop over the objects and output them. To do this, we will use a `<cfloop>` tag. Locate the `<cfoutput>` tag and replace it with the following line of code:

```
<cfloop array="#blogPosts#" index="blogPost">
```

4. Replace the closing `</cfoutput>` tag with a closing `</cfloop>` tag, on or around line 39.
5. Before proceeding, wrap the `<cfloop>` block with a `<cfoutput>` so that the data will be output to the screen.
6. You now need to start replacing the placeholder information with the variables from the `blogPost` object. First, append the `ID` to the `URL` so that you can view the more detailed blog post later. Update the `href` attribute of the `<a>` tag and append `#blogPost.id#` to the link. The link should now read:

```
blogpost.cfm?id=#blogpost.id#
```

7. Next, replace the `myBlog.title` variable with `blogPost.title`.
8. Replace the `myBlog.datePosted` variable with `blogPost.datePosted`.
9. Replace the `myBlog.summary` variable with `blogPost.summary`.
10. The final step is to output the number of comments that the blog post contains. To do this, call the `getComments()` function on the `blogPost` object and use the `arrayLen` function to count how many entries were returned by replacing the number '12' with the following code:

```
#arrayLen(blogPost.getComents())#
```

11. Go to the `/www/blog.cfm` page in your browser and confirm that you now see all your blog posts listing on the page.
12. Next, update the blog post detail page to display the relevant blog post information. Open up the `/www/blogpost.cfm` file in your code editor.
13. At the top of the file, load in the blog post object and set it to a variable called `blogPost` using the following code:

```
<cfset blogPost = EntityLoad('blogPost',url.id,true) />
```

14. Then replace the placeholder content with the real content. Locate the `<h2>` tag and replace its content with `#blogPost.title#`.
15. Replace the date with `#dateformat(blogPost.dateposted,'mm/dd/yyyy')#`.
16. Replace the blog body placeholder text; this is the next five `<p>` tags. Delete these and replace them with `#blogPost.body#`.
17. The last step is to put the blog post's ID value into the 'Export to PDF' link. Append the following code to the hef value of the `<a>` tag:

```
#blogPost.id#
```

18. Your code should look similar to this:

```
<h2>
    #blogPost.title#
</h2>
<p>
    <strong>Date Postedv/strong>: #dateformat(blogPost.datePosted,
'mm/dd/yyyy')#
</p>
#blogPost.body#
<p>
    <a href="exportToPDF.html?id=#blogPost.id#" target="_new">Export to
PDF</a>
</p>
```

```
<h3>
    Comments (1)
</h3>
```

19. You need to output how many comments the blog has, which is done by doing the same thing you have done previously by using the `arrayLen` function. Replace the number '2' with the following code:

```
#arrayLen(blogpost.getComments())#
```

20. Now, loop over the comments and output them. To do this, use a `<cfloop>` tag. Just after the HTML comment tag, `Start Comment`, create a `<cfloop>` tag with the following attributes:
    - **array**: #blogPost.getComents()#
    - **index**: comment
21. Place a closing `</cfloop>` tag before the HTML comment tag, `End Comment` and remove the other `<li>` tag.
22. The comment object will now be available in the comment variable. Replace the date with the following code:

```
#dateFormat(comment.createdDateTime,'mm/dd/yyyy')#
```

23. Replace the time with the following code:

```
#timeformat(comment.createdDateTime,'short')#
```

24. Replace 'Simon Free' with the following code:

```
#comment.author#
```

25. And finally, replace the placeholder text in the `<p>` tag with the following code:

```
#comment.comment#
```

26. Your code should look similar to this:

```
<ul>
     <cfloop array="#blogPost.getComents()#" index="comment">
     <li>
          <p>
               Posted On: #dateFormat(comment.createdDateTime,
'mm/dd/yyyy')# at #timeformat(comment.createdDateTime,'short')# By
#comment.author#
          </p>
          <p>
               #comment.comment#
          </p>
          <div class="clr hline"> </div>
     </li>
     </cfloop>
</ul>
```

27. Next, wrap the entire `<div class="left>` block in a `<cfoutput>` tag block. The opening `<cfoutput>` will start on or around line 25. The closing `</cfoutput>` should be placed on or around line 93.

28. If you were to run this page now, you would see no comments as there are no comments in the system. The next step is to allow users to submit comments. Go to the top of the file and create a `<cfparam>` tag with the following attributes:

    - **name**: form.submitted
    - **default**: 0

29. Next, add the logic to save the comment only if the form has been submitted. You can check if the form has been submitted by looking at the `form.submitted` value. After the `<cfset>` tag on or around line two, create a `<cfif>` tag, which checks if `form.submitted` is true. Your code should look similar to this:

```
<cfparam name="form.submitted" default="0" />
<cfset blogPost = EntityLoad('blogPost',url.id,true) />
<cfif form.submitted>


</cfif>
```

30. Inside the <cfif> block, create a new instance of the `blogComment` entity and set the author and comment values from the `form` scope. Your code should look similar to this:

```
<cfset comment = entityNew('blogComment') />
<cfset comment.author = form.author />
<cfset comment.comment = form.comment />
```

31. Next, set the `createdDateTime` value using the `now()` function and add the comment to the `blogPost` object. Your code should look similar to this:

```
<cfset comment = entityNew('blogComment') />
<cfset comment.author = form.author />
<cfset comment.comment = form.comment />
<cfset comment.createdDateTime = now() />
<cfset blogPost.addComment(comment) />
```

32. Now that you have created the comment and added it to the blog post, you need to save these changes by calling `entitySave` on the `blogPost` object. Add the following line of code after your last <cfset> tag:

```
<cfset EntitySave(blogPost) />
```

33. Your final code block should look similar to this:

```
<cfif form.submitted>
    <cfset comment = entityNew('blogComment') />
    <cfset comment.author = form.author />
    <cfset comment.comment = form.comment />
    <cfset comment.createdDateTime = now() />
    <cfset blogPost.addComment(comment) />
    <cfset EntitySave(blogPost) />
</cfif>
```

34. Locate the <form> tag on or around line 78 and append the following code to the action attribute:

```
#blogPost.id#
```

35. Open up the `/www/blog.cfm` page in your browser. Navigate to a blog post and enter a comment. You should now see your comment displayed in the comment section.

Hands On 21

By Simon Free

In this hands on, we are going to add categories to blog posts. The logic for this involves creating a linking entity to which both the blog post and category will be associated. This is an alternative method to using a many-to-many relationship.

Tags Used: <cfloop>, <cfset>
Functions Used: EntitySave, for, listAppend, EntityDelete

1. In order to be able to associate multiple categories to a blog post and also be able to associate multiple blog posts to a category, we need to create a linking entity. Create a new file called `blogPostCategory.cfc` in the `/www/com/entity/` folder.
2. Open up the `/www/com/entity/blogPostCategory.cfc` file in your code editor.
3. Add the following lines of code:

```
component persistent="true" {

    property name="blogPost" fieldtype="id,many-to-one" cfc="blogPost"
column="blogPostid" ;

    property name="blogCategory" fieldtype="id,many-to-one"
cfc="blogCategory" column="blogCategoryid" ;

}
```

4. Now that the linking entity is created, you need to associate that entity to both the `blogpost` entity and the `blogCategory` entity. To do this, open up the `/www/com/entity/blogPost.cfc` file in your code editor.
5. Locate where the comment relationship is defined. This should be on or around line 11.
6. After this line of code add the following line of code:

```
property name="categories" fieldtype="one-to-many" cfc="blogPostCategory"
fkcolumn="blogpostid";
```

7. The line you just entered associates the blog post to the `blogPostCategory` entity. The next step is to do the same for the `blogCategory`. Open up the `/www/com/entity/blogCategory.cfc` file in your code editor.
8. Locate the `name` property definition on or around line 2 and add the following line of code below it:

```
property name="posts" fieldtype="one-to-many" cfc="blogPostCategory"
fkcolumn="blogCategoryid";
```

9. Now that the entity properties have been set, reload the application so the ORM changes can take effect. To do this, go the `/www/` page in your browser and append `?reload=1` to the URL.
10. Your application has now created the new linking table and you have the new entity properties at your disposal. Next, let's write the logic which will save categories to a blog post. Open up the `/www/admin/content/blog/editblogpost.cfm` file in your code editor.
11. Locate the `Add Category to Entity` comment.
12. Below the comment, instantiate a new instance of the `BlogPostCategory` and call the variable `blogPostCategory`.
13. Below that line, load in a `blogCategory` object and use the variable `categoryID` as the identifier and call the variable `blogCategory`.
14. Your code should look similar to this:

```
<cfloop list="#form.categories#" index="categoryID">
    <!--- Add Category to Entity --->
    <cfset blogPostCategory = EntityNew('blogPostCategory') />
    <cfset blogCategory = EntityLoad('blogCategory',categoryID,true) />
</cfloop>
```

15. After the `blogCategory` variable has been created, set the values of the `blogPostCategory`. The first value to be set is the `blogCategory`. To do this, set the `blogCategory` property value of the `blogPostCategory` object to the `blogCategory` variable. Your code should be similar to this:

```
<cfset blogPostCategory.blogCategory = blogCategory />
```

16. Next, set the `blogPost` property of `blogPostCategory` with the `blogPost` variable which was saved earlier in the page.
17. Once this is done, save the entity. Simply call the `entitySave` function and pass in the `blogPostCategory` object. Your final code should look similar to this:

```
<cfloop list="#form.categories#" index="categoryID">
    <!--- Add Category to Entity --->
    <cfset blogPostCategory = EntityNew('blogPostCategory') />
```

```
    <cfset blogCategory = EntityLoad('blogCategory',categoryID,true) />
    <cfset blogPostCategory.blogCategory = blogCategory />
    <cfset blogPostcategory.blogPost = blogPost />
    <cfset entitySave(blogPostCategory) />
</cfloop>
```

18. Now that this logic is complete, we can associate a category with a blog post and save it. Open up the `/www/admin/content/editBlogPost.cfm` page in your browser.
19. Enter some blog information, select a category, and click 'save'.
20. You will see on the listing page that the blog post has been saved. Click on the blog post you just saved and review the data on the screen.
21. You will notice that the category is not pre-checked. This is not because the category has not been saved, but because we have not put the logic in place to generate the list of category ID's that the page needs. To do this, you need to set a form value called 'categories'. Locate the `Get Entity Data` comment tag.
22. Below the last `<cfset>` tag in the `<cfif>` block, create a new `<cfset>` tag that sets `blogpost.categoryids` to the variable `form.categories`. Your code should look similar to this:

```
<cfset form.categories = blogPost.categoryids  />
```

23. If you were to run the page now, you would generate an error. The functionality to get the category ID's is not present in the `blogPost` entity. We must first create that functionality by adding a custom function into the `blogPost` entity. Open up the `/www/com/entity/blogPost.cfc` file in your code editor.
24. Below the last property definition, create a public function that returns a string and is called `getCategoryIDs`.
25. Inside the function, create a variable called `categoryList`, assign it to the `var` scope and set it to an empty string. Your code should look similar to this:

```
public string function getCategoryIDs(){
    var categoryList = '';


}
```

26. Next, check that the `blogPost` actually has some categories from which to generate a list. To do that, create an `if` statement that calls the `hasCategories()` function.

27. Inside of the `if` statement, begin looping over the categories and concatenating a string of their ID's. To do this, create the following `for` loop inside the `if` statement:

```
for ( var categoryPost in getCategories() ) {


}
```

28. Inside the `for` loop, get the ID of the `blogCategory` and append it to the `categoryList` variable. To get the category ID access the `blogCategory` property and then access the ID value from the returned `BlogCategory` object. Use the `listAppend` function to append that result with the `categoryList` variable. The code will look similar to this:

```
categoryList = listAppend( categoryList, categoryPost.blogCategory.id);
```

29. Finally, return the generated list by calling the `return` statement and passing it the `categoryList` variable.

30. Our final function should look similar to this:

```
public string function getCategoryIDs(){
    var categoryList = '';
    if ( hasCategories() ) {
        for ( var categoryPost in getCategories() ) {
            categoryList = listAppend( categoryList, categoryPost.
blogCategory.id);
        }
    }
    return categoryList;
}
```

31. Open up the `/www/admin/content/blog/listblogposts.cfm` page in your browser and append the `?reload=1` value to the URL.

32. Click on the blog post and notice that the category is now checked.

33. If you were to click 'save' again, your blog post would end up with duplicate categories associated to it. This is because we have added logic to 'add categories', but we have not checked if the blog post is already associated to the category. The easiest way to solve that is to remove all categories from the blog post when editing, then re-add the new categories. To do this, open up the `/www/admin/content/blog/editBlogPost.cfm` file in your code editor.

34. Find the `Edit Entity` comment tag and go to the last `<cfset>` tag. After the `<cfset>` tag, create a `<cfloop>` tag with the following attributes:
    - **array**: #blogPost.getCategories()#
    - **index**: category
35. Inside the `<cfloop>`, create a `<cfset>` tag that calls the `entityDelete` function and pass it in the variable `category`. Your final code should look similar to this:

```
<cfloop array="#blogPost.getCategories()#" index="category">
    <cfset entityDelete(category) />
</cfloop>
```

36. Go back to the `/www/admin/content/blog/listblogposts.cfm`, click on a blog post, and save the blog to confirm you have no errors.

Homework

Create a function similar to the `getCategoryIDs` function. Instead of returning ID's, create a function that returns a list of the category names. Replace the category output, which is currently 'ColdFusion', in `/www/blog.cfm` with this new function call.

9

Mail

By Guust Nieuwenhuis

About Guust Nieuwenhuis

Guust Nieuwenhuis works as a consultant for Trasys, and is seconded to the European Commission's DG Enterprise and Industry. His tasks include developing web applications used by the different Commission services as well as by the citizens of the European Union. Guust also started his own company, Orange Lark, which delivers web solutions (sites, applications and consultancy) to various international clients.

Within the ColdFusion community Guust is a very active member. He speaks at conferences (CFCamp 2012, MuraCon 2012, Scotch on the Road 2012, BarCamp Antwerp) and usergroups (ColdFusion UserGroup Belgium, Dutch ColdFusion UserGroup, CFMeetup, The Mura Show), organises conferences (Scotch on the Rocks) and usergroup meetings (ColdFusion UserGroup Belgium) and shares his opinions, discoveries, experiences on his blog and Twitter. As a recognition of all this, Guust has been invited by Adobe to join there Adobe Community Professional program in 2011.

In his free time, he plays the double bass and drums both in bands as in symphony orchestra's and likes meeting friends for a chat, game or drink. When he still has some time left, he mainly spends it behind his computer to fulfil his hunger for the latest trends in IT.

Configure Mail Settings

Before you can send emails with ColdFusion, a mail server you want to use needs to be set. Configuring the mail server can be done in the ColdFusion Administrator.

If you do not have a mail server of your own, your localhost can act as a mail server. ColdFusion will act normally, but since there is no mail server set up on your localhost, emails will not arrive at their destination. Another option is to use the mail server of Gmail (a Gmail account is required for this). The settings for this mail server can be found at the end of this section.

Mail Server Settings

In the ColdFusion Administrator, on the left menu, below "Server Settings", you will find "Mail". To configure the mail server, specify the host name or IP address of the mail server in the first field. In case the mail server requires authentication, you will need to provide a "user name" and "password" as well.

The "Verify mail server connection" option should be selected so that you will be informed if ColdFusion finds the mail server after you saved the settings. Do not enable this option when you use localhost as your mail server.

The default port of a mail server is port 25. If this is not the case for your mail server, you can provide the correct port for your mail server.

Now that you have configured the mail server, save the settings by clicking the "Submit Changes" button at the bottom right (or top right).

When you enabled the "Verify mail server connection" option, ColdFusion will inform you if it was able to connect to the mail server. If ColdFusion was able to connect to the mail server, you are done. If not, review the settings or choose a different mail server.

Gmail Mail Server Settings

- **Mail server:** smtp.gmail.com
- **User name**: your Gmail email address
- **Password**: your Gmail password
- **Enable TLS**: yes

Please note that Gmail will automatically rewrite the "from" email address to your gmail address.

Send Email

For sending emails ColdFusion provides the `cfmail` tag. The tag has three required attributes:

- **from**: the email address from which you want to send the email.
- **to**: the email address to which you want to send the email.
- **subject**: the subject of the email.

The content of your email will be placed between the opening and closing tag:

```
<cfmail from="from@domain.com" to="to@domain.com" subject="My first email
sent with ColdFusion">
   Hello,
   This is my first email sent with ColdFusion!
</cfmail>
```

Sending a copy or a blind copy of your email to someone can be done via the `cc` attribute and `bcc` attribute. Like the `to` attribute, specify a valid email address:

```
<cfmail from="from@domain.com" to="to@domain.com" cc="cc@domain.com"
bcc="bcc@domain.com" subject="My first email sent with ColdFusion">
   Hello,
   This is my first email sent with ColdFusion!
</cfmail>
```

Multiple recipients

If you want the email to be sent to multiple people at once, you can specify all email addresses as a comma-separated list in the `to` attribute:

```
<cfmail from="from@domain.com" to="to01@domain.com,to02@domain.com"
subject="An email sent to multiple people">
   Hello,
   This is an email sent to multiple people!
</cfmail>
```

HTML Email

Text emails are simple and easy, but sometimes you need more control over the presentation of your email, such as with a newsletter or a promotional email. What you then need is an HTML email.

Add the `type` attribute to your `cfmail` tag and give it the value "html". Now you can use HTML in the body of your email.:

```
<cfmail from="from@domain.com" to="to@domain.com" subject="My first HTML
email sent with ColdFusion" type="html">
   <html>
        <head>
           <style type="text/css">
           body {
                font-family:sans-serif;
                font-size:12px;
                color:navy;
           }
           </style>
        </head>
        <body>
           <p>Hello,</p>
           <p>This is my first HTML email sent with ColdFusion!</p>
           <p>Email sent by <a href="http://www.coldfusion.com" ><img
src="http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/images/
shared/product-totems/80x80/totem-coldfusion-10-80x80.png" /></a></p>
       </body>
   </html>
</cfmail>
```

The email client of the recipient will render the html and display it to the recipient.

HTML emails are very powerful, but there are two things you need to know:

- Not all email clients support all features of HTML; some only support the very basics features of HTML.
- Not all email clients support HTML emails or the recipient disabled HTML emails in their email client.

In those cases, the recipient will see the source code of your HTML email, which is not always readable.

More information on the first issue can be found at www.emailology.org, including an email boilerplate, tips and tricks, and a list of what's supported by which email clients.

For the second issue, there is an easy solution: send an email with both text and HTML.

HTML and Text Email

The best way to make sure that the recipient of your email will receive the right version for his email client is to send both a text and an HTML version of the email. This can be done by adding two `cfmailpart` tags to the body of your email. The only required attribute of this tag is the `type` attribute that specifies which version the `cfmailpart` represents: "text" for the text email and "html" for the HTML email.

An optional attribute for the `cfmailpart` tag with the `type` attribute set to "text" is `wraptext`. This attribute specifies the maximum line length, in characters of the mail text:

```
<cfmail from="from@domain.com" to="to@domain.com" subject="My first email
sent with ColdFusion" type="html">


   <cfmailpart type="text" wraptext="60">
       Hello,
       This is my first text and HTML email sent with ColdFusion!
       Email sent by ColdFusion
   </cfmailpart>


   <cfmailpart type="html">
       <html>
            <head>
                <style type="text/css">
                body {
               font-family:sans-serif;
               font-size:12px;
               color:navy;
               }
               </style>
            </head>
            <body>
           <p>Hello,</p>
           <p>This is my first text and HTML email sent with
ColdFusion!</p>
```

```
            <p>Email send by <a href="http://www.coldfusion.com" ><img
src="http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/
images/shared/product-totems/80x80/totem-coldfusion-10-80x80.png"
alt="ColdFusion" /></a></p>
            </body>
        </html>
    </cfmailpart>
</cfmail>
```

Adding Attachments

By using the `cfmailparam` tag inside your `cfmail` tag, you can add attachments to your email:

```
<cfmail from="from@domain.com" to="to@domain.com" subject="Q1 Financial
Results" type="html">

    <cfmailpart type="text" wraptext="60">
        Dear,
        In attachment you will find the financial results of Q1.
        Email sent by ColdFusion
    </cfmailpart>

    <cfmailpart type="html">
        <html>
            <head>
                 <style type="text/css">
                body {
                font-family:sans-serif;
                font-size:12px;
                color:navy;
                }
                </style>
             </head>
```

```
            <body>
                <p>Dear,</p>
                <p>In attachment you will find the financial results of
Q1.</p>
                <p>Email sent by <a href="http://www.coldfusion.com"
><img src="http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/
images/shared/product-totems/80x80/totem-coldfusion-10-80x80.png"
alt="ColdFusion" /></a></p>
            </body>
        </html>
    </cfmailpart>


    <cfmailparam file="c:\Documents\Q1FinancialResults.pdf">


</cfmail>
```

Through the optional `type` attribute, you can indicate what the MIME media type of the attachment is; with the optional `disposition` attribute, you can indicate if you want the attachment to be presented inline ("inline") or as an attachment ("attachment"):

```
<cfmail from="from@domain.com" to="to@domain.com" subject="Q1 Financial
Results" type="html">


    <cfmailpart type="text" wraptext="60">
        Dear,
        In attachment you will find the financial results of Q1.
        Email send by ColdFusion
    </cfmailpart>


    <cfmailpart type="html">
        <html>
            <head>
                 <style type="text/css">
                body {
```

```
                font-family:sans-serif;
                font-size:12px;
                color:navy;
                }
                </style>
            </head>
            <body>
                <p>Dear,</p>
                <p>In the attachment you will find the financial results of
Q1.</p>
                <p>Email sent by <a href="http://www.coldfusion.com"
><img src="http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/
images/shared/product-totems/80x80/totem-coldfusion-10-80x80.png"
alt="ColdFusion" /></a></p>
            </body>
        </html>
    </cfmailpart>


    <cfmailparam file="c:\Documents\Q1FinancialResults.pdf"
type="application/pdf" attachment="attachment">


</cfmail>
```

For a list of all registered MIME media types, visit www.iana.org/assignments/media-types.

Send Bulk Emails

When you want to send an email to a lot of people, using the `to` attribute of the `cfmail` tag is not recommended. Several mail clients and mail servers will mark the message a SPAM when sending it to a large quantity of people; you may also not want to share the list of email addresses with your recipients.

By providing the `cfmail` tag with a query result that contains the email addresses, every recipient will receive a personal email:

```
<cfquery name="getList">
    SELECT email
```

```
   FROM stockholders
</cfquery>

<cfmail query="getList" from="from@domain.com" to="#email#" subject="Q1
Financial Results" type="html">


   <cfmailpart type="text" wraptext="60">
       Dear,
       In attachment you will find the financial results of Q1.
       Email sent by ColdFusion
   </cfmailpart>


   <cfmailpart type="html">
       <html>
           <head>
               <style type="text/css">
               body {
               font-family:sans-serif;
               font-size:12px;
               color:navy;
               }
               </style>
            </head>
            <body>
               <p>Dear,</p>
               <p>In the attachment you will find the financial results of
Q1.</p>
               <p>Email sent by <a href="http://www.coldfusion.com"
><img src="http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/
images/shared/product-totems/80x80/totem-coldfusion-10-80x80.png"
alt="ColdFusion" /></a></p>
           </body>
       </html>
   </cfmailpart>
```

```
   <cfmailparam file="c:\Documents\Q1FinancialResults.pdf"
type="application/pdf" attachment="attachment">

</cfmail>
```

You can personalize the email even more by passing additional information to the `cfmail` tag through the query result:

```
<cfquery name="getList">
   SELECT email, firstname, lastname
   FROM stockholders
</cfquery>

<cfmail query="getList" from="from@domain.com" to="#email#" subject="Q1
Financial Results" type="html">

<cfmailpart type="text" wraptext="60">
   Dear #firstname# #lastname#,
   In the attachment you will find the financial results of Q1.
   Email send by ColdFusion
</cfmailpart>

<cfmailpart type="html">
   <html>
       <head>
            <style type="text/css">
           body {
           font-family:sans-serif;
           font-size:12px;
           color:navy;
           }
           </style>
```

```
        </head>
        <body>
            <p>Dear #firstname# #lastname#, </p>
            <p>In the attachment you will find the financial results of
Q1.</p>
            <p>Email sent by <a href="http://www.coldfusion.com" ><img
src="http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/
images/shared/product-totems/80x80/totem-coldfusion-10-80x80.png"
alt="ColdFusion" /></a></p>
        </body>
    </html>
</cfmailpart>


<cfmailparam file="c:\Documents\Q1FinancialResults.pdf"
type="application/pdf" attachment="attachment">


</cfmail>
```

Using CFScript

In you prefer to writing in script over tags, you can use `cfscript` to send emails. ColdFusion provides a wrapper around the `cfmail` tag that is automatically available.

Create a new `Mail` object.

```
email = new Mail();
```

Set all properties through the setters.

```
email.setFrom("from@domain.com");
email.setTo("to@domain.com");
email.setSubject("My first email sent with cfscript");
```

Add the mail parts for text and html.

```
email.addPart( type="text", wraptext="60", body="
   Dear,
   In the attachment you will find the financial results of Q1.
   Email sent by ColdFusion
");

email.addPart( type="html", body="
   <html>
       <head>
           <style type='text/css'>
           body {
           font-family:sans-serif;
           font-size:12px;
           color:navy;
           }
           </style>
       </head>
```

```
       <body>
           <p>Dear #firstname# #lastname#, </p>
           <p>In the attachment you will find the financial results of
Q1.</p>
           <p>Email sent by <a href='http://www.coldfusion.com' ><img
src='http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/
images/shared/product-totems/80x80/totem-coldfusion-10-80x80.png'
alt='ColdFusion' /></a></p>
       </body>
   </html>
");
```

Add the attachment.

```
mail.addParam(file="c:\Documents\Q1FinancialResults.pdf",
type="application/pdf", attachment="attachment");
```

The last step is to send the email by using the send method of the `Mail` object.

```
email.send();
```

If you want to take a look at the wrapper, you will find it at the following location: `[CF root]/cfusion/CustomTags/com/adobe/coldfusion/mail.cfc`.

View Undelivered Mail

ColdFusion saves a copy of all undelivered emails in a special folder (`[CF root]/Mail/undelivr/`). The ColdFusion Administrator has an interface to this folder for easier browsing. To view the undelivered emails, open the ColdFusion Administrator. In the left menu, below "Server Settings", you will find "Mail". In the "Mail Spool Settings" section, click the "View Undelivered Mail" button.

This page displays a list of all undelivered email. You can delete these emails through the "delete" button, or resend them using the "respool" button.

Overwrite Default Mail Server Settings

Sometimes you may want to use a different mail server than the one set up in the ColdFusion Administrator, or perhaps you don't have access to the ColdFusion Administrator. In these situations, you can specify the mail server settings directly in the `cfmail` tag through the `server` attribute, `port` attribute, `username` attribute, and `password` attribute:

```
<cfmail from="from@domain.com" to="to@domain.com" subject="An email sent
with an alternative mail server" server="localhost">
   Hello,
   This is an email sent with an alternative mail server!
</cfmail>
```

Hands On 22

By Simon Free

In this hands on example, you are going to add email functionality to the contact form.

Tags Used: <cfmail>

1. Open up the `/www/contact.cfm` file in your code editor.
2. First we need to create a `<cfmail>` tag. Locate the line of code which reads 'Form submitted successfully!' on or around line 59.
3. Before the close tag but still inside the `<cfelse>` tag, create a `<cfmail>` tag with the following attributes:
   - **from**: #form.email#
   - **to**: me@domain.com
   - **subject**: Contact Request
   - **type**: HTML
4. Once the tag is created, we need to create our email body. After the `<cfmail>` tag, write the following HTML code:

```
<h2>Contact Request</h2>
<p>
    From: #form.contactName# (#form.email#)
</p>
<p>
    #form.message#
</p>
```

5. After the closing `</p>` tag, create a closing `</cfmail>` tag.
6. Once the `<cfmail>` is created, set up the mail server in the ColdFusion Administrator. In your browser, navigate to your ColdFusion Administrator and login. The URL will most likely be http://localhost:8500/CFIDE/administrator/.
7. Click on the 'Mail' link on the left hand side under 'Server Settings'.
8. In the 'Mail Server' box, enter 'Localhost' and click 'Submit Changes'. All mail will now go through Localhost, which will allow for viewing in the 'undeliverable' folder.
9. Open up the `/www/contact.cfm` page in your browser.

10. Test the contact form and make sure the email is being sent. Fill out all the form fields and click 'Submit'.
11. Go back to the Mail page in the ColdFusion Administrator.
12. Click on the 'View Undeliverable Mail' button, which is under the Mail Spool Settings Heading.
13. The first email in the list should be the email that was just generated by the Contact form. Click on it, if not already highlighted, and review the email body below the list. You have now successfully sent an email. When you have access to a mail server, you can update the mail settings in the ColdFusion Administrator and it will be sent out to the intended parties.

10

Document Handling

By Tim Cunningham

About Tim Cunningham

Tim Cunningham has been obsessed with programming since his dad bought a Kaypro IV to help run the family business in 1984. He started developing web applications in 1995 and developing ColdFusion applications since 1999. He serves as Vice-President of IDMI (Information Distribution and Marketing Incorporated) a group of ColdFusion centric companies aimed at the personal property insurance industry, credit card processing and print management. As Vice-President he strives to keep a forward vision toward new technology to improve customer service, improve employee skill sets and increase company profitability. Tim is proud to be an Adobe Community Professional and a member of the CodebassRadio.net team. He blogs at http://cfmumbojumbo.com and hosts community interviews on http://bolttalks.com

cfhttp

cfhttp makes HTTP calls from your ColdFusion server to an internet address of your choice. It is important to remember that it is the ColdFusion server that will be calling the URL, not the browser that is calling your ColdFusion page. Think of cfhttp as if you have proxy browser on your server that can send and receive information to any address on the internet. Imagine that this "virtual browser" on the server can save the information that it receives to a variable, so that it can be manipulated or passed to the user who has called your ColdFusion page.

Making a HTTP call

There are many attributes that the cfhttp tag can take. The simplest cfhttp call can be done like this:

```
<cfhttp url="http://cfmumbojumbo.com" method="get" />
```

This is equivalent to you getting on the server, starting the browser of your choice and putting http://cfmumbojumbo.com in the address bar. All the information returned from that HTTP request has been put into a variable called cfhttp. If you dump the cfhttp variable like this:

```
<cfhttp url="http://cfmumbojumbo.com" method="get" />
<cfdump var="#cfhttp#" />
```

You can see all the information about the raw html return (fileContent) as well as the mime type, the header information, and status codes. The response body, called filecontent by ColdFusion, contains (in this case) all the raw HTML from the page. If you had pointed the URL to a JPG or image file, it would return a java byte array.

Commonly used attributes

As of CF 10, there are 30 attributes you can use with cfhttp. For this chapter, we will cover the most commonly used ones:

Method defines the HTTP method (sometimes referred to as "verbs") of action to be performed on the URL. The methods that cfhttp understands are GET, POST, HEAD, PUT, DELETE, and OPTIONS. The most commonly used verbs are GET and POST. GET retrieves whatever information returns from the URL. POST can submit data to the chosen URL (form fields, uploaded files, etc.) and then return the resultant page. OPTIONS will return with information explaining what verbs the web server will accept from you. HEAD is another verb; it responds the same as a GET, but only returns the meta-information that is in the returning page's header. It will not return the response body (HTML, image, file body, etc.). PUT can upload file on the remote server. DELETE can remove a file on the requested URL. The web server on the URL must be set up for PUT and DELETE.

URL is the address to which you want to talk. It can be an IP address, hostname, or fully qualified domain name. If you pass URL parameters in the url, they will be included in your request. For example, url="http://www.myserver.com?id=5&myname=Tim" will be passed to the remote

server just as if it were typed into a web browser. You can include `http://` or `https://`; if you do not, `cfhttp` will default to `http://`. If you include a port number in the URL, it will override the port attribute. For example, `url="http://www.myserver.com:8088"` will connect on port 8088. Remember, the ColdFusion server is going to need permissions to talk to the remote server on port 8088, otherwise you may get unexpected results.

File and Path is used if you want the response body to be written to a file; use this attribute and give it a file name and path:

```
<cfhttp url="cfmumbojumbo.com" path="C:/test" file="test.html" />

<!--- Will take the HTML from the web page and write it to C:/test/test.
html on your ColdFusion server. --->
```

GetAsBinary tries to convert the response to binary. This can be set to Auto, Never, No, or Yes. Auto will convert the response to binary if it is recognized as proper form binary data; otherwise it will converted to text. Never will treat all returned data as test, regardless of its true MIME type. `GetAsBinary` is useful when you are using `cfhttp` to grab images, zip files, or other binary data off other web sites.

```
<cfhttp url="http://6e33d2c506c5fcafb083-2091e9475e9ec26fdd926321647b4
6b0.r56.cf1.rackcdn.com/tim-bennadel.png" />
```

UserName and password is used when the web page is protected by basic authentication on the web server. Note: these attributes are not useful for a typical web site designed login screen.

Result is the name of the variable you want to hold returned results.

- **Port** is the port number you wish to make the HTTP request through. The server that your ColdFusion code is running on must have permissions to communicate on this port. The default is port 80.
- **Timeout** is the value in seconds that the ColdFusion server will wait for a successful response before it considers the request to have failed.

cfdocument

`cfdocument` will take your combination of CFML and HTML and convert it to a PDF. At its simplest, you can stick some text between the opening and closing tags of `cfdocument` (there is currently no built-in `cfdocument` script equivalent) and it will render a PDF to the screen. Here is some sample code:

Creating a Simple PDF

```
<cfdocument format="PDF">
   <cfoutput>
   Bacon ipsum dolor sit amet sirloin fatback #dateformat(now(),
"short")#
   </cfoutput>
</cfdocument>
```

What HTML is Supported?

It would be optimal if anything you produce in a browser will look exactly the same in the `cfdocument` generated PDF. However, `cfdocument` currently only supports HTML 4.01, XML 1.0, DOM Level 1 and 2, and CSS1 and CSS2. To be fair, the current support level will cover 90% of most HTML generation you attempt to do. There are 76 supported CSS styles; see http://help.adobe.com/en_US/ColdFusion/10.0/CFMLRef/WSc3ff6d0ea77859461172e0811cbec22c24-7c21.html for the entire list. Also note that if you have a Word file that you save HTML, the resulting PDF will not look anything like the Word document.

Setting Page Attributes

`cfdocument` has attributes that allow you to set margins, page size, page orientation, and passwords.

```
<!--- These attributes will create a password protected legal sized PDF
with the following characteristics
Bottom Margin: 2.0 inches
Top Margin: 1 inch
Left Margin: 1/2 inch
Right Margin: 1 1/2 inch
User Password: secret123
Encrypted: 128-bit encryption
```

```
-->
<cfdocument format="PDF" pagetype="legal" marginbottom="2.0"
margintop="1.0" marginleft="0.5" marginright="1.5"
userpassword="secret123" encryption="128-bit">
   <cfoutput>
   Bacon ipsum dolor sit amet sirloin fatback #dateformat(now(),
"short")#
   </cfoutput>
</cfdocument>
```

Headers, Footers & Page Breaks using cfdocumentitem

`cfdocumentitem` can be used to create PDF headers, footers, and page breaks. If you are not using `cfdocumentsection` (covered next), then where you place the `cfdocumentitem` in your HTML will make a difference as to how it affects the entire document.

```
<cfdocument format="PDF">
    <cfoutput>
         <cfdocumentitem type="header">
             <h1 style="text-align:center;">BIG FANCY HEADER</h1>
         </cfdocumentitem>
         Bacon ipsum dolor sit amet sirloin fatback #dateformat(now(),
"short")#
         <cfdocumentitem type="pagebreak"/>
         Bacon ipsum dolor sit amet sirloin fatback #dateformat(now(),
"short")#
         <cfdocumentitem type="footer">
             <h1 style="text-align:center;">Page #cfdocument.
currentPageNumber# of #cfdocument.totalPageCount#</h1>
         </cfdocumentitem>
    </cfoutput>
</cfdocument>
```

cfdocumentsection

Sometimes you may want to create a PDF that does not have the same header and footer for every single page (like a title page), or you may have few pages that need different margins. To deal with

this, you can use `cfdocumentsection`, which puts your HTML content into separate blocks, each of which can have their own settings for margins, headers, and footers defined in a `cfdocumentitem` nested in that `cfdocumentsection`.

```
<cfdocument format="PDF">
<cfoutput>
    <!--- Section 1 --->
    <cfdocumentsection name="bookmark1">
        <cfdocumentitem type="header">
        <h1 style="text-align:center;">BIG FANCY HEADER</h1>
        </cfdocumentitem>
        Bacon ipsum dolor sit amet sirloin fatback #dateformat(now(),
"short")#
        <cfdocumentitem type="footer">
        <h1 style="text-align:center;">Page #cfdocument.
currentPageNumber# of #cfdocument.totalPageCount#</h1>
        </cfdocumentitem>
    </cfdocumentsection>


    <!--- Section 2 --->
    <cfdocumentsection name="bookmark2">
        <cfdocumentitem type="header">
        <h1 style="text-align:center;">2nd page header</h1>
        </cfdocumentitem>
        Bacon ipsum dolor sit amet sirloin fatback #dateformat(now(),
"short")#
        <cfdocumentitem type="footer">
        <h1 style="text-align:center;">Page #cfdocument.
currentPageNumber# of #cfdocument.totalPageCount#</h1>
        </cfdocumentitem>
    </cfdocumentsection>
</cfoutput>
</cfdocument>
```

Tips for Working with cfdocument

If you find that your application must use a lot of `cfdocument`, here are some tips for improving performance and rendering.

Avoid using relative file paths:

```
<!--- Don't use these below --->
<link rel="stylesheet" type="text/css" src="../assets/mainStyle.css">
<img src="../images/mypic.png">


<!--- INSTEAD use expand path to provide the full path to the file --->
<link rel="stylesheet" type="text/css" src=
"#expandPath('../assets/')#mainStyle.css">
<img src="#expandPath('../images/')#mypic.png">
```

Set the `localURL` attribute to yes. When you set this attribute of `cfdocument` to true, it tells ColdFusion to retrieve image files directly from the server rather by attempting to use HTTP.

cfpdf

Whereas `cfdocument` is used to create PDFs, the `cfpdf` tag is used to manipulate existing PDFs. With `cfpdf`, you can read an existing PDF, write meta-data to it, merge PDFs together, delete pages, create thumbnails of the pages, extract text & images, add or remove watermarks, manipulate headers & footers, create PDF portfolios, and deal with PDF passwords, permissions and Encryption.

Reading a PDF

The `cfpdf` tag accepts an action property, which currently has 18 possible values. Choosing the 'read' action will take an existing PDF, read it to memory, and store it in a variable name of your choosing.

```
<cfpdf action="read" name="myDoc" source="C:\docs\mypdf.pdf" />
<cfdump var="#myDoc#" />
```

The above code will dump out the metadata for the chosen PDF such as the author, date created, keywords, etc. This is the same information you would receive if you used the `action="getinfo"` argument; however, with the getInfo action, you can access the values of resulting operation. If you wanted to display the PDF to the browser, the following code could be used:

```
<cfpdf action="read" name="myDoc" source="C:\docs\mypdf.pdf" />
<cfcontent variable="#toBinary(myDoc)#" type="application/pdf" />
```

Creating Thumbnails of PDF Pages

```
<cfpdf action="read" name="myDoc" source="C:\docs\mypdf.pdf" />
<cfpdf action="thumbnail" source="myDoc" destination="C:\docs\"
overwrite="yes" />
<cfimage action="read" name="img" source="C:\docs\thumbnail_page_1.jpg"
format="jpg" />
<cfcontent reset="true" variable="#imageGetBlob(img)#" type="image/jpeg"
/>
```

The above code will read a PDF and create 25% scale JPG thumbnails for all the pages in the `C:\Docs` folder. The default naming convention in CF 10 is `thumbnail_page_1`, `thumbnail_page_2`, and so on. Format can be set to JPG, TIFF, or PNG. The thumbnail action also has other arguments that work along with it to determine the scale, max breadth, resolution, and naming scheme for the generated thumbnail.

Extracting images

Most images embedded in a PDF can be extracted and saved to a folder of your choice using a file prefix of your choice. By default, the file prefix is `"cfimage-"` and the image number. The default destination is in the same folder as the ColdFusion page calling the `cfpdf` tag.

```
<cfpdf action="extractimage" source="./mypdf.pdf"  overwrite="yes" />
<img src="cfimage-0.jpg" />
<br />
<img src="cfImage-1.jpg" />
```

Extracting Text

If you desire to extract the text of a PDF, such as if you wanted to search and catalog words used in the PDF, you can do so with the `extractText` action. The `extractText` action will return a XML document; each page of text is in its own XML node name with a corresponding page number. You can then use ColdFusion's powerful XML parsing tags to interact with the XML document. The code below will output the XML from a PDF to the browser for you.

```
<cfpdf action="extracttext" source="./mypdf.pdf" name="myXML" />
<cfcontent type="text/xml" />
<cfoutput>#myXML#</cfoutput>
```

Merging PDF Documents

Merging using a List

There are a variety of ways ColdFusion can merge different PDF documents together. The simplest is to pass a comma delimited list of PDF files, which will append them in the order you list them.

```
<cfpdf action="merge" source="mypdf.pdf,beer.pdf"
destination="mergedPDF.pdf" overwrite="yes" />
<cfpdf action="read" name="myPDF" source="mergedPDF.pdf" />
<cfcontent variable="#toBinary(myPDF)#" type="application/pdf" />
```

Merging using cfpdfparam

You can also use `cfpdfparam` tags nested within an opening and closing `cfpdf` tag to more accurtely control the final PDF. The `cfpdfparam` tag accepts `source`, `pages`, and `password` arguments. The pages attribute can choose what page(s) you want to merge into the final document.

You can choose one page or a list of page numbers. Password is used for password protected PDFs. The code below will combine two PDFs: `mypdf.pdf` and `beer.pdf`. However, rather than appending it, we will take the first page of `mypdf.pdf`, then all of `beer.pdf`, and finally the second page of `mypdf.pdf`.

```
<cfpdf action="merge"  destination="mergedPDF.pdf" overwrite="yes">
    <cfpdfparam source="myPDF.pdf" pages="1" password="test" />
    <cfpdfparam source="beer.pdf" />
    <cfpdfparam source="myPDF.pdf" pages="2" password="test" />
</cfpdf>
<cfpdf action="read" name="myPDF" source="mergedPDF.pdf" />
<cfcontent variable="#toBinary(myPDF)#" type="application/pdf" />
```

Merging a Directory of PDFs

If you use the directory argument on a `cfpdf` merge action, you can specify a folder that contains multiple PDFs and merge them into one PDF. The order strategy is either by name (ascending alphabetical) or time (ascending by file time stamp). Each file in the directory is evaluated if it is a valid PDF readable by ColdFusion; if you have `stoponerror="yes"`, then the `cfpdf` tag will error if any of the files in the directory are not valid PDFs. If you have `stoponerror="no"`, then any non-PDF files will be skipped.

```
<cfpdf action="merge"  directory="./" orderascending="yes"
    stoponerror="false" overwrite="yes" destination="mergedPDf.pdf" />
<cfpdf action="read" name="myPDF" source="mergedPDF.pdf" />
<cfcontent variable="#toBinary(myPDF)#" type="application/pdf" />
```

Deleting Pages

You can remove a single page or a range of pages using the `action="delete"` on the `cfpdf` tag. Specifying a destination is optional for delete actions. If you do not specify a destination, the page or pages will be deleted from the original source PDF.

```
<cfpdf action="deletepages" pages="2-3" source="mergedPDF.pdf" />
<cfpdf action="read" name="myPDF" source="mergedPDF.pdf" />
<cfcontent variable="#toBinary(myPDF)#" type="application/pdf" />
```

Creating and Removing Watermarks

To add a watermark to a PDF, you can either use an existing watermark from an image or another PDF. If you specify a destination, a new PDF will be created with the watermark; otherwise the watermark will be added to the source PDF. By default, the watermark is placed in the center of the page.

```
<cfpdf action="addwatermark" source="mypdf.pdf"  image="draft.png"
    foreground="yes" overwrite="yes" />
<cfcontent variable="#toBinary(myPDF)#" type="application/pdf" />
```

To modify the positioning of the watermark, you can use the `rotation` and `position` arguments. Rotation is the number of degrees the watermark image will be turned. The `position` argument takes Cartesian coordinates, with the top-left corner being the 0,0 position. The code below will put a watermark at a 45 degree angle in the top left corner of the PDF:

```
<cfpdf action="addwatermark" source="mypdf.pdf"  image=
"draft.png" foreground="yes" overwrite="yes" name="mypdf" rotation="45"
position="0,700" opacity="2">
<cfcontent variable="#toBinary(myPDF)#" type="application/pdf" >
```

Removing a watermark is very simple:

```
<cfpdf action="removewatermark" source="mypdf.pdf" />
<cfpdf action="read" name="mypdf" source="mypdf.pdf" />
<cfcontent variable="#toBinary(myPDF)#" type="application/pdf" />
```

Optimizing PDFs

PDFs can accumulate a lot of baggage. To help slim down the file size and speed up the rendering of a PDF, you can use the optimize action on the `cfpdf` tag. The result can be saved to a new file, the original source, or a variable. You must also supply the algorithm used to compress the PDF; your choices are `Nearest_Neighbour` (notice the non-American spelling), `bicubic`, `bilinear`.

- **Nearest_Neighbour (default)**: Lossy image quality, fast processing.
- **bicubic**: High quality image, slowest processing.
- **bilinea**: Mid-range image quality, mid-range processing speed.

The above guidelines are very general as both the resultant compression is highly dependent on the actual content of your PDF. In the example PDF built for this chapter, its original PDF was 221 KB. `Nearest_Neighbour` took it down to 49 KB, `bicubic` to 47 KB, and `bilinear` to 48 KB.

```
<cfpdf action="optimize" source="mypdf.pdf"  overwrite="yes"
algo="bilinear" />

<cfpdf action="read" name="mypdf" source="mypdf.pdf" />

<cfcontent variable="#toBinary(myPDF)#" type="application/pdf" />
```

There are also 12 other arguments which can be passed to `cfpdf` to remove other PDF features, such as attachments, bookmarks, font styling, javascript, etc.

Headers and Footers

`cfpdf` also has actions called `addHeader` and `addFooter` which will add a header and a footer to an existing PDF. The header or footer can be simple text or an image.

```
<cfpdf action="addheader" source="mypdf.pdf" name="mypdf" text=
"Header Here" />

<cfcontent variable="#toBinary(myPDF)#" type="application/pdf" />
```

Using the text argument, you have access to four dynamic options: `_LASTPAGELEBEL`, `_LASTPAGENUMBER`, `_PAGELABEL`, and `_PAGENUMBER`. For example, if you wanted your footer to have 'Page # of #' on each page, you could use the following code:

```
<cfpdf action="addFooter" source="mypdf.pdf" name="mypdf" text=
"Page _PAGENUMBER of _LASTPAGENUMBER" />

<cfcontent variable="#toBinary(myPDF)#" type="application/pdf" />
```

cfpdfform

`cfpdfform`, is one of the most powerful yet overlooked features of ColdFusion. Up until now you have been working with "flat" PDF files. `cfpdfform` is useful when you have to generate forms that combine boilerplate text and dynamic text; it can also be used to collect data. There are two options in creating the initial PDF form: Adobe Acrobat and Adobe LiveCycle Designer. For the purpose of Learn CF in a Week, we will only be covering PDF's created using Adobe Acrobat.

Read a PDF Form

The source of the PDF can be a path to a PDF file or a variable that holds the byte array of the PDF. You can store the results of this read in a variable.

```
<cfpdfform action="read" source= "testForm.pdf" result="formData"/>
<cfdump var="#formData#" label="formData" />
```

The above code will dump all the form names and the data they hold.

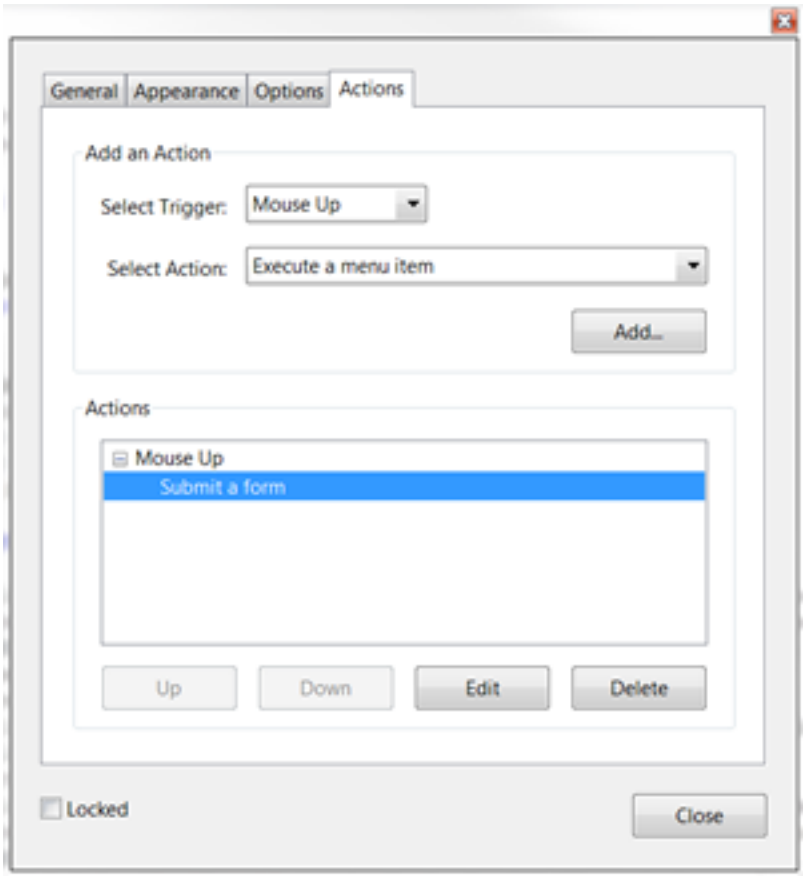
Populating a PDF Form

Once you know the name of the fields in your PDF, you can use `cfpdfformparam` inside the `cfpdfform` tag to populate the fields. The below code loads all the field names from `testForm.pdf` and loops through them, populating them with "XX".

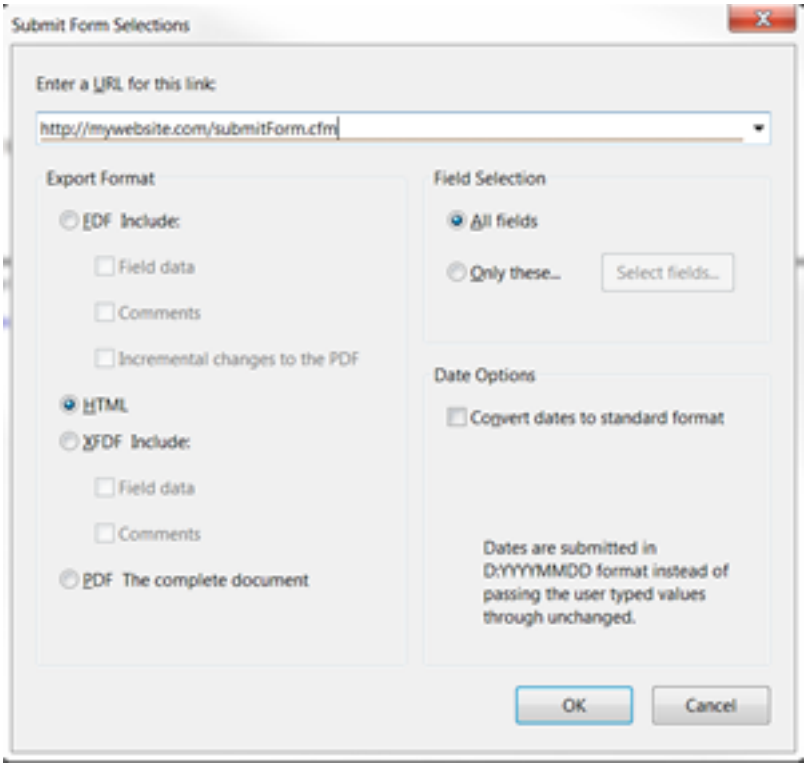
```
<cfpdfform action="read" source= "testForm.pdf" result="formData"/>
<cfpdfform action="populate" source="testForm.pdf" destination=
"testPDF.pdf" overwrite="yes" >


<!--- Loop through all the Form Fields and populate them with xx --->
<cfloop collection="#formData#" item="item">
    <cfpdfformparam name="#item#" value="XX" />
</cfloop>
</cfpdfform>
<cfpdf action="read" name="pdfData" source="testPDF.pdf" />
```

Submitting PDF Forms

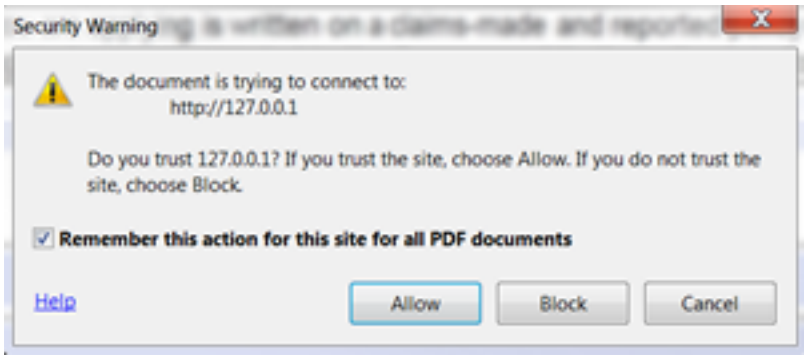
In Adobe Acrobat, you can add a button to a form. The properties of the button have some action options. Actions are based on triggers to the button, like the "Mouse Up" action, for instance. One of the actions you can choose is Submit a form.



The Submit Form action has some options you can set as to what type of data will be submitted when the button is clicked; below, we have chosen HTML. This option will send all field names that contain data and other information the user has filled out to the chosen URL, just as an HTML form would. The PDF option will send the entire PDF along with the information the user has filled out.



When users click the PDF submit button, they will be informed that the document is trying to connect to a website and give them the option to trust the site. This is for security purposes; users should be informed prior that they will receive this warning and not to panic.



If the target page displays any output, it will be presented to the user in another PDF. When you have the data, you may access it in the form scope.

File Manipulation

A common request when building a website is the ability to upload and manipulate files. ColdFusion offers a number of tags and functionality that allow you to control the files on the file system . The functionality offered ranges from the more simple process of creating a new file all the way up to manipulating a files meta data.

Let's take a look at the functionality in the order that you will probably use it.

Uploading Files

More often than not, the file that you wish to manipulate belongs to the users. For you to get the file onto the server they must first upload it via a form. The upload form used has no special attributes that are specific to ColdFusion; it just points to a ColdFusion page on the server.

To upload the file, you will need to use the `cffile` tag. The `cffile` tag offers a number of different actions that can be performed; most of these actions also have an equivalent function that can be used if you prefer `cfscript` format. However, the first action we are going to look at, `upload`, does not currently have a `cfscript` equivalent. For the sake of simplicity, this chapter will focus mostly on the functions, rather than the tags, as they are less verbose and can be used in tags or in `cfscript` calls.

When using `cffile` to upload a file, you will give it the action of upload by telling the tag to upload the file, then provide a destination. The destination is where you want the file to be uploaded to. You will provide the location of the folder on the server. Next, you will use the `filefield` attribute to specify the name of the form field that has the file in it. This will be the value of the `name` attribute in the input you created in the form that is of type file.

With these three attributes set, you can run the code and be able to upload a file. There are, however, some additional attributes that will be useful, and, in most cases, should be used. The first attribute to consider is the `nameconflict` attribute. When uploading a file, if there is already a file in the destination folder with the same name, an error will be thrown. Sometimes this might be a desirable outcome, but most of the time you will still want the file to upload. The `nameconflict` attribute accepts the following values:

- **error**: This is the default action and will throw an error stating that there is already a file with that name.
- **makeunique**: The most common action used and will append a number to the file name.
- **overwrite**: This action will overwrite the file on the server with the newer file.
- **skip**: This action will not put the file on the server, but will also not throw an error.

Another useful attribute of the `cffile` tag is the `accept` attribute. The `accept` attribute takes in a list of mime types that it will upload. For example, one mime type that could be placed in the accepts attribute could be 'image/jpeg'. If a file was uploaded that was not a JPEG image, such as a PDF, an error will be thrown. This attribute allows you to limit what gets put onto your server. This is very important as uploading files opens up a vulnerability to your server that some people could try to exploit. Only allowing specific file types will limit someone's ability to get an executable on your

server.

The final attribute we are going to review is the `result` attribute. When a file is uploaded, a struct of data is created automatically. Inside that struct is a lot of useful information about the file, such as its file name, when it was uploaded, its file name on the server, the location of the file, etc. If you do not specify a value for the result attribute, then this struct will be stored in a variable called `cffile`. If a value is specified in the result attribute, then this value will be the name the struct is stored as.

If you were to use all the attributes mentioned above, your `cffile` tag might look something like this:

```
<cffile
    action="upload"
    destination="/Users/simon/Sites/aSite/files/"
    filefield="aFile"
    nameconflict="makeunique"
    accept="image/png"
    result="fileUploadResult" />
```

This would upload the file in the form called `aFile` to the `/Users/simon/Sites/aSites/files/` folder only if it is a PNG image. If there was already a file with that name, it would make the filename unique and store its result into a variable called `fileUploadResult`. When dumped out, the `fileUploadResult` variable would looks something like this:

struct	
ATTEMPTEDSERVERFILE	export_pdf.png
CLIENTDIRECTORY	[empty string]
CLIENTFILE	export_pdf.png
CLIENTFILEEXT	png
CLIENTFILENAME	export_pdf
CONTENTSUBTYPE	png
CONTENTTYPE	image
DATELASTACCESSED	{d '2012-11-03'}
FILEEXISTED	YES
FILESIZE	4150
FILEWASAPPENDED	NO
FILEWASOVERWRITTEN	NO
FILEWASRENAMED	YES
FILEWASSAVED	YES
OLDFILESIZE	4150
SERVERDIRECTORY	/Users/simon/Sites/aSite/files
SERVERFILE	export_pdf1.png
SERVERFILEEXT	png
SERVERFILENAME	export_pdf1
TIMECREATED	{ts '2012-11-03 13:24:46'}
TIMELASTMODIFIED	{ts '2012-11-03 13:24:46'}

When uploading files, it is important to consider the security implications. If you allow users to upload any files and place those files within the web root, it is possible for the user to upload something malicious and run it. It is essential to take all available steps to stop this from being possible. In addition to the mime type check that the `cffile` tag can perform, you can also choose not to upload the file into the web root, but upload it to a location that is outside the web root and not accessible via the web. Once in this location, you can do some additional checks to confirm that the file is safe, and if you deem it to be safe, it can then be moved to its final destination.

ColdFusion makes this process easier by providing a temporary directory that can not be accessed via the web and is periodically purged of old files. Calling the function `getTempDirectory` will return the location of the temporary directory. You can then upload your file to this location, then use the result struct to get the file location and filename.

Another function which can be very useful when dealing with files and folders is the `expandpath` function. In the example above, we provided a hardcoded path to the folder we wished to upload it to. This path is accurate, but only accurate on the machine we are currently developing on. If this code were pushed to a production server, the chances are that the file path will not stay the same. The `exapndPath` function accepts a path from the current file location and generates the absolute path for you. For example, if the code above were placed in the root folder, we could have used `#expandPath('files/')#` rather than typing out the exact path ourselves. Also, if the file that contained that line of code were in a subfolder, we could have used `expandPath` as so: `#expandPath('../files/')#`.

Manipulating Files

Once the file needed is accessible, either by uploading or it being pre-existing, we can then do some manipulation to it. If the file that you are manipulating is an image, not all of these functions will apply to you. ColdFusion offers a number of image specific functions which will be covered in the next section.

When dealing with file functions in ColdFusion, it requires either a file object or an absolute path. When you plan on doing multiple updates to a file, it is much easier to create a file object that can be passed to the different functions. If you do not use a file object, then the file will need to be read and written multiple times; as IO speeds can be slow sometimes, using the file object will increase the processing of the code. To create a file object, use the `fileOpen` function, which accepts the absolute path of the file. It has an optional argument called `mode`. This mode is what action you plan on performing on the file, such as read, write, or append. If you do not specify the mode, ColdFusion defaults to read. If you are in read mode, you will not be able to write any data to the file using this file object. Once you have the file object, you will have access to a struct of data pertaining to the file, such as file name, lastmodified and so on.

The `fileRead` function is how you read in data to a variable. The function accepts either an absolute file path or a file object. The contents of the file will then be returned and can be saved in a variable It is important to note that for large files, this process could take a while to run, and as the variable is in the memory, it is possible that you will see issues with your server if it contains large amounts of data in memory.

If you plan on reading and manipulating a large file, you might want to consider the `fileReadLine`

function. This function will read in a line at a time; since you are only dealing with one line at a time you will not fill up the server's memory with the file contents. Also, as you are only loading up one line, your code will most likely run faster.

To read in a file one line at a time, you will need to call a function called `FileisEOF`. This function accepts the file object and will tell a loop if it has read all the lines in the file or not. Here is an example of these two functions at work:

```
while(!FileisEOF(afile)) {
     line = FileReadLine(afile);
     WriteDump(line);
}
```

It is important to remember that when calling the fileOpen function, it is like opening the file with an application. While that file is open with ColdFusion, other applications will not be able to access the file. All files that are opened need to be closed. To do that, use the `fileClose` function and pass it in the file object. If you do not close the file, the file system will see it as still in use and will continue to lock the file. If `fileClose` is not called, it is possible that the file will remain locked until the server is restarted.

The next function we will discuss is the `fileWrite` function. The `fileWrite` function allows you to write data to a file. The function accepts a file path or a file object and a string of data. When writing data to a file, if the file does not exist, then the file will be created. If the file already exists, the data will be overwritten by the new data. If you do not wish to overwrite the data, first check if the file exists. To do that, call the `fileExists` function and pass it in the path to the file. If the file exists, then it will return true, else it will return false.

If at any point you want to delete a file, call the `fileDelete` function and pass in an absolute path to the file.

cfdirectory

As well as needing to manipulate files, it is sometimes necessary to manipulate directories. Just like with files, there is one main tag that can be used for a number of functions. And, just like with files, there is one important action that does not have a script equivalent.

The tag in question is the `cfdirectory` tag. This tag has a number of actions available to it; for now, we are going to focus on the list action. The list action will return all the files and directories that are located in the provided directory. The `directory` attribute specifies which directory to list the contents of. In addition to the action and the directory attributes, a name attribute is also required; this specifies the name of the variable which will be created with the resulting data. The resulting data from this tag is a query. In this query are a number of columns of information and one row per file in the directory. The data returned in the query is:

- **name**: The file or directory name
- **directory**: The directory path
- **size**: The file size
- **type**: If it is a file or a directory
- **dateLastModified**: When the directory or file was last updated
- **attributes**: The file attributes if there are any

Sometimes you might only be concerned about a certain type of file that is in a folder. If that is the case, the filter attribute can be used to filter the data that is returned based on file extension. If you only wanted to return PNG images, you could give the filter attribute the value of '*.png' and only PNG images would be returned.

Here is an example of a `cfdirectory` call that will list out all PNG images and place them in the qImages variable:

```
<cfdirectory
    action="list"
    directory="#expandpath('images/')#"
    name="qImages"
    filter="*.png" />
```

In addition to listing out the contents of a directory, the `cfdirectory` tag also allows you to create, delete, copy, and rename directories. These actions also have equivalent functions that can be called from `cfscript`.

Image Manipulation

A great feature set of ColdFusion is its image functionality. ColdFusion has the ability to easily manipulate images, create new images, draw images, and write them back to the file system. All this functionality comes right out of the box with ColdFusion without need of any extra plugins or installs.

There are so many pieces of image related functionality that it is not feasible to explain all of them in this section. For the purpose of this section, we will review a few of the more common functions used. To see a full list of image functionality available, go to: http://help.adobe.com/en_US/ColdFusion/10.0/CFMLRef/WSc3ff6d0ea77859461172e0811cbec1a60c-7ffc.html#WSc3ff6d0ea77859461172e0811cbec22c24-66e4.

Reading in an Image

Before you start manipulating an image, the first thing is to read the image into memory by calling the `ImageRead` function. The `ImageRead` function accepts one argument as the path to the image. The image can either be a path to its location on your server, the path to an image in the virtual file system, or a URL to the image located on a different server. If you want to manipulate an image provided by the user, the image must be already uploaded to the server, which was discussed in the previous section.

The `imageRead` function returns an image object which can then be stored in a variable. This image object will be needed for the other image functions. Once you have this image object, you can start to manipulate the image.

Scaling and Cropping an Image

The most used image functions are the functions that will resize and crop images. This functionality will allow you to reduce the size of an image so that it will display better on your web site. Rather than resizing an image using the `img` tag on your web site and scaling it down, the best course of action is to resize the image programmatically.

The best function to use when resizing an image is the `ImageScaleToFit` function. This function accepts a number of arguments, the first being the image object of which the action will be performed on; the ensuing options are the desired width followed by the desired height. When run, the function will proportionally resize the image so that it is no wider than the specified width and no taller than the specified height.

As an option, the function also accepts values for interpolation and blur factor. The interpolation value controls the quality of the image created. There is a list of a number of algorithms available which can be found in the documentation. The blur factor value specifies how much blur to add to the image. The value is between one and ten, where ten is the blurriest.

Below is an example of an image object being scaled to a 150px by 200px area.

```
ImageScaleToFit(imageObj,150,200);
```

If, when scaling an image, your concern is fitting the image into one of the proportions, such as width, then you can opt to pass in an empty string into the other value, as ColdFusion will only scale the image for the specified value. For example, if you only wanted to scale the image to fit into a 100px wide area, you could use the following code:

```
ImageScaleToFit(imageObj,100,'');
```

In some situations, it might be necessary to crop an image so that it fits the desired output area. To do this, you will use the `imageCrop` function. The `imageCrop` function accepts a number of arguments, all which are required. Along with the `image` object, you will need to pass in the `height` and `width` of the cropped image you are generating as well as the `x` and `y` coordinates of where the cropping should begin. For example, if you ran the code sample below, you would generate an image that is 200px high and 150px wide. It was cropped from the image contained in `imageObj`, and the cropping started 10px from the top and 20px from the left.

```
ImageCrop(imageObj,20,10,150,200);
```

Getting Image Information

At some point in your image manipulation process, you may need to gather certain information about the image, such as `width` and `height`. Perhaps you will be using it for logic decisions, or possibly you wish to store it in the database along with a record of the image. Whatever the reason, ColdFusion makes it very simple to do.

The `ImageInfo` function accepts an image object and returns a struct containing information about the image. In addition to outputting the usually expected information like width and height, it also provides additional information about the image, such as transparency and color space. Here is an example of the information returned when doing `ImageInfo` on the Google logo:

struct	
colormodel	struct
	alpha_channel_support: NO
	alpha_premultiplied: NO
	bits_component_1: 8
	bits_component_2: 8
	bits_component_3: 8
	colormodel_type: IndexColorModel
	colorspace: Any of the family of RGB color spaces
	num_color_components: 3
	num_components: 3
	pixel_size: 8
	transparency: OPAQUE
height	95
source	https://www.google.com/images/srpr/logo3w.png
width	275

Optionally, if you just care about the image dimensions, you can call the `ImageGetWidth` and `ImageGetHeight` functions, passing in your image object, to get the width and height values of your image.

Creating Image

In the above examples, we have assumed that the image being manipulated is an image that already exists. However, with ColdFusion you can create a new image and manipulate this new image.

To create a new image, you would call the `ImageNew` function. This function returns an image object. `ImageNew` has no required arguments; if no arguments are passed, a blank image object will be created and additional functions will need to be called to set its width, height, etc. `ImageNew` does have a number of optional arguments which are explained below.

- **source**: The source argument is another image. It can be another image object, a file path, or a URL. If the source argument is used, then the width and height will automatically be set to the dimensions of this image. It cannot be overridden in this function call, but can be changed later. Using the source argument is similar to copying the image, as this will make an entirely new image.
- **width**: Sets the width of the image, if the source argument has not been provided.
- **height**: Sets the height of the image, if the source argument has not been provided.
- **imageType**: This argument allows you to specify the type of image you are creating. You are able to create RBG, ARBG, and grayscale images.
- **canvasColor**: The background color for the image. This argument accepts a hexidecimal value (such as FFFFFF), a string value (such as Blue), or a list of 3 RGB numbers.

Drawing on an Image

When you have your image object, either via loading in an image or creating a new image, you have the ability to draw on the image. Using the drawing functionality is slightly different than the previous functions you have encountered. Most of the other functions you have reviewed so far have always encapsulated all their data. Other settings that you can control do not affect the output of those functions; when it comes to image functions, that is not the case.

When thinking of the image functions, think of them as if you were using your favorite design software. When you are drawing a line on the page, you would first select the color, then the line thickness, then you would draw the line from point A to point B. That is how it is when using the image functions. Rather than calling a function to draw a line and passing in the line color, the line thickness, and the x and y coordinates of point A and point B, you will set the basic information first, then call the functionality to draw the line.

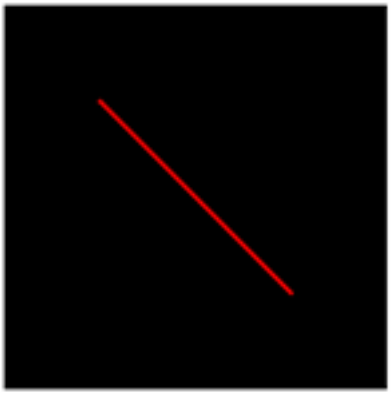
For example, say you wanted to draw a red line that is 2px thick. The first thing to do is call the `ImageSetDrawingColor` function, and pass in the image object and the color red. Then you would call the `ImageSetDrawingStroke` function and pass in the `image` object and a struct containing our stroke properties. Once that is done, then call the `ImageDrawLine` function and pass it in the image object, the `x` and `y` coordinates of our start point, and the `x` and `y` coordinates of our end point, which will then draw the line.

The advantage of calling `ImageSetDrawingColor` and `ImageSetDrawingStrike` is that once we draw the first line, we can draw multiple ones and they will all have the same color and stroke styles. If we ever need to change the color, we simply call the `ImageSetDrawingColor` function with the new color, then draw the next line.

Here is the code for the example discussed above:

```
imageObj = ImageNew('','200','200','rgb');
ImageSetDrawingColor(imageObj,'Red');
ImageSetDrawingStroke(imageObj,{width=2});
ImageDrawLine(imageObj,50,50,150,150);
```

And here is the image generated:



Writing Images

Once you have finished manipulating your image, the next thing is to write the image. To write an image to the file system, you will use the `ImageWrite` function. If the image object you are trying to write was created by reading in an image from the file system, you can call the `ImageWrite` function and only pass in the image object; it will overwrite the old image with this new one. If you created the image by calling `ImageNew` or if you do not wish to overwrite the original image, you can also pass in a `destination`, which is an absolute file path, and a file name of where to save the image. Additionally, `ImageWrite` has two option arguments that can be passed. The first is if the image should overwrite another image if it already exists; the second is the `quality` that should be used when encoding the image. The available quality values range from 0 to 1 and only apply to JPG and JPEG images.

Outputting Images

During the development process, it can be cumbersome to constantly be writing files to the file system, opening them up in an image editor, and checking if they are correct or not. To speed up the process for development, you can use the `cfimage` tag and provide an action of `writetobrowser`. In addition to the action, you provide a `source` attribute and pass in the image object. The `cfimage` tag will then output the image to the browser for you rather than having to constantly write it to the file

system. Here is an example:

```
<cfimage action="writeToBrowser" source="#imageObj#" />
```

Spreadsheets

When creating a website that generates reports, having the ability to generate Excel spreadsheets is a great feature to offer users. Allowing people to provide an Excel file of data rather than forcing them to manually enter everything into your system is also a great feature for users. ColdFusion allows for both the creation and reading of spreadsheets.

ColdFusion offers the ability to use tags and functions to manipulate spreadsheets. The `cfspreadsheet` tag is best used when reading in a spreadsheet; the spreadsheet functions are best used when manipulating the spreadsheet. There is the `SpreadsheetRead` function available to you, but it does not return the data contained in the spreadsheet. For this section, we will use the `cfspreadsheet` tag to read in data, but will use the spreadsheet functions for creation and manipulation.

Reading a Spreadsheet

Reading in a spreadsheet is very simple. The first thing is make sure the file is on your server. If this is a user supplied spreadsheet, this can be done by using the file upload techniques discussed in the previous section.

To read in a spreadsheet and assign it to a variable, use the `cfspreadsheet` tag. The `action` to use with the tag is 'Read'. You then must specify the location of the XLS or XLSX file by using the `src` attribute; then, specify a value to store the result in via the query attribute. With those three attributes set, you are able to read in the spreadsheet and access it from the variable name specified in the query attribute. The variable that you will have right now is a query object. If you did not want the data from the spreadsheet, but wanted to manipulate the spreadsheet, you could supply the tag with a `name` attribute. A variable will be created with the name you provided in the `name` attribute, which will contain the spreadsheet object that the spreadsheet functions require. For now, let's focus on the query.

Even though the `cfspreadsheet` tag will run with those three required attributes, the data that you have will probably not be in the best format for use. A prime example of this is that most spreadsheets have a header row that says what is in the column. Without specifying that the file has a header column and that the header column should be used for naming the query columns, you will end up with a query that has the headers as a row and the query columns named COL_1, COL_2 and so on. Specifying a `headerrow` attribute and the row number that contains the header information in your file will allow the `cfspreadsheet` tag to name the columns in the query according to the header values of your document. Also, specifying the attribute `excludeheaderrow` and setting it to true will stop your header row being included in the query data.

Once the `cfspreadsheet` tag has run and you have the data of the spreadsheet in your query variable, you can then add additional logic to add the data to your website.

Creating a Spreadsheet

Creating a spreadsheet is a simple process. The first thing that needs to be done is to create a spreadsheet object. Just like with the file and image functionality, an object rather than a file path is needed when calling the functionality. To create a spreadsheet object, call the `SpreadsheetNew`

function. Passing in no arguments will create a simple spreadsheet object; optionally, you can pass in a sheet name which will be the name of the first sheet in the spreadsheet. You can also specify if you want the spreadsheet to be in the XML format or not. If the file is in the XML format, you can save it as a .xlsx file, thus making it readable by Excel 2007 or later.

Once you have the spreadsheet object, the first thing you most likely will want to do is add a header row to the file. Adding a header row uses the same functionality you would use if you were just adding a single row to the file. The function you will call is `SpreadsheetAddRow`. This function accepts the spreadsheet object and a comma delimited list of data. This row of data will be added to the last row of data in the file.

If you are adding many rows of data to your spreadsheet and they are in a query or array object, you can use the function `SpreadsheetAddRows`. Using this function rather than looping over your data and calling `SpreadSheetAddRow` is much more desirable; it is less code and faster. The `SpreadsheetAddRows` function accepts the spreadsheet object and the query or array object. All rows will be added after the last row of data in the spreadsheet. It is possible to change where the rows are inserted and if they overwrite the data or not by supplying some additional attributes. For further information on the subject, you can go to: http://help.adobe.com/en_US/ColdFusion/10.0/CFMLRef/WSc3ff6d0ea77859461172e0811cbec22c24-6790.html.

Once the data is in your spreadsheet object, you may want to format the look and feel of the data a bit. The most common formatting done with a spreadsheet is to bold the header row so that it stands out. Formatting data is an easy task when using the `SpreadsheetFormatRow` function. This function accepts the spreadsheet object, a struct containing the formatting data, and the row that the formatting should be applied to.

There are a number of different styles that can be specified in the formatting struct that is passed to the function. This is not a complete list of the formats available, but it should give you an idea of the amount of control you have when formatting your spreadsheet.

- font
- fontsize
- italic
- bold
- alignment
- textwrap
- bgcolor
- bottomborder
- bottombordercolor
- topborder
- topbordercolor
- leftborder
- leftbordercolor
- rightborder
- rightbordercolor

When exporting data, sometimes the data you are outputting needs to be separated. When dealing with spreadsheets, this is done by creating separate sheets in the spreadsheet file. To create a new

sheet in your spreadsheet, you would use the `SpreadsheetCreateSheet` function. This function allows you to pass in the spreadsheet object and, optionally, the name to use for the sheet. Once this sheet is created, you will need to switch to the sheet before entering data. To do this, call the `SpreadsheetSetActiveSheet` function. This function accepts the spreadsheet object and the sheet name of the sheet you wish to set as the active sheet. Once this call is made, all updates will now occur on that sheet. As this function requires you provide the name of the sheet, and not the sheet position, it is recommended that you always name your sheets.

Writing a Spreadsheet

Once you have added all the data to your spreadsheet and made all the necessary style and formatting updates, it will be necessary to write the spreadsheet to the file system. To do this, call the `SpreadsheetWrite` function.The `spreadsheetWrite` function accepts the spreadsheet object as well as the location of where the spreadsheet should be written. When specifying the path, remember to include the file name.

`SpreadsheetWrite` also has some optional arguments that may be useful. In addition to being able to specify if the function should overwrite a pre-existing file if there already is one, you can also specify a password. This is a very useful feature when dealing with sensitive information.

Spreadsheet Example

Below is an example of all the spreadsheet functionality discussed in this section. In this example, we create a spreadsheet, add some data to it, format the data, and then repeat this for a second sheet. Once completed, we then write the file to the file system.

```
//Create Spreadsheet
spreadsheetObj = SpreadsheetNew('Names');
//Add Header Row
SpreadSheetAddRow(spreadsheetObj,'ID,Name');
//Add Data
SpreadSheetAddRow(spreadsheetObj,'1,Simon');
SpreadSheetAddRow(spreadsheetObj,'2,Carl');
//Format Header
SpreadsheetformatRow(spreadsheetobj,{bold=true,alignment='center'},1);
//Add Sheet
SpreadSheetcreateSheet(spreadsheetobj,'Towns');
//Switch to Names Sheet
SpreadsheetSetActiveSheet(spreadsheetobj,'Towns');
```

```
//Add Header Row
SpreadSheetAddRow(spreadsheetObj,'ID,Town');
//Add Data
SpreadSheetAddRow(spreadsheetObj,'1,Detroit');
SpreadSheetAddRow(spreadsheetObj,'2,Sheffield');
//Format Header
SpreadsheetformatRow(spreadsheetobj,{bold=true,alignment='center'},1);
//Write File
Spreadsheetwrite(spreadsheetobj,expandpath('myData.xls'),true);
```

Hands On 23

By Simon Free

In this hands on example, you are going to make a call to Twitter to get a list of your latest tweets and output them to the contact page.

Tags Used: <cfhttp>, <cfloop>, <cfoutput>
Functions Used: dateFormat

1. Open up the `/www/contact.cfm` file in your code editor.
2. First, make a `<cfhttp>` call to get the data from Twitter. Create a `<cfhttp>` tag directly after the `<cfset>` block. Your code should be on or around line 7. The `<cfhttp>` tag should have the following attributes:
   - **url**: https://api.twitter.com/1/statuses/user_timeline.xml?&count=5&screen_name=[Your Screen Name]
   - **method**: get
   - **result**: twitterFeed

Note: If you do not have a twitter screen name you can use the 'simonfree' screen name for now.

3. The code should look similar to this:

```
<cfhttp url="https://api.twitter.com/1/statuses/user_timeline.
xml?&count=5&screen_name=simonfree" method="get" result="twitterFeed" />
```

4. Now that you have the feed data in the `twitterFeed` variable, you need to output it on the page. To do this, loop over the XML data that was received from Twitter and create a `<cfloop>` tag. Locate the comment that reads `Twitter Output` and create the `<cfloop>` tag before the first `<li>` tag. The `<cfloop>` tag should have the following properties:
   - **array**: #xmlParse(twitterFeed.fileContent).statuses.status#
   - **index:** feedItem
5. Place the closing `</cfloop>` tag after the closing `</li>` tag.
6. Inside the `<li>` tags, place the following line of code:

```
#dateformat(feedItem.created_at.xmlText,'mm/dd/yyyy')# - #feedItem.text.
xmlText#
```

7. Now wrap the entire `<cfloop>` block in a `<cfoutput>` tag so that the variables will be displayed to the user. Your code should look similar to this:

```
<ul>
    <cfoutput>
        <cfloop array="#xmlParse(twitterFeed.fileContent).statuses.
status#" index="feedItem" >
            <li>
                #dateformat(feedItem.created_at.xmlText,
'mm/dd/yyyy')# - #feedItem.text.xmlText#
            </li>
        </cfloop>
    </cfoutput>
</ul>
```

8. Open up the `/www/contact.cfm` page in your browser and notice that the 5 latest tweets are now displaying under the 'Latest Tweet' heading.

Hands On 24

By Simon Free

In this next hands on, you are going to add functionality that will allow you to export a blog post in PDF format. We will also add a watermark to the PDF.

Tags Used: <cfparam>, <cfset>, <cfif>, <cfdocument>, <cfdocumentitem>, <cfpdf>, <cfcontent>
Functions Used: EntityLoad, isNull, toBinary

1. Create a new file in the `/www/` folder called `ExportToPDF.cfm`.
2. Open the `/www/ExportToPDF.cfm` file in your code editor.
3. On the first line of the file, create a `<cfparam>` tag with the following attributes:
   - **name**: url.id
   - **default**: 0
4. Next, load in the blog post so you can get the data needed. To do this, create a `<cfset>` that does an `entityLoad` call requesting a `blogPost` with the ID of 'url.id' and sets it into a `blogPost` variable by adding the following line of code:

```
<cfset blogPost = EntityLoad('blogPost',url.id,true) />
```

5. Before proceeding with the PDF generation, make sure that the ID that was passed was for a valid blog post by checking to make sure the `blogPost` object does not have a null value. To do this, create a `<cfif>` statement and use the `isNull` function on the `blogPost` variable. Your code should look similar to this:

```
<cfparam name="url.id" default="0"  />
<cfset blogPost = EntityLoad('blogPost',url.id,true) />


<cfif !isNull(blogPost)>


</cfif>
```

6. Inside of the `<cfif>` block, create a `<cfdocument>` tag with the following attribute:
   - **format**: PDF
7. Add a closing `</cfdocument>` tag.
8. Inside the `<cfdocument>` tags add the following code:

```
<cfoutput>
<h1>
    #blogPost.title#
</h1>
<p>
    <strong>Date Posted</strong>: #dateformat(blogPost.datePosted,
'mm/dd/yyyy')#
</p>
#blogPost.body#
<p>
</cfoutput>
```

9. Open up the `/www/blog.cfm` page in your browser, navigate to a blog post, and click on 'Export To PDF'. You should see a PDF of the blog post.
10. Next, add a header to the PDF which states it came from the website. Go back to the `ExportToPDF.cfm` file in your code editor and add a `<cfdocumentitem>` tag inside the `<cfdocumenttag>` that has the following properties:
    - **type**: header
11. Inside the `<cfdocumentitem>` add the following code:

```
<h1 style="text-align:center;">Generated from our website</h1>
```

12. Your `<cfdocument>` code should looks similar to this:

```
<cfdocument format="PDF">
    <cfdocumentitem type="header">
        <h1 style="text-align:center;">Generated from our website</h1>
    </cfdocumentitem>
    <cfoutput>
    <h1>
        #blogPost.title#
    </h1>
```

```
    <p>
        <strong>Date Posted</strong>: #dateformat(blogPost.
datePosted,'mm/dd/yyyy')#
    </p>
    #blogPost.body#
    </cfoutput>
</cfdocument>
```

13. Go back to the `ExportToPDF.cfm` file in your browser and refresh. You should now see the title displayed in the PDF.

14. Now add a watermark to the PDF. First, save the generated PDF to a variable rather than having it automatically output on the page. You can do this by updating the `<cfdocument>` tag and adding the following property:

    - **name**: myGeneratedPDF

15. If you were to re-run the `ExportToPDF.cfm` file in your browser, you would no longer see the PDF as it is now stored in a variable. Now, create a `<cfpdf>` tag that will allow us to add the watermark. After the closing `</cfdocument>` tag, create a `<cfpdf>` tag with the following attributes:

    - **action**: addWatermark
    - **source**: myGeneratedPDF
    - **image**: assets/images/watermark.jpeg
    - **foreground**: yes
    - **overwrite**: yes

16. Once again, if you were to run this file, you would still see no output; you need to manually tell ColdFusion to display the PDF. This is done with the `<cfcontent>` tag. After the `<cfpdf>` tag, create a `<cfcontent>` tag with the following attributes:

    - **variable**: #toBinary(myGeneratedPDF)#
    - **type**: application/pdf

17. Reload the `ExportToPDF.cfm` file in your browser and you should now see your PDF with a large W in the middle. The W is the Watermark.

Hands On 25

By Simon Free

In this hands on, you are going to create an image file upload. Once the image is uploaded, you are going to resize it to fit specific measurements.

Tags Used: <cffile>, <cfif>, <cfset>
Functions Used: listFindNoCase, getReadableImageFormats, imageScaleToFit, imageWrite, getTempDirectory

1. Open up the `/www/admin/content/portfolio/editportfolio.cfm` file in your code editor.
2. The first thing that needs to be done is to check that an image has been provided in the form to upload. To do this create a `<cfif>` tag that checks if the `form.image` variable has a length. Locate the `Image Upload Process` comment and add it after.
3. Upload the image onto the server. Once on the server, you can manipulate it however desired. To start, find the `Image Upload Process` comment and create a `<cffile>` tag with the following properties:
   - **action**: upload
   - **filefield**: image
   - **destination**: #getTempDirectory()#
   - **nameconflict**: makeunique
4. This tag will upload the file that is in the form field called image and will place it into a temporary directory on the server, making the file name unique if the file already exists on the server. The `<cffile>` tag should look similar to this:

```
<cfif len(form.image)>
    <cffile action="upload" filefield="image" destination=
"#getTempDirectory()#" nameconflict="makeunique" />
</cfif>
```

5. Next, confirm that the file that was uploaded is of a file type that can be processed. To do this, create a `<cfif>` statement that uses the `ListFindNoCase` function. The first argument of the function will be a call to the `getReadableImageFormats()` function, and the second argument will be the serverFileExt variable in the cffile structure. Your code should look similar to this:

```
<cfif listFindNoCase(getReadableImageFormats(),cffile.serverFileExt)>
</cfif>
```

6. Inside the <cfif> tag you will add the image resize logic. If the checks in the <cfif> have passed, then we know that the image is safe to resize. Inside the <cfif> create a <cfset> tag. In this set tag you are going to read in the image using the imageRead function and set it to the variable name imageObject. Your code should look similar to this:

```
<cfset imageObject = imageRead(cffile.serverDirectory & '/' & cffile.
serverfile) />
```

7. Next, call the imageScaleToFit function and pass it the imageObject with the dimensions to resize the image. As the imageScaletoFit function does not return anything, you do not need to assign it to a variable. Add the following line of code below the <cfset> tag:

```
<cfset imageScaleToFit(imageObject,'202','131') />
```

8. Once the image has been resized, save the image object back to the file system. When writing the file, write it to a location in the web root, rather than a temporary directory, so that it can be displayed to the user. Use the imageWrite function and pass it the imageObject variable and the desired path to where the file should be written. Your code will look similar to this:

```
<cfset imageWrite(imageObject,expandpath('../../../assets/images/
portfolio/#cffile.serverfile#')) />
```

9. Once the file has been written, overwrite the form.image value with the image file value so it can be stored in the database. The name of the image is stored in the cffile structure under the key of serverfile. Your <cfset> tag will look similar to this:

```
<cfset form.image = cffile.serverfile />
```

10. It is possible that the user has tried to upload a file that is not accepted on the server. If this occurs we will simply remove the value from the form.image variable. Before the closing </cfif> on or around line 32, create a <cfelse> tag and place the following code inside:

```
<set form.image='' />
```

11. Your final <cfif> block should look similar to this:

```
<cfif listFindNoCase(getReadableImageFormats(),cffile.serverFileExt)>
    <cfset imageObject = imageRead(cffile.serverDirectory & '/' & cffile.
serverfile) />
    <cfset imageScaleToFit(imageObject,'202','131') />
    <cfset imageWrite(imageObject,expandpath('../../../assets/images/
```

```
portfolio/#cffile.serverfile#')) />
     <cfset form.image = cffile.serverfile />
<cfelse>
     <cfset form.image='' />
</cfif>
```

12. Open up the `/www/admin/` page in a browser and navigate to the portfolio section.
13. Once in the portfolio section click on 'New Portfolio'.
14. Fill in the form and provide an image to be uploaded.
15. Click on submit to create the new portfolio item.
16. Open up the `/www/portfolio.cfm` page in your browser and confirm that the new portfolio item is being displayed along with the resized image.

Hands On 26

By Simon Free

In this hands on, we are going to import and export data into the blog section using Excel.

Tags Used: <cffile>, <cfspreadsheet>, <cfloop>, <cfset>, <cfscript>, <cfheader>, <cfcontent>
Functions Used: getTempDirectory, EntityNew, EntitySave, ormFlush, EntityLoad, spreadsheetNew, spreadsheetAddRow, spreadsheetFormatRow, spreadsheetAddRows, spreadsheetWrite

1. First, let's look at the import process. Open up the `/www/admin/content/blog/importBlog.cfm` file in your code editor.
2. Before importing a spreadsheet, we must first move it to the server. Locate the `Upload File` comment tag. On the line below, create a `<cffile>` tag with the following attributes:
    - **action**: upload
    - **destination**: #getTempDirectory()#
    - **filefield**: importFile
    - **nameconflict**: makeunique
3. Your code should look similar to this:

```
<cffile action="upload" destination="#getTempDirectory()#"
filefield="importFile" nameconflict="makeunique" />
```

4. Once the file is on the server, we can parse the file and put it into a query variable by using the `<cfspreadsheet>` tag. Locate the `Read Spreadsheet` comment and create a `<cfspreadsheet>` tag on the line below with the following attributes:
    - **action**: read
    - **src**: #cffile.serverDirectory#/#cffile.serverfile#
    - **query**: importData
    - **headerrow**: 1
    - **excludeheaderrow**: true
5. Your code should look similar to this:

```
<cfspreadsheet action="read" src="#cffile.serverDirectory#/#cffile.
serverfile#" query="importData" headerrow="1" excludeheaderrow="true"
/>
```

6. Once the spreadsheet has been read into a query variable, loop over the query and create a new `blogPost` entity for each row. To do this, locate the `Import Data` comment tag and create a new `<cfloop>` tag below it with the following attribute:

   - **query**: importData

7. Inside the `<cfloop>` tag, create a `<cfset>` tag that loads in a new `blogPost` entity and saves it in a variable called `blogPost`.

8. Below the `<cfset>`, create new `<cfset>` tags that set the title, summary, body, and date posted values. Note that because the Date Posted column in the spreadsheet has a space in it, you must use bracket notation to access the value rather than dot notation. Your code should look similar to this:

```
<cfloop query="importData">
    <cfset blogPost = EntityNew('blogPost') />
    <cfset blogPost.title = importData.title />
    <cfset blogPost.summary = importData.summary />
    <cfset blogPost.body = importData.body />
    <cfset blogPost.dateposted = importData['Date Posted'] />
</cfloop>
```

9. The next step is to save the entity. Just before the closing `</cfloop>` tag, create a `<cfset>` tag that calls `EntitySave` on the `blogPost` entity.

10. After the closing `</cfloop>` tag, create another `<cfset>` tag that calls `ormFlush()`. This will commit all changes to the database. Your final code should look similar to this:

```
<!--- Upload File--->
<cffile action="upload" destination="#getTempDirectory()#"
filefield="importFile" nameconflict="makeunique" />


<!--- Read Spreadsheet --->
<cfspreadsheet action="read" src="#cffile.serverDirectory#/#cffile.
serverfile#" query="importData" headerrow="1" excludeheaderrow="true" />


<!--- Import Data --->
<cfloop query="importData">
    <cfset blogPost = EntityNew('blogPost') />
```

```
    <cfset blogPost.title = importData.title />
    <cfset blogPost.summary = importData.summary />
    <cfset blogPost.body = importData.body />
    <cfset blogPost.dateposted = importData['Date Posted'] />
    <cfset EntitySave(blogPost) />
</cfloop>


<cfset ormFlush() />
```

11. Open up the `/www/admin/content/blog/importBlog.cfm` page in your browser.
12. Select an Excel file and click 'Import'. A template Excel file can be found at: `/www/assets/blogImport.xslx`.
13. Go to the `/www/admin/content/blog/listblogpost.cfm` page in your browser and you will see the imported blog posts.
14. Now that the import process is completed, you are going to create an export process. Create a new file called `exportBlog.cfm` in the `/www/admin/content/blog/` folder.
15. Open up the `/www/admin/content/blog/exportBlog.cfm` file in your code editor.
16. For this task, you are going to write some of it in `<cfscript>`, so you will need to create a new `<cfscript>` block.
17. Inside the `<cfscript>`, create a variable called `blogPosts` that contains all the `blogPost` entities. You can do this by calling `EntityLoad('blogPost')`.
18. On the next line, create a new spreadsheet called `exportSpreadhseet` by calling the `SpreadsheetNew` function and pass it in a string of 'Blog Posts'. This will name our first sheet Blog Posts. Your code should look similar to this:

```
<cfscript>
    blogPosts = EntityLoad('blogPost');
    exportSpreadsheet = SpreadsheetNew('Blog Posts');
</cfscript>
```

19. Once you have the spreadsheet object created, add a heading row. To do this, call the `SpreadsheetAddRow` function and pass it in the `exportSpreadhseet` object with a comma delimited string of the column headings you want. For this example, the code will look similar to:

```
SpreadsheetAddRow(exportSpreadsheet,'ID,Title,Summary,Body,Date
Posted');
```

20. As this row is a header row, you will want to add some styles to denote that. Using the `SpreadsheetFormatRow` function, format the first row so it is Bold and aligned center. To do this, place the following code below the `SpreadsheetAddRow` call:

```
SpreadsheetFormatRow(exportSpreadsheet,{bold=true,alignment=
'Center'},1);
```

21. Once the header is formatted, add the remaining data to the spreadsheet. Loop over the query and call `SpreadsheetAddRow` on each iteration and pass in the `spreadsheetObject`, in this case `exportSpreadsheet`, and a list of data. The code should look like this:

```
for(blogPost in blogPosts){

    SpreadsheetAddRow(exportSpreadsheet,'#blogPost.id#,#blogPost.
title#,#blogPost.summary#,#blogPost.body#,#blogPost.datePosted#');

}
```

22. Now that all the data is in the spreadsheet, save the spreadsheet to the server. To do that, call the `spreadsheetWrite` function and pass it in the `spreadsheetObject`, which is the path of the file we want it to be written to, and you can choose to overwrite the file that might already be there. In this case, write the file to the servers temp directory and have it overwrite any file that might already exist with the same name by using the following code:

```
SpreadsheetWrite(exportSpreadsheet,getTempDirectory() & 'blogPosts.
xls',true);
```

23. Our completed `<cfscript>` block should look similar to this:

```
<cfscript>

    blogPosts = EntityLoad('blogPost');

    exportSpreadsheet = SpreadsheetNew('Blog Posts');

    SpreadsheetAddRow(exportSpreadsheet,'ID,Title,Summary,Body,Date
Posted');

    SpreadsheetFormatRow(exportSpreadsheet,{bold=true,alignment=
'Center'},1);

    for(blogPost in blogPosts){

        SpreadsheetAddRow(exportSpreadsheet,'#blogPost.id#,#blogPost.
title#,#blogPost.summary#,#blogPost.body#,#blogPost.datePosted#');

    }
```

```
    SpreadsheetWrite(exportSpreadsheet,getTempDirectory() & 'blogPosts.
xls',true);
</cfscript>
```

24. Now that the spreadsheet has been created, you need to serve it up to the user. Use a <cfheader> tag and a <cfcontent> tag. First, start with the <cfheader> tag, which tells the browser to serve it up in line and what the filename should be. Place the following code after the closing </cfscript> tag:

```
<cfheader name="Content-Disposition" value="inline; filename=blogPosts.
xls" />
```

25. Finally, use the <cfcontent> tag, which tells what file needs to be served and what type of file it is.You can do this by using the following code, which should be placed right after the <cfheader> tag:

```
<cfcontent file="#getTempDirectory()#blogPosts.xls" type="vnd.ms-excel"
/>
```

26. Your completed file should look similar to this:

```
<cfscript>
    blogPosts = EntityLoad('blogPost');
    exportSpreadsheet = SpreadsheetNew('Blog Posts');
    SpreadsheetAddRow(exportSpreadsheet,'ID,Title,Summary,Body,Date
Posted');
    SpreadsheetFormatRow(exportSpreadsheet,{bold=true,alignment=
'Center'},1);
    for(blogPost in blogPosts){
        SpreadsheetAddRow(exportSpreadsheet,'#blogPost.id#,#blogPost.
title#,#blogPost.summary#,#blogPost.body#,#blogPost.datePosted#');
    }
    SpreadsheetWrite(exportSpreadsheet,getTempDirectory() & 'blogPosts.
xls',true);
</cfscript>
```

```
<cfheader name="Content-Disposition" value="inline; filename=blogPosts.
xls" />

<cfcontent file="#getTempDirectory()#blogPosts.xls" type="vnd.ms-excel"
/>
```

27. In a browser, navigate to the `/www/admin/content/blog/exportBlog.cfm` page. You might be prompted to download a file if the download does not start automatically. Open up the Excel file and review the data that has been exported.

11

Caching

By Dan Skaggs

About Dan Skaggs

Dan Skaggs has been coding in one form or another since the mid-1980s. A web developer since 1998, Dan has been developing ColdFusion applications since 2000 and has spoken at several ColdFusion conferences. Dan is the co-founder of Web-Meister Designs, a web development consulting firm he and his wife started in 1999 and is a contributing partner to the Model-Glue MVC framework for ColdFusion. Dan also enjoys shooting, motorcycling, and amateur radio.

As you write more and larger ColdFusion applications, you will start looking for ways to improve the performance of your applications. There are many ways to do this, but perhaps the easiest is to use ColdFusion's caching mechanisms to reduce the amount of work your application has to do over and over. Caching simply refers to the idea that you create a piece of content or data once and hold it in application memory for some period of time. During that time frame, any part of your application that needs that content or data uses the copy that was previously generated rather than regenerating it.

ColdFusion has several different caching mechanisms built in, but they generally fall into two main categories--programmatic caching and application server caching.

Programmatic Caching

This type of caching is controlled by your application code. You decide which parts of your application would benefit from being cached and use CFML tags and attributes to determine what content is cached, as well as how long your application should use the cached copy before it is discarded and/or regenerated.

Query Caching

One of the first types of caching that developers turn to is query caching. Query caching stores the result of a `cfquery` action in memory so you can quickly reuse the dataset returned by the query. This is especially useful for queries whose result set changes infrequently or for queries that take some time to execute.

To illustrate this, consider a common use case of selecting countries that a company ships its products to. That list of countries likely doesn't change very often, so it would be a good candidate to be cached by ColdFusion. Your original query might look something like this:

```
<cfquery datasource="myDatabase" name="shipToCountries">
     SELECT countryId, countryName
     FROM countries
</cfquery>
```

With this query, everywhere in your code that needs a list of countries you can ship products to will create a call to the database and retrieve this information. By adding one attribute to your `cfquery` tag, you can reduce the number of times that query is run. For example, the following query will be cached in server memory for 1 hour, meaning that no matter how many times a page that requires that country list is accessed, the query will only be run a maximum of 24 times per day.

```
<cfquery datasource="myDatabase" name="shipToCountries"
cachedWithin="#createTimeSpan( 0, 1, 0, 0 )#">
     SELECT countryId, countryName
     FROM countries
</cfquery>
```

The `cachedWithin` attribute to the `cfquery` tag instructs ColdFusion to execute this query ONLY if it does not have a result set from that exact query in memory already. The `createTimeSpan()` method is used to give ColdFusion a range of time to compare to the current system time in order to decide whether to use the result set in memory or to execute the query again and cache the result of that execution. In this case, the timespan is 0 days, 1 hour, 0 minutes, and 0 seconds.

An important note to remember here is that ColdFusion caches result sets based on the SQL statement contained in the `cfquery` tag. Therefore, if you have a dynamic query, a new cached value will be stored for every variation of the query. For instance, the following query would create a separate cached query result set for each different value of `form.firstName` that was provided (Bob, Bill, Mitch, etc).

```
<cfquery datasource="myDatabase" name="user"
cachedWithin="#createTimeSpan( 0, 1, 0, 0 )#">
    SELECT userId, firstName, lastName
    FROM users
    WHERE lastName = <cfqueryparam cfsqltype="cf_sql_varchar"
value="#form.firstName#" />
</cfquery>
```

Caching Generated Content with cfcache

For caching non-query content, ColdFusion provides the `cfcache` tag. The `cfcache` tag can be used to cache an entire rendered page of content for a specified amount of time. An example of this might be a page listing products that all belong to the same category. On high-traffic sites, caching the rendered output for this page even for as little as 1-2 minutes could cause the application to run significantly faster. To do this, you simply wrap your page in a `cfcache` tag like so:

```
<cfcache action="cache" timespan="#createtimespan(0,0,2,0)#">
    <html>
        <head></head>
        <body>
            ...page content here...
        </body>
    </html>
</cfcache>
```

The `cfcache` tag can also be used to cache HTML fragments that are likely to be used across multiple pages. An example might be taking our country query from earlier and building a select box for use on registration forms, payment screens, search forms, etc. You could potentially run the

query, build the HTML for a select list using the query results, and cache that HTML fragment for use later on by all the different pages that needed that fragment. Below is an example of how this would work to cache the HTML fragment of the select list for one day.

```
<cfcache action="cache" timespan="#createtimespan(1,0,0,0)#">

    <cfquery datasource="myDatabase" name="shipToCountries">
      SELECT countryId, countryName
      FROM countries
    </cfquery>

    <select name="country" id="country">
        <cfloop query="shipToCountries">
            <option value="#shipToCountry.countryId#">#shipToCountry.
countryName#</option>
        </cfloop>
    </select>

</cfcache>
```

The `cfcache` tag offers fine-grained control on removing content from the cache as well. This is accomplished by using `action="flush"` on the `cfcache` tag. Coupled with the `expireURL` attribute, you can flush the entire cache, a group of pages that match a particular URL pattern, or a single content item. This is especially useful when the rendered content that you're caching would be affected by changes to the database that drives it. In the example above, were a country added to the countries table, the flush action could be used to ensure that bit of content was flushed from the cache and rebuilt with the new values on the next request.

Caching Generated Content Manually

You can manually place data or content inside the internal ColdFusion Cache, if you prefer. ColdFusion uses the very popular and robust ehCache product internally. ehCache is an enterprise level caching product and has many benefits apart from the integrations you'll find in ColdFusion. Let's look at how we might put data in the cache, then retrieve it.

```
CachePut(id, value);
CacheGet(id);
```

In the example above, the `id` must be unique to the cache. The value is whatever data or content you want to cache. Here's a practical example:

```
<cfset CachePut(42, 'AnswerToEverything') />
<cfset CacheValue = CacheGet(42) />
<cfdump var="#CacheValue#" />
```

Once the code completes, you'll have inserted the value 'AnswerToEverything' into the cache and assigned it the key of 42. Then you will have requested the value matching the key of 42 and received your original text ''AnswerToEverything'.

You'll do most of your work with `CachePut`; to help you cache the right data, there are two optional arguments to `CachePut`:

```
CachePut(id, value[, timeSpan[, idleTime]]);
```

The `timespan` argument is the duration the cache should keep the object. Use `CreateTimeSpan()` to easily make a timespan for your cache function.

The `idleTime` argument is a number of days in which to toss out the cached item if the item has not been accessed in that time. In the next example, we'll cache our 'AnswerToEverything' value under the key 42 and ask the cache to cache it for 2 days.

```
<cfset CachePut(42, 'AnswerToEverything', CreateTimespan( 2, 0, 0, 0) )
/>
<cfset CacheValue = CacheGet(42) />
<cfdump var="#CacheValue#" />
```

The cache will get rid of items automatically according to the rules you give it. In the above example, the key and value assigned to 42 will be evicted from the cache in 2 days. You can manually evict whatever you want from the cache with the `CacheRemove()` function. The below line of code will manually remove the key and value assigned to 42.

```
<cfset CachePut(42, 'AnswerToEverything', CreateTimespan( 2, 0, 0, 0) )
/>
<cfset CacheValue = CacheGet(42) />
<cfdump var="#CacheValue#" />
<cfset CacheRemove(42) />
<cfset CacheValue = CacheGet(42) />
```

```
<cfdump var="#isNull(CacheValue)#" />
<cfdump var="#CacheValue#" />
```

If you run the code above, you'll see the cached value appear in the first dump. Once we use `CacheRemove()`, if we try to retrieve the key again, we'll get a null. The example wrapped the first null value in an `IsNull()` function so you can see how to properly check and work with this case. If you try to use a null value in ColdFusion, you will get an error, because of the last line in which we attempt to use a value that is null.

Getting Metrics about Caching

Caching saves processing and database time at the expense of memory. In order to do caching well, you'll need to know how well your caching strategy is performing. The first important concept in caching is a `cache-hit`. A `cache-hit` occurs when a request is made for a cached item and the item is already in the cache. The second important concept in caching is the concept of the `cache-miss`. A `cache-miss` occurs when an item is requested from the cache and the cache does not have it. You can get metrics about `cache-hits` and `cache-misses` for each of your cached keys. When you run this example, you'll see the number of `cache-hits` and `cache-misses` for our item # 42.

```
<cfset CachePut(42, 'AnswerToEverything', CreateTimespan( 2, 0, 0, 0) )
/>
<cfset CacheValue = CacheGet(42) />
<cfdump var="#CacheValue#">
<cfdump var="#cacheGetMetaData(42)#" />
```

It is generally a good idea to seek the largest ratio between `cache-hits` and `cache-misses`. This means the cached item is being requested from the cache. However, the main reason behind caching is to save processing, data access, web service requests, and other such resources from being required for each request. Thus, if you have a computationally expensive request, it may be ok if the difference between `cache-hits` and `cache-misses` isn't so much.The right solution depends on your hardware, application, types of requests and other factors. Since you now know how to get information about your cached content, feel free to experiment.

Application Server Caching

In addition to the programmatic methods of caching discussed earlier, ColdFusion also provides some caching mechanisms managed at the server level. Settings for these are found in the ColdFusion Administrator web application and apply to all ColdFusion applications being served by that particular instance of ColdFusion. The three most important of these are Trusted Cache, Cache Template in Request, and Save Class Files. These three caching mechanisms are primarily used on production systems and are not recommended for use on a ColdFusion instance that is being used for development purposes.

Trusted Cache

With Trusted Cache disabled, each time you request a ColdFusion file, the server will check to see if that file is different from the version that it previously compiled to Java bytecode. If the file has not changed, the Java bytecode previously compiled is used. If the file is different, ColdFusion recompiles the file and uses the new Java bytecode just compiled. Turning on the Trusted Cache option in the ColdFusion Administrator will cause ColdFusion to no longer scan your source code files for changes. Enabling this option eliminates the check for differences on each request for the file. That means that the first time a ColdFusion file is requested, the server will compile it as normal. Thereafter, it will always use the previously compiled version of the file, even if you have updated the file in the meantime. The ColdFusion Administrator provides a way to manually clear the Trusted Cache so that the next request will recompile the current version of the source code and include any changes to the files since the files were last compiled. Additionally, the ColdFusion Admin API provides a way to clear the trusted cache from your code, possibly as part of an automated deployment process.

Since there can be potentially dozens of ColdFusion files accessed during a single request cycle, enabling Trusted Cache can have a significant positive effect to the responsiveness of the server. On production sites, where source code is likely not changing often, it is unnecessary to have ColdFusion constantly checking source code files for changes, and thus that step can be eliminated. Conversely, if this setting were enabled on a development machine, your development process would have to include a step to manually clear the Trusted Cache every time you made a change to a file, which is less than productive.

Cache Template in Request

The Cache Template in Request setting is very similar to the Trusted Cache setting discussed above, with the exception that the check is only skipped for that particular request. For instance, if you call a file more than once during a request with this setting enabled, the server will compile the file to Java bytecode on the first invocation and will use that same bytecode for each subsequent call to the file without checking to see if the file is different up through the end of the request. However, unlike Trusted Cache, subsequent requests will recompile the file the first time it is called during that request. So while Trusted Cache disables file change checks until you clear the Trusted Cache or restart the ColdFusion server, Cache Template in Request only disables file change checks for the duration of the current request.

It should be noted that if you using if you have enabled the Trusted Cache feature, you won't see any performance increases from also enabling Cache Template in Request, as Trusted Cache essentially supersedes this feature due to the way it works.

Save Class Files

We've mentioned earlier how the ColdFusion server compiles ColdFusion files from source code down to Java bytecode. Those bytecode files are housed in Java class files. Typically, this happens when the ColdFusion file is first accessed after the server starts. However, you can enable this option to have ColdFusion save the compiled class files to a directory on the disk and read them back into memory after a restart. Depending on how many class files and the amount of traffic to your application, this can be faster than recompiling the source code to Java bytecode again.

Your system has a finite amount of memory, and caching done incorrectly can cause more trouble than you might initially think. There are a couple settings in the ColdFusion administrator to help control the amount of caching of certain features.

Maximum Number of Cached Templates (ColdFusion Admin -> Server Settings > Caching) ColdFusion will cache templates, pages, and ColdFusion Components into memory so they do not have to be read from disk when requested.

You can clear the cached templates by clicking the ‘Clear Template Cache Now’ button. It is a good idea to do this when you change your source code files on production. You can also clear a specific folder if you know which files you specifically want to clear from the template cache.

Maximum Number of Queries Cached (ColdFusion Admin -> Server Settings > Caching) ColdFusion will let you place a limit on the number of cached queries in your application. As a cached query is requested, it moves to the top of the stack. Unused queries move toward the bottom of the stack. The query unused the longest is evicted from the Query Cache as the cached query count hits the limit you set in the ColdFusion administrator.

You can clear the cached queries by clicking the ‘Clear Query Cache Now’ button. You may need to do this if the underlying data changes, but the current caching strategy on the server is still showing the old information.

Hands On 27

By Simon Free

In this hands on, you are going to add caching to the blog and to the portfolio sections.

Tags Used: <cfcache>, <cfset>
Functions Used: sleep, cacheGet, isNull, cachePut

1. Before you add caching, turn on the debugging information of the page so that the changes we are about to make can be detected. Login to your ColdFusion administrator. The ColdFusion Administrator will be located at http://localhost:8500/CFIDE/administrator/ .
2. Once logged in, go to 'Debugging Output Settings' under Debugging & Logging.
3. Check the 'Enable Request Debugging Output' option and click 'Submit Changes'.
4. Open up the `/www/blog.cfm` page in your browser.
5. Scroll to the bottom of the page and take note of the additional information being displayed.
6. Open up the `/www/blog.cfm` file in your code editor.
7. The first thing you need to do is create a `<cfcache>` tag. Locate the `Blog Posts` comment and create a `<cfcache>` tag on the line below it with the following attributes:
   - **action**: cache
   - **timespan**: #createtimespan(0,1,0,0)#
8. Place the closing `</cfcahe>` tag after the closing `</cfoutput>` tag.
9. Reload the `/www/blog.cfm` page in your browser. Take note of the execution time.
10. Refresh the page again and take note of the execution time. If this were a larger page, you would have noticed a decrease in the execution time.
11. One problem right now is that even though the output is being cached, all blog posts are being pulled at the beginning of the page request. To rectify this, move the `<cfset>` located on line 1 so that it is inside the `<cfcache>` tag.
12. If you were to run the page now, the output would still be cached as the cache will not expire for an hour. To clear the cache, call the `<cfcache>` tag with an action value of flush. Create a new page in the `/www/` folder and call it `clearCache.cfm`.
13. Create a `<cfcache>` tag with the following attribute:
    - **action**: flush
14. In your browser, go to the `/www/clearcache.cfm` page. You will not see any output but the cache is now cleared.

15. Go back to the `/www/blog.cfm` page in your browser and refresh. Take note of the execution time.
16. Refresh the page again and compare the two execution times. If the page took long enough to run, you will see an improvement in the execution times.
17. To make the differences more visible, let's slow down the page load. Right after the opening `<cfcache>` tag, add the following line of code:

```
<cfset sleep(500) />
```

18. The line of code you added will have ColdFusion wait for 500 milliseconds before it continues. Before testing this functionality, go back to the `clearcache.cfm` file in your browser and reload the page.
19. Now that the cache is cleared, go back to the `blog.cfm` file in your browser and reload it. Take note of the execution time.
20. Refresh the `blog.cfm` page and review the execution time. You will notice that the execution time is drastically different. This is because the cached content is being called, rather than processing the block of code.
21. Remove the sleep tag we just entered.
22. Now that the `<cfcache>` tag is in place, lets look at some programmatic caching. Open up the `/www/portfolio.cfm` file in your code editor.
23. At the top of the page you should have a `<cfquery>` tag. Prior to this, create the following `<cfset>` tag:

```
<cfset myPortfolio = cacheGet('myPortfolio') />
```

24. The `cacheGet` function pulls the cache from memory that has the identifier of `myPortfolio`. Right now, there is nothing cached with that identifier. First check if anything was returned; if not, you need to execute the code and store the result in cache. After the `<cfset>` tag, create a `<cfif>` tag, which will check if the `myPortfolio` variable is null. The code should look similar to this:

```
<cfif isNull(myPortfolio)>


</cfif>
```

25. Move the `<cfquery>` that was already in the file inside the `<cfif>` tag.
26. After the query, create a `<cfset>` tag that calls the `cachePut` function, passing in the string

'myPortfolio' as well as the query object called `myPortfolio`. The resulting code block should look similar to this:

```
<cfset myPortfolio = cacheGet('myPortfolio') />
<cfif isNull(myPortfolio)>
     <cfquery name="myPortfolio">
          SELECT
               id,
               title,
               summary,
               website,
               image
          FROM
               portfolio
     </cfquery>
     <cfset cachePut('myPortfolio',myportFolio) />
</cfif>
```

27. The `myPortfolio` query will now get cached if it is not already in cache. Run the page a few times and compare the execution times. If you want to clear the cache, reload the `/www/clearCache.cfm` page.

12

Security

By David Epler

About David Epler

David Epler is a Software Architect with AboutWeb in Rockville, MD. As a member of AboutWeb's solutions team, he has built, deployed, and maintained systems compliant with the most demanding regulations and mandates needed to pass security certification and accreditation for Federal Government clients. He has been developing with ColdFusion since version 4 and is an active member of the ColdFusion community. David has contributed to several open source ColdFusion projects and frameworks, along with the blog he maintains (www.dcepler.net). He was responsible for creating and maintaining Unofficial Updater 2 which makes patching ColdFusion 8 and 9 significantly easier before the Hotfix installer was introduced in ColdFusion 10. David has been at speaker at various user groups and conferences like CFUnited, RIACon, and Adobe Government Technology Summit He also helps run the Capital Area Cyber Security User Group in the DC Area.

Introduction

Security is a broad topic area and the threats are constantly evolving. Security encompasses more than just writing secure code, but also items like the configuration and setup of the servers and network, and practices and procedures for handling sensitive data.

In this chapter, we'll focus on areas of security that you have the most control over as a ColdFusion developer in order to help you write more secure ColdFusion code and understand the security settings in the ColdFusion Administrator, making it more difficult for an attacker to exploit your web application. We say "more difficult" because no web application can be 100% secure.

There are several shifts in thought required.

1. The web browser, be it desktop or mobile, is not the client interface to your web application; anything that can communicate with the HTTP protocol is the client interface to your web application. This includes telnet, wget, web application testing tools, or web attack tools.
2. Do not trust any data that comes from the client. This is more than just the URL parameters and the form post data; it also includes cookies and CGI variables. Anything that originates from the client can be manipulated by an attacker.
3. Validate all input on the server side. Client side validation is nice for the user experience, but can be circumvented.

Black List Versus White List

Black lists are lists of known "bad" patterns. Since there are always new attacks, black lists will always keep growing and are only as good as the known pattern it can detect. ColdFusion 7 introduced a feature called Script Protect that provided minimal Cross-site Scripting (XSS) prevention and is an example of a black list. It could block input that included `<script>` tags while allowing `<iframe>` because it was not included in the pattern to look for.

White lists, on the other hand, are a list of "good" values or patterns that the web application will accept for a given input, and all others are rejected. White lists have the advantage in that they are a finite set and can be used as part of the server side validation of data. An example of a white list would be a list of US States abbreviations used to check the value of the State form field on a form.

Using white lists is the preferred approach and will be shown throughout this chapter in the examples.

Injection

Injection attacks occur when data is sent to an interpreter which contain unintended commands with the data that are run by the interpreter. The most common injection flaw in web applications are SQL, but it is also possible to have injection flaws effect LDAP queries, XPath queries, and OS commands. We are going to cover SQL injections, but the techniques used to validate and control the input to the SQL interpreter are applicable to the other types of injections.

SQL Injection (SQLi)

In the earlier Database chapter you saw the use of the `cfqueryparam` tag. It is one of the simplest steps you can take to help prevent SQL injection attacks on your web application, but it can only be used in the WHERE clause, INSERT values, and UPDATE values of an SQL statement. Other parts of an SQL statement require more work to protect against it. The example below is using `cfqueryparam`, but it is still susceptible to SQL injection attack through the ORDER BY clause.

```
<cfparam name="url.state" default="" />
<cfparam name="url.orderby" default="LASTNAME" />


<cfquery name="request.listing" datasource="cfartgallery">
    SELECT      ARTISTID, FIRSTNAME, LASTNAME, EMAIL, THEPASSWORD,
ADDRESS, CITY, STATE, POSTALCODE, PHONE, FAX
    FROM        ARTISTS
    WHERE       1=1
    <cfif Len(url.state)>
    AND         STATE = <cfqueryparam cfsqltype="cf_sql_varchar"
value="#url.state#" />
    </cfif>
    ORDER BY    #url.orderby#
</cfquery>
```

By validating the URL parameters against a list of values we know to be good while changing all references from the URL scoped variables to variables we explicitly set, the SQL injection vulnerability is removed. The `cfparam`'s were also removed because they are now redundant from the use of `StructKeyExists()` in the `if` statements.

```
<cfscript>
    // list of valid values
    variables.validStates = "CA,CO,DC,FL,GA,NM,NY,OK,SD";
    variables.validOrderBys = "LASTNAME,CITY,STATE";


    // check what was passed against list of valid values
    if ( StructKeyExists(url, "state") AND ListFind(variables.
validStates, url.state) ) {
        variables.state = url.state;
    } else {
        variables.state = "";
    }


    if ( StructKeyExists(url, "orderby") AND ListFind(variables.
validOrderBys, url.orderby) ) {
        variables.orderby = url.orderby;
    } else {
        variables.orderby = "LASTNAME";
    }
</cfscript>


<cfquery name="request.listing" datasource="cfartgallery">
    SELECT      ARTISTID, FIRSTNAME, LASTNAME, EMAIL, THEPASSWORD,
ADDRESS, CITY, STATE, POSTALCODE, PHONE, FAX
    FROM        ARTISTS
    WHERE       1=1
    <cfif Len(variables.state)>
    AND         STATE = <cfqueryparam cfsqltype="cf_sql_varchar"
value="#variables.state#" />
    </cfif>
    ORDER BY    #variables.orderby#
</cfquery>
```

While the above code removes the SQL injection, it can be made better. First, remove the hard coded list of States and reuse the query used to generate the list of States for the filter in the validation. Next, change the 'Order By' values from columns used in the database to a structure that references them. While this example has some pretty common and guessable columns names, the main purpose is to make it more difficult for an attacker to get knowledge of the database structure the web application is using.

```
<cfscript>
    // valid values
    variables.validStates = ValueList(request.uniqueStates.STATE);
    variables.validOrderBys = {'name': {column: 'LASTNAME', display:
'Name'}
                              ,'city': {column: 'CITY', display: 'City'}
                              ,'state': {column: 'STATE', display:
'State'}};


    // check what was passed against valid values
    if ( StructKeyExists(url, "state") AND ListFind(variables.
validStates, url.state) ) {
        variables.state = url.state;
    } else {
        variables.state = "";
    }


    if ( StructKeyExists(url, "orderby") AND StructKeyExists(variables.
validOrderBys, url.orderby) ) {
        variables.orderby = url.orderby;
    } else {
        variables.orderby = "name";
    }
</cfscript>
```

```
<cfquery name="request.listing" datasource="cfartgallery">
    SELECT      ARTISTID, FIRSTNAME, LASTNAME, EMAIL, THEPASSWORD,
ADDRESS, CITY, STATE, POSTALCODE, PHONE, FAX
    FROM        ARTISTS
    WHERE       1=1
    <cfif Len(variables.state)>
    AND         STATE = <cfqueryparam cfsqltype="cf_sql_varchar"
value="#variables.state#" />
    </cfif>
    ORDER BY    #variables.validOrderBys[variables.orderby]["column"]#
</cfquery>
```

The technique shown above works in other parts of SQL statements as well.

Stored Procedures

It is generally noted that using stored procedures also resolves SQL injection, but that is not always the case. It is still possible to have a SQL injection occur inside the stored procedure depending upon how it is written, particularly if there is string concatenation using the input variables. It is recommended to use `cfstoredproc`, `cfprocparam`, and `cfprocresult` instead of calling a stored procedure through SQL.

```
<!--- bad --->
<cfquery name="request.SalesReport" datasource="Sales">
exec sp_SalesReport '#url.division#', '#url.startDate#', '#url.endDate#'
</cfquery>
<!--- better --->
<!--- similar validation of parameters done before execution --->
<cfstoredproc procedure="sp_SalesReport" datasource="Sales">
    <cfprocparam type="in" cfsqltype="cf_sql_varchar" value="#variables.
division#" />
    <cfprocparam type="in" cfsqltype="cf_sql_date" value="#variables.
startDate#" />
    <cfprocparam type="in" cfsqltype="cf_sql_date" value="#variables.
endDate#" />
    <cfprocresult name="request.SalesReport" result="1" />
</cfstoredproc>
```

ORM

ORM makes it easier to interact with the database without having to deal with SQL; however, it is still possible for an SQL injection to occur if you do not validate values as was done when working with `cfquery` and `cfstoredproc`. When using `ormExecuteQuery()`, use the `params` attribute, which will bind the value like `cfqueryparam` does for `cfquery`.

```
<!--- bad--->
<cfset artists = ormExecuteQuery( "FROM artist WHERE firstname like
'#url.firstName#%' ORDER BY lastname") />


<!--- better --->
<!--- similar validation of parameters done before execution --->
<cfset artists = ormExecuteQuery( "FROM artist WHERE firstname like
:firstname ORDER BY lastname", { firstname: "#variables.firstName#%"}) />
```

Additional Resources:

- OWASP SQL Injection Prevention Cheat Sheet

Cross-site Scripting (XSS)

Cross-site Scripting (XSS) is the most prevalent web application security flaw and occurs when user supplied data is sent to the browser without properly validating or escaping that content. XSS flaws can allow the attacker to:

- Deface web page (examples http://xssed.com, http://www.alpacahack.com/)
- Steal session cookies so attackers can impersonate victims without having to steal passwords
- Spoof login prompts to steal passwords
- Keystroke logging
- Redirect user to malware site
- Total takeover of end user through tools like BeEF (Browser Exploitation Framework)

There are three types of XSS:

- **Reflected**
  Reflected attacks are where the injected code is sent as part of the request and shown in the response. Common locations for reflected XSS are in error messages or search results. Reflected attacks require getting the user to click on the specially crafted URL or form with the injection. They are usually embedded in phishing emails or hidden through URL shorteners. Reflected XSS is also referred to as first order XSS.
- **Stored**
  Stored attacks are where the injected code is permanently stored in the web application. Common locations for stored XSS are in message forums, blog comments, or comment fields. The stored attack is sent to the user when they access the information. Stored XSS is also referred to as Persistent or second order XSS.
- **DOM Based**
  As the name suggests, the DOM based attack directly manipulates the browser through the DOM. DOM based attacks are different in that the response from the server is not manipulated, but the client side scripting is manipulated to modify how it runs.

Dealing with XSS requires properly decoding the input, validating the input, and then encoding the input for the context it will be used.

How Many Ways to Encode < as an HTML Entity?

So how many ways can you think of? Probably, < maybe <. There are actually 68 variations when using UTF-8. The problem only gets worse when attackers encode a string multiple times and mix the encodings to bypass validation filters. Below are some examples of how < can be encoded using different techniques:

- **&lt&#59** - double encoding
- **%26lt%3b** - double encoding with multiple schemes
- **%252525253C** - multiple encoding
- **&%6ct;** - nested encoding with multiple schemes

It becomes very clear that trying to create black lists of patterns is futile when `<script>` can be encoded 1,677,721,600,000,000 ways. ColdFusion 10 introduced a new function, `Canonicalize()`, which reduces an encoded string down to its simplest form. Canonicalize takes three arguments; the first is the string to be processed, the next is a boolean to indicate whether multiple encodings should be restricted, and the third is another boolean to indicate whether mixed encodings should be restricted. If either of the boolean flags are set to true, ColdFusion will throw an error if the string contains a given restriction. Input that is encoded multiple times and/or contains mixed encodings is something a normal user would not do and should be regarded as an attack on the web application.

Below is an example function that will canonicalize all the keys of a structure, blocking any input that contains multiple or mixed encodings and logging them.

```
<cffunction name="decodeScope" access="public" returntype="void"
output="false">
    <cfargument name="scope" type="struct" required="true" />


    <cfset var key = "" />


    <cfloop collection="#arguments.scope#" item="key">
        <cfif IsSimpleValue(arguments.scope[key])>
            <cftry>
                <!--- do not allow multiple and mixed encodings, most
likely an attack --->
                <cfset arguments.scope[key] = canonicalize(arguments.
scope[key], true, true) />


            <cfcatch type="any">
                <!--- if logging actual string, encoded it properly for
context.
                      Truncating cfcatch.logmessage so value is not
included --->
                <cflog file="encodingErrors" text="#key# -
#Left(cfcatch.LogMessage, Find(' in', cfcatch.LogMessage))#"
type="error" application="true" />
                <cfset arguments.scope[key] = "" />
            </cfcatch>
```

```
            </cftry>
        </cfif>
    </cfloop>
</cffunction>

<cfset decodeScope(url) />
<cfset decodeScope(form) />
<cfset decodeScope(cookie) />
<cfset decodeScope(cgi) />
```

Canonicalize makes it easier to do validation now that the input has the encoding simplified. Do not output any values that have been canonicalized without properly encoding them for context which they will be used.

Validation

Now that the input has been canonicalized, it needs to be validated. As stated before, the best approach is for a white list, where you only allow good known patterns of the input through; anything that does not match the pattern is rejected. ColdFusion's IsValid() is the best choice since it has many predefined types and the ability to use regular expressions. The input should be checked for length and if it is in a valid range in the case of drop-downs, radio buttons, and checkboxes using values from the database.

```
<cfscript>
    // IsValid using pre-defined type for US Zip Code
    if (NOT IsValid("zipcode", form.zipcode)) {
        variables.error += "Bad Zip Code<br />";
    }

    // IsValid using regex
    if (NOT IsValid("regex", form.zipcode, "^\d{5}(-\d{4})?$")) {
        variables.error += "Bad Zip Code<br />";
    }
```

```
    // Another IsValid using regex limiting characters (only
alphanumeric, space, ., -, ', and ,)
    if (NOT IsValid("regex", form.address, "^[a-zA-Z0-9 \.\-,']+$")) {
        variables.error += "Bad Address<br />";
    }
</cfscript>
```

There are also many validation functions available through CFLib. Validation should always take place on the server side. Client side validation should not be relied upon since it can be circumvented.

What About Rich Text Editors?

Inevitably, there will be a requirement for the web application to accept rich text formatting in `textarea`, which involves accepting HTML markup/ this can be dangerous because it is harder to filter XSS. If possible, only accept rich text from users that are logged in to the web application. Filtering HTML used for rich text is difficult, and it is recommend to use OWASP AntiSamy instead of trying to create your own filters. There are several blog postings on how to integrate AntiSamy with ColdFusion; the best one is by Pete Freitag, Using AntiSamy with ColdFusion.

Another approach would be to use something other than HTML to markup the content. There are several lightweight markup languages such as Markdown, BBCode, or various wiki text formats which could be used. It is a little more complex to implement, but since the web applications are directly controlling the formatting and rendering of the markup, it becomes significantly harder to introduce XSS to the content.

Proper Encoding for Output

Now that the input has been canonicalized and validated, is it safe to use it? Actually, the answer is no; it is still possible that the validation let something through, as attackers are always finding new ways to create XSS attacks. Any input that is used after it has been canonicalized and validated needs to be encoded for the context it will be used in to ensure that it is not interpreted to be anything other than text. In previous versions of ColdFusion, one would use `HTMLEditFormat()`, `XMLFormat()`, or `URLEncodedFormat()`, but they do not properly encode everything correctly if you are using the input in Javascript or CSS. ColdFusion 10 introduced several functions using the OWASP ESAPI library which addressed these shortcomings.

Context	Function	Example
HTML	EncodeForHTML()	`<p>Hello #EncodeForHTML(url.name)#</p>`
HTML Attribute	EncodeForHTMLAttribute()	`<div id="#EncodeForHTMLAttribute(url.name)#">`
Javascript	EncodeForJavascript()	`<a onclick="welcome('#EncodeForJavascript(url.name)#');" />` `<script>var name = #EncodeForJavascript(url.name)#;</script>`
CSS	EncodeForCSS()	`<div style="background-color: #EncodeForCSS(url.color)#;"></div>`
URL	EncodeForURL()	`<a href="http://www.example.com/edit.cfm?id=#EncodeForURL(url.id)#">Edit</a>`
XML	EncodeForXML()	<userinfo><name>#EncodeForXML(url.name)#</name></userinfo>

Unfortunately, Adobe did not implement all the encoders that are available in ESAPI, which would be particularly useful for `EncodeForXMLAttribute`, `EncodeForLDAP`, `EncodeForXPath`.

Because the older functions like `HTMLEditFormat`, `URLDecode`, `URLEncodedFormat`, and `XMLFormat` do not encode and decode as well as the new functions based upon ESAPI, Adobe is likely to deprecate them in favor of the newer functions.

```
<!--- Using OWASP ZAP 1.4.0.1 fuzzing with jbrofuzz/XSS - all selected
(223 patterns) and URL encoded --->
<cfif NOT StructIsEmpty(form)>
    <cfoutput>
    <dl>
        <dt>Raw:</dt>
        <dd>#form.myvalue#</dd>
        <!--- blocked 42 patterns with Script Protect ALL --->

        <dt>HTMLEditFormat:</dt>
        <dd>#HTMLEditFormat(form.myvalue)#</dd>
        <!--- blocked 204 patterns --->

        <dt>EncodeForHTML:</dt>
```

```
        <dd>#EncodeForHTML(form.myvalue)#</dd>
        <!--- blocked all 223 patterns --->
    </dl>
    </cfoutput>
</cfif>

<form name="myForm" id="myForm" method="post">
    <input name="myvalue" type="text" value="" />
    <input type="submit" />
</form>
```

Older Versions of ColdFusion

As seen above, older functions that were relied upon to properly encode and decode data do not work as well as the new ESAPI based functions in ColdFusion 10. If ColdFusion 8 or 9 has been patched with APSB11-04 or higher, the ESAPI Java library can be used by calling the Java library. There is another excellent blog post by Pete Freitag, ColdFusion's Built in Enterprise Security API; here, he addresses the method that CFBackport uses and provides the same encode functionality for ColdFusion 8 and 9 along with other functions introduced in ColdFusion 10. Another alternative would be to use CFESAPI, which is a full implementation of ESAPI in ColdFusion.

Additional Resources:

- OWASP XSS (Cross Site Scripting) Prevention Cheat Sheet
- OWASP DOM based XSS Prevention Cheat Sheet
- Don’t Write Your Own Security Code – The Enterprise Security API Project (PDF)

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an attack which forces an end user to execute unwanted actions on a web application in which he/she is currently authenticated. This occurs because web browsers automatically include most credentials with each request, such as session cookies, basic authentication header, IP address, and client side SSL certificates.

One of the many examples occurred with Netflix in 2006; if you used the "Remember Me" functionality and came across any web page that had `<img src="http://www.netflix.com/AddToQueue?movieid=70011204" />`, embedded in it, a copy of "SpongeBob Squarepants" would be added to your Netflix queue.

ColdFusion 10 introduced two new functions to deal with CSRF; `CSRFGenerateToken()` and `CSRFVerifyToken()`. To use the functions, the web application needs to have Session Management enabled, which works by creating a random token that can be checked when the submission occurs.

```
<cfif NOT StructIsEmpty(form) >

    <cfif NOT CSRFverifyToken(form.token)>
        <cfabort showerror="Invalid Token" />
    </cfif>

    <cfoutput><p>Hello, #EncodeForHTML(form.name)#</p></cfoutput>
</cfif>

<cfoutput>
<form method="post" name="csrfexample">
    <input name="token" type="hidden" value="#CSRFGenerateToken()#" />
    <input name="name" type="text">

    <input name="submit" type="submit" value="Submit">
</form>
</cfoutput>
```

If there is a Cross-Site Scripting (XSS) vulnerability in the web application, it is not possible to prevent CSRF since the XSS vulnerability will allow the attacker to grab the token and include the token with a forged request.

If the action that needs to be done is sensitive, another approach to prevent CSRF is to force the user to re-authenticate with their password before allowing the action to proceed.

Additional Resources:

- OWASP Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet

Session Identifier Protection

Session identifiers (CFID, CFTOKEN, and JSESSIONID) need to be protected since they provide an attacker with an easy way to impersonate a user if they are acquired.

One of the biggest leakages of session identifiers occurs with `cflocation` because the `ADDTOKEN` attribute defaults to true, appending the session identifiers to the URL, and thus be easily captured. The `ADDTOKEN` attribute should be set to false in almost all cases.

An additional measure to protect the session identifiers is to set the cookie which they are delivered to be `HTTPOnly`. When a cookie is flagged `HTTPOnly`, it is not possible for the cookie to be accessed in the browser via Javascript. ColdFusion 9 added the ability to set `HTTPOnly` cookies with `cfcookie` and ColdFusion 9.0.1 added a JVM flag to enable `HTTPOnly` cookies for the session identifiers. Pete Freitag has an excellent blog posting showing how to deal with this in older versions of ColdFusion: Setting up HTTPOnly Session Cookies for ColdFusion.

ColdFusion 10 added several new configuration items for working with the session cookie and authorization cookie for `cflogin`. They can be configured globally through the ColdFusion Administrator or on a per-application basis by applying the settings in `Application.cfc`.

```
<cfscript>
    this.sessioncookie.httponly = true;     // if cookie to be httponly
    this.sessioncookie.timeout = "10";      // session cookie age (in
days)
    this.sessioncookie.secure = true;       // only https session cookie
    this.sessioncookie.domain = ".example.com"; // domain for which
session cookies are valid
    this.sessioncookie.disableupdate = true;    // if session cookie can
be modified by coldfusion tags (cfcookie, cfheader)


    // CFLogin settings
    this.authcookie.timeout = createTimeSpan(0, 0, 0, 20);  //
authentication cookie age
    this.authcookie.disableupdate = true;                   // if
session cookie can be modified by coldfusion tags (cfcookie, cfheader)
</cfscript>
```

ColdFusion 10 also introduced two new functions to work with sessions, `SessionInvalidate()` and `SessionRotate()`. `SessionInvalidate` will clear all data stored in the session. If you are using J2EE sessions it will not invalidate the underlying `JSESSIONID`. `SessionRotate` is useful to prevent session fixation issues when used after the user is successfully authenticated.

`SessionRotate` will copy the current session data, invalidate the current session, create a new session, and finally populate the new session with the copied data.

Additional Resources:

- OWASP Session Management Cheat Sheet

File Uploads

Accepting file uploads is another common requirement for web applications, but also pose a great risk to both the server and the users of the web application. If not handled correctly, an uploaded file can lead to a compromised server or spread a virus infected file to other users.

The default behavior of the file upload should be to delete the file if it does not pass a validation check. When the file has passed all the checks, move it to the proper location using a system generated file name.

The first and most important thing is that files should **NEVER** be uploaded to a web accessible directory. They should always be placed in a temporary location, generally the ColdFusion temporary directory from `GetTempDirectory()`. On UNIX systems should also restrict access to the uploaded file by specifying the `mode` attribute, preferably 600 so that only the ColdFusion process can read or write to the file.

The types of files accepted in the upload should always be limited through the `ACCEPT` attribute and not allow all file types. There were several changes to cffile `action="upload"` in ColdFusion 10 on how it handles what file types are allowed. In previous versions, the `ACCEPT` attribute only allowed for a list of mime types like "image/jpeg,image/png,application/pdf"; in ColdFusion 10 you can use a list of file extensions like "*.jpg,*.png,*.pdf". You should not mix the two in the attribute; use one or the other.

Also new in ColdFusion 10 is the `strict` attribute which defaults to true. In previous versions of ColdFusion, the mime type (content-type and content-subtype) were based upon what the client told ColdFusion the file is, not the actual contents. It was possible to check specific file types by using `IsPDFFile()`, `IsImageFile()`, or `IsSpreadsheet()`, but that left out a lot of other valid files that could not be checked. With strict set to true, the mime type of the file is checked when the file upload occurs; however, this means that ACCEPT must be a list of mime types and not file extensions. If ACCEPT has a list of file extensions and strict is set to true, ColdFusion will throw an error.

ColdFusion 10 introduced a new function, `FileGetMimeType()`, which can now return the mime type for any file.

```
<cfscript>
    variables.validMimeTypes =  {'application/pdf': {extension: 'pdf',
application: 'Adobe Acrobat'}
                                ,'application/vnd.ms-powerpoint':
{extension: 'ppt', application: 'PowerPoint (97-2003)'}
                                ,'application/vnd.openxmlformats-
officedocument.presentationml.presentation': {extension: 'pptx',
application: 'PowerPoint (2007)'}
                                ,'image/jpeg': {extension: 'jpg'}
                                ,'image/png': {extension: 'png'}};
```

```
</cfscript>
<cftry>

    <cffile action="upload" filefield="uploadFile"
            destination="#GetTempDirectory()#" mode="600"
            accept="#StructKeyList(variables.validMimeTypes)#"
            strict="true"
            result="request.uploadResult"
            nameconflict="makeunique" />


<cfcatch type="any">
    <!--- file is not written to disk if error is thrown  --->
    <!--- prevent zero length files --->
    <cfif FindNoCase("No data was received in the uploaded", cfcatch.
message)>
        <cfabort showerror="Zero length file" />


    <!--- prevent invalid file types --->
    <cfelseif FindNoCase("The MIME type or the Extension of the uploaded
file", cfcatch.message)>
        <cfabort showerror="Invalid file type" />


    <!--- prevent empty form field --->
    <cfelseif FindNoCase("did not contain a file.", cfcatch.message)>
        <cfabort showerror="Empty form field" />


    <!---all other errors --->
    <cfelse>
        <cfabort showerror="Unhandled File Upload Error" />


    </cfif>
```

```
</cfcatch>
</cftry>


<!--- get actual mime type --->
<cfset variables.actualMimeType = FileGetMimeType(request.uploadResult.
ServerDirectory & '/' & request.uploadResult.ServerFile, true) />


<!--- redundant check with strict="true", does not hurt to double check
Adobe --->
<cfif NOT StructKeyExists(variables.validMimeTypes, variables.
actualMimeType)>
    <cffile action="delete" file="#request.uploadResult.
ServerDirectory#/#request.uploadResult.ServerFile#"  />
    <cfabort showerror="Invalid file type (checked)" />
</cfif>


<!--- generate unique filename for move to destination, DO NOT reuse
filename sent by user --->
<cfset request.uploadFile.destination = ExpandPath("/") & "uploads/" &
CreateUUID() & "." & variables.validMimeTypes[variables.actualMimeType]
["extension"] />


<cffile action="move"
        source="#request.uploadResult.ServerDirectory#/#request.
uploadResult.ServerFile#"
        destination="#request.uploadFile.destination#" mode="644" />
```

Additional Measures

If possible limit file uploads to only authenticated users of the web application.

What is not shown through the code sample above is processing the upload through any type of virus scanner or any additional file size checks that could be done beyond the post limit size set in ColdFusion Administrator or through the web server configuration.

Also, additional restrictions could be put in place using OS level permissions and/or using Sandboxing within ColdFusion to limit where ColdFusion can read and write the file.

Secure Password Storage

The ARTISTS table in the `cfartgallery` datasource used for examples is an excellent example of how **NOT** to store passwords. First, they are stored in clear text and the column is limited to only 8 characters.

ARTISTID	EMAIL	THEPASSWORD
4	user@demodata.com	demo
10	diane@demo.com	demo
15	me@m.com	admin

The next step most people take is to hash the password and store the hash in the database (assuming we increase the length of THEPASSWORD to accommodate it). Hashing provides a one way encoding of a string to a fixed length string. Below shows what the table would look like if THEPASSWORD for each user was hashed using MD5.

```
<cfset variables.hashedPassword = Hash(form.password) />
```

ARTISTID	EMAIL	THEPASSWORD
4	user@demodata.com	FE01CE2A7FBAC8FAFAED7C982A04E229
10	diane@demo.com	FE01CE2A7FBAC8FAFAED7C982A04E229
15	me@m.com	21232F297A57A5A743894A0E4A801FC3

What you'll notice is that the hashed password for both `user@demodata.com` and `diane@demo.com` are still the same. This makes it easier for the attacker to compromise multiple accounts because the input password is the same for both accounts. To make the hashes unique for each user, we need to generate a string to be appended to the password, or a salt. The salt needs to be unique and randomly generated for each user. Both the salt and the hashed password need to be stored in the database. Also, change the hashing algorithm to SHA-512, because both MD5 and SHA-1 are vulnerable to collision attacks.

```
<cfset variables.salt = Hash(GenerateSecretKey("AES"), "SHA-512") />
<!--- could use Rand("SHA1PRNG") instead of GenerateSecertKey() --->


<cfset variables.hashedPassword = Hash(form.password & variables.salt,
"SHA-512") />
<!--- insert both variables.salt and variables.hashedPassword into table
--->
```

ARTISTID	EMAIL	THEPASSWORD	THESALT
4	user@demodata.com	457AEB091B4F87EB5D 98D9972FB707DBFDC2 E687B5C8ADA550C997 1B89F3BB3C6F863668 67C5F1F8A7935D9DC6 B3BFC4801D97CD9D78 C368DA0022E2022527A5	1BF869C286A1D2A4B0 D704C620C365BA60B1 B2457200D5918B1F50 F7E27806013963E824 D7ADB140994A56298B 0F20B6246768612418 91660224AF2AA91D8 CCE
10	diane@demo.com	2F275B973C5D617F56A 8957F266AE38A20F0B3 3DC88143D44F9110480 000D39E78297A6BFAA7 39C937379FBA3397DF4 47142FBC647C62289EC A5195544656354	0EDCEE1F72ABD928B1 934EDF6ED0FF963532 D023E6E0F2BFD705138 44489D063AE36CF4F31 44117946959075CA49F 8E5EAC475A0BE324266 626BDC06CFEA2837
15	me@m.com	ED0148FBC717B9F1217 7BCA92C390A142264D 0729B7657416E557F9E 0BA064893810C474573 5F88DB85902C2AFD09 C40841035C101750E5 70685F7E843F718F2	C8FE4F050FFE75F6FA 023FB34105D766BE43 32EEFA8925676C8803F DB849FFDFA1CEFF4E1 407030065DAA55A2592 829B5932C7C97292D49 FEAEAA282F5CEC9DD

As you can see, THEPASSWORD for all the users is unique because they have been salted.

ColdFusion 10 enhanced `Hash()` by adding an additional parameter to provide the number of iterations the Hash should be run.

```
<cfset variables.numIterarions = 1000 />


<cfquery name="request.getPwdAndSalt" datasource="cfartgallery">
SELECT  THEPASSWORD, SALT
FROM    ARTISTS
WHERE   EMAIL = <cfqueryparam cfsqltype="cf_sql_varchar" value=
"#form.user#" maxlength="50" />
</cfquery>


<cfif request.getPwdAndSalt.RecordCount EQ 1>
    <cfif request.getPwdAndSalt.THEPASSWORD EQ Hash(form.password
```

```
& request.getPwdAndSalt.SALT, "SHA-512", "utf-8", variables.
numIterarions)>
        <cfset SessionRotate() />
        <!--- Password is good --->
    <cfelse>
        <!--- Bad Password --->
    </cfif>
<cfelse>
    <!--- Bad User --->
</cfif>
```

Additional Resources:

- OWASP Password Storage Cheat Sheet
- Using bCrypt in ColdFusion 10
- OWASP Forgot Password Cheat Sheet
- Why passwords have never been weaker—and crackers have never been stronger

ColdFusion Configuration

The previous sections focused on secure ColdFusion coding practices, but if the installation of ColdFusion is configured insecurely, all of that work is for nothing. Securing the ColdFusion Application Server requires making the OS, web server, and the ColdFusion configuration secure. For ColdFusion 9, Adobe published the Adobe ColdFusion 9 Server Lockdown Guide which covered how to properly configure ColdFusion 9 on Windows or Linux. It covered various OS settings, connecting ColdFusion to IIS or Apache, and settings within the ColdFusion Administrator.

Secure Profile

ColdFusion 10 introduced the Secure Profile that can be enabled during installation. It simplifies securing ColdFusion by applying more secure defaults to the configuration instead of having to apply them manually. Some of the settings changes are:

- Disables the RDS service
- Enables separate username and password for Administrator and RDS
- Restricts access to ColdFusion Administrator to a list of IP addresses
- Provides default missing template, site-wide error, and request queue timeout pages to reduce information leakage
- Disables all debugging

Adobe recommends using Secure Profile for production or public-facing servers.

Additional Resources:

- Administrator settings affected by enabling Secure Profile

Sandboxing

ColdFusion has had security sandboxing for quite a long time, but is probably an under utilized option that can help secure ColdFusion because it is only really useful in Enterprise. Sandboxing allows you to restrict access to data sources, ColdFusion tags/functions, directories, and servers/ports on a subdirectory. If a piece of ColdFusion code tries to access a restricted resource in the sandbox, ColdFusion will throw an error. The best use of sandboxing is to restrict everything on the webroot of a server and then only allow what is needed per the subdirectory (web application).

Additional Resources:

- Using sandbox security

More Resources

Websites

- Security improvements in ColdFusion 10
- The Open Web Application Security Project

Books

- The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, Second Edition by Dafydd Stuttard and Marcus Pinto
- SQL Injection Attacks and Defense, Second Edition by Justin Clarke
- XSS Attacks: Cross Site Scripting Exploits and Defense by Jeremiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D. Petkov and Anton Rager

Hands On 28

By Simon Free

In this hands on, we are going to add security to the blog comments. As this section allows users to supply data that will be stored in a database and will also be output to other users, this is a weakest point of the application.

Tags Used: <cfif>, <cfset>, <cfthrow>

Functions Used: isSimpleValue, canonicalize, CSRFGenerateToken, CSRFverifyToken, encodeForHTML

1. Open up the `/www/blogpost.cfm` file in your code editor.
2. The first thing we are going to do is add some checks to make sure the values in the form are all simple values. Locate the `<cfif>` statement that checks if the `form` has been submitted on or around line 3.
3. Inside the `<cfif>`, prior to creating a new `blogcomment` entity, create a new `if` statement that checks if the `form.author` variable is a simple value. Your code should look similar to this:

```
<cfif isSimpleValue(form.author)>


</cfif>
```

4. Once we know the value is a simple value, we need to call the `canonicalize()` method on the `form.author` value. To make things easier, assign the result of the `canonicalize()` call back to the `form.author` variable. The code will look similar to this:

```
<cfset form.author = canonicalize(form.author, true, true) />
```

5. Do the same for the remaining `comment` variable. Your final code should look similar to this:

```
<cfif isSimpleValue(form.author)>
    <cfset form.author = canonicalize(form.author, true, true) />
</cfif>
<cfif isSimpleValue(form.comment)>
    <cfset form.comment = canonicalize(form.comment, true, true) />
</cfif>
```

6. Next, check if any of the values were not simple values. If one was not a simple value, it will throw an error. Create a `<cfif>` statement that checks if either are not simple values, and if one isn't, throw an error with the message `Validation Error`. Your code should look similar to this:

```
<cfif !isSimpleValue(form.author) || !isSimpleValue(form.comment)>
    <cfthrow message="Validation Error" >
</cfif>
```

7. Next, we will utilize ColdFusion's CSRF support by generating and validating a CSRF token. Locate the hidden field in the comment form on or around line 99.
8. Create a new hidden field called `token`, and give it the value:

```
#CSRFGenerateToken()#
```

9. Go to the top of the page and create a new `<cfparam>` tag for the `form.token` variable and default it to empty.
10. Go back to the `<cfif>` statement on or around line 11 which checks if any of the `form` fields is not a simple value.
11. Inside the `<cfif>` tag, check if the token value passed is a valid CSRF token. If the token is not valid, the same error will be thrown by the validation. Your final code should look similar to this:

```
<cfif !isSimpleValue(form.author) || !isSimpleValue(form.comment) ||
!CSRFVerifyToken(form.token)>
    <cfthrow message="Validation Error" >
</cfif>
```

12. Now that all the data has been checked on input, we now need to validate the data on output. Locate where the comment body is output to the screen on or around line 75.
13. Wrap the `#comment.comment#` output in an `encodeForHTML()` call so that the line of code looks similar to this:

```
#encodeForHTML(comment.comment)#
```

14. Make the same update for the author name output on or around line 72.
15. Open up `/www/blog.cfm` in your browser and navigate to a blog post.

16. Confirm that the page loads successfully.
17. Post a new comment and confirm that it still saves and outputs to the screen. Your blog now has additional security! Remember, even though it has security, it still not be considered 100% secure.

Hands On 29

By Simon Free

In this hands on we are going to secure our session and improve our admin security by hashing our passwords.

Tags Used: <cfif>, <cfset>, <cfquery>, <cfqueryparam>

Functions Used: len, trim, Hash, GenerateSecretKey, SessionRotate, SessionInvalidate

1. Open up the `/www/Application.cfc` file in your code editor.
2. When cookies are enabled on a web site (which is the case for this web site), it is important to secure them. Locate the block of variable declarations that sets values of the `this` scope. They should be at the top of the file.
3. After the last variable declaration, add the following lines of code:

```
this.sessioncookie.httponly = true;
this.sessioncookie.timeout = "10";
this.sessioncookie.disableupdate = true;
```

4. Now that the session cookies are secured, improve the security of the admin area by updating the administrator passwords. Open up the `/www/admin/content/system/editadministrator.cfm` file in your code editor.
5. Locate the line of code that checks if there were any validation errors on or around line 38.
6. The first thing to do when implementing hashing into the admin is to check if a password has been submitted. If no password has been submitted when the record is being updated, then we do not plan on updating the password and do not need to hash anything. Create a `<cfif>` tag that checks if `form.password` has a length. Remember to trim the value just in case any errant spaces have been entered. Your code should look similar to this:

```
<cfif len(trim(form.password))>


</cfif>
```

7. Inside of the `<cfif>` tag, create two values: the first is the `salt` value we will be using and the other will be the hashed `password`. To create the salt value, create a `<cfset>` tag that creates a variable called `salt`. In the `<cfset>` tag call the `Hash()` function and pass in a call to the `GenerateSecretKey` function as the string to hash, and pass in `SHA-512` as the algorithm to use when hashing. In the `GenerateSecretKey` call, pass in `AES` as the algorithm to use. Your `<cfset>` should look similar to this:

```
<cfset salt = Hash(GenerateSecretKey("AES"), "SHA-512") />
```

8. Next, generate the hashed password. To do this, create another `<cfset>` tag that sets a variable called password. In the `<cfset>`, call the `Hash()` function again and pass it in a concatenated string of the `form.password` value and the `salt` value. Tell the `Hash()` function to use the `SHA-512` algorithm. Your `<cfset>` should look similar to this:

```
<cfset Password = Hash(form.password & salt, "SHA-512") />
```

9. Once there is a salt and a hashed password created, we need to update the queries to accept these values. Locate the `<cfquery>` tag that updates the administrator record on or around line 45. Update the SQL code so that the password value stored is the new password variable. Also, update the SQL so that the salt value is saved into a column called `salt`. The update query should look similar to this:

```
<cfquery>
    UPDATE
        administrator
    SET
        firstname = <cfqueryparam value="#trim(form.firstname)#"
cfsqltype="cf_sql_varchar" />,
        lastname = <cfqueryparam value="#trim(form.lastname)#"
cfsqltype="cf_sql_varchar" />,
        emailaddress = <cfqueryparam value="#trim(form.emailaddress)#"
cfsqltype="cf_sql_varchar" />
        <cfif len(trim(form.password))>
            ,password = <cfqueryparam value="#password#"
cfsqltype="cf_sql_varchar" />
            ,salt = <cfqueryparam value="#salt#" cfsqltype=
"cf_sql_varchar" />
        </cfif>
    WHERE
        id = <cfqueryparam value="#form.id#" cfsqltype="cf_sql_integer"
/>
</cfquery>
```

10. Once the update functionality is changed, it is time to change the insert functionality. Locate the `<cfquery>` that creates a new `administrator` and update the statement to use the new `password` variable. Also update the SQL to store the `salt` value. Your `<cfquery>` should look similar to this:

```
<cfquery>
    INSERT INTO
         administrator (
         firstname,
         lastname,
         emailaddress,
         password,
         salt
    ) VALUES (
         <cfqueryparam value="#trim(form.firstname)#" cfsqltype=
"cf_sql_varchar" />,
         <cfqueryparam value="#trim(form.lastname)#" cfsqltype=
"cf_sql_varchar" />,
         <cfqueryparam value="#trim(form.emailaddress)#" cfsqltype=
"cf_sql_varchar" />,
         <cfqueryparam value="#password#" cfsqltype="cf_sql_varchar" />,
         <cfqueryparam value="#salt#" cfsqltype="cf_sql_varchar" />
    )
</cfquery>
```

11. Now that the newly encrypted password can be saved, we need to update the login logic to accommodate the changes that have been made. Before we do this, update the password for the current administrator. If we do not, as soon as the new logic is in place you will no longer be able to login to the admin. Open up `/www/admin/` in your browser.
12. Login with the username: `system@yoursite.com` and password: `admin`.
13. Navigate to: System > List Administrators.
14. Edit the Administrator, provide a new password and click 'Save'. You have now updated your account with a newly hashed password.
15. Now that the new password is created, we can update the login logic.
16. Open up the `/www/admin/login.cfm` file in your code editor.

17. Update the query that checks the `username` and `password`. Because we need to hash the password supplied with the applicable salt value, we can no longer search on the `password` value. Locate the `<cfquery>` tag with the name of `qLoginCheck` on or around line 10.

18. Remove the part of the query that searches against the password and update the select list to return the `password` and `salt` columns.

19. Locate the `<cfif>` statement that checks if any records were returned from the query, on or around line 22.

20. Next, hash the password that was submitted by the form with the salt that was returned in the query. If that hashed value does not match what was returned by the query, then the password is incorrect. Update the `<cfif>` statement so it reads:

```
<cfif !qLoginCheck.recordcount || qLoginCheck.password NEQ
Hash(form.password & qLoginCheck.salt, "SHA-512")>
```

21. The logic is now in place to check the submitted password with the hashed password. Before testing it out to make sure everything is working, we are going to add 2 more function calls to protect our session identifiers. Locate the `<cflocation>` tag on or around line 27.

22. Before this tag make a call to `SessionRotate()` using a `<cfset>` tag. This will rotate the session identifiers every time we log in.

23. Next, make sure the session is completely removed when we log out; open up the `/www/admin/logout.cfm` file in your code editor.

24. Locate the `<cflocation>` tag on or around line 3.

25. Before this tag, make a call to `SessionInvalidate()` using a `<cfset>` tag.

26. Now that we have our hashed passwords set up and our session protection in place, it is time to test that everything is working. Log out by clicking on the `system@myWebsite.com` email address on the top right of the admin.

27. Fill in the log in details for the administrator account you edited.

28. Once logged in, click around the admin and then click log out. You have now successfully added security to our admin area.

13

Error Handling and Debugging

By Simon Free

About Simon Free

Simon Free has extensive experience in the online space. As an ACP, an ACI, and application developer for ten24, Simon has proven his skills in both the educational and development arenas. Constantly learning and sharing his knowledge, Simon is a trusted name in the conference community. His education is evident in his ability to produce high profile projects ranging from elaborate ecommerce applications to editorial CMS platforms for increasing the workflow within the publishing industry. Simon's technology background includes development in ColdFusion, jQuery, Flex, and AIR.

Error Handling

While the best efforts are made to stop errors from happening, they do happen. Sometimes the errors might be due to some bad code; other times it may be due to external resources that are out of one's control. During the development process, these errors hold valuable information that allows us to improve and fix our code, but in production environments, these errors hold information that can make our servers vulnerable to attack. It is important that as a developer you anticipate errors and gracefully handle them. Thankfully, ColdFusion offers a number of ways to trap those errors and even allow developers to react to those errors and call alternative functionality.

Understanding Errors

Error Types

Before we look at how to handle the errors, lets first take a look at the errors themselves. There are three types of ColdFusion Errors:

- **Exception** - Where the error stops the request from completing its process.
- **Missing Template** - When an HTTP request for a page can not be found.
- **Form Field Data Validation** - When server side form validation fail.

The most common error type you are going to experience is the Exception type. Unless the users are requesting a page that does not exist, or you have decided to use ColdFusions in-built form validation (not recommended by most ColdFusion developers), you will receive an Exception error.

These exception errors, excluding custom errors, fall into one of the following exception types:

- **Database** - When there is a problem with a database cal,l such as malformed SQL or database connection issue.
- **MissingInclude** - When an included file can not be found.
- **Template** - When a general error occurs, usually from a malformed tag or incorrect script syntax.
- **Object** - When an error occurs with an object.
- **Security** - When an error occurs related to security functionality.
- **Expression** - When an expression fails, such as 1 + "a".
- **Lock** - When an error occurs with a piece of code that has been locked by the application. This can be due to the code failing at runtime or a lock timing out.
- **SearchEngine** - When there is an issue with the Verity Search Engine.
- **Application** - When custom errors are generated by the cfthrow tag that do not have a type defined.

Knowing the type of exception thrown can be useful when handling your errors gracefully. Depending on the type of exception, it might be possible to retry a section of code again. For example, if a section of code that communicates with an external service times out, you might want to try again to see if the second time is successful. If you try this approach with some errors, it is important to keep in mind that your request might never be successful and that the error must then be handled a different way.

Error Data

For every error thrown, there are 2 standard error formats that contain relevant information to the error. The format of error you get depends on the settings within your ColdFusion Administrator. Under the Debugging & Output Settings there is an option called Debug Output Settings. On that page there is an option called Enable Robust Exception Information. Checking that will provide additional information to the page. These two formats are only relevant if the error is displayed to the user. Checking this box will not alter the information that is provided to the system if the error is caught programmatically. It is strongly advised to never enable the Robust Exception Information on a production server. If your error handling were to fail, this would display sensitive information to the user which you would not want them to see.

If the error is caught programmatically you will have access to the following information:

- **Message** - This provides you a brief, one line summary of the error.
- **Detail** - This provides additional information about the error along with suggested solutions, if there are any.
- **Stack Trace** - This shows the contents of the java stack at the time of the exception.
- **Tag Context** - This provides a list of all files that were called called and from what lines they were called.
- **Type** - The type of the exception.

Depending on the type of exception thrown, you might have access to additional pieces of information; however, these fields will always be present.

Error Logs

ColdFusion has extensive capabilities when it comes to error logs. Later in this chapter we will review how to create your own log entries, but for now, let's look at the automatic log entries.

Every time that the default error handler is used, which is when ColdFusion displays the error for you on the screen, an entry is entered into the ColdFusion Error Log. This Error Log is accessible via a log viewer application or via the ColdFusion Administrator. To access the Error Log, simply open up your ColdFusion Administrator and go to: Debugging & Logging > Log Files. From this screen you will see all the log files that ColdFusion creates. If you created any custom logging, you would also see the log files here. Notice that each log file has a number of different icons, allowing for different actions on the log. These actions include searching and viewing the log, archiving the log, and deleting the log. To view the errors on your application, you can select the `application.log` file or the `exception.log` file. Each log file gives a different type of information and can be used to find errors within your application.

Error Management

Now that there is an understanding of what errors are, let's take a look at Error Management. There are a number of ways in which you can capture an error and handle it with ease. When capturing the error, you might decide to run an alternate piece of code, but sometimes you might just want to notify the user that an error has occurred and send the information back to yourself for resolving later. Whatever way you wish to handle the error, you will need to use one of these forms of Error Handling.

cftry/ccatch

The use of the `cftry` and `cfcatch` tags allows you to provide error handling around a specific section of code. This method is often used when you wish to provide alternate processing of code. The `cftry` tag is wrapped around the section of code that you can monitor for issues. If anything within the `cftry` tag causes an error, the matching `cfcatch` tag will catch the error and allow you to provide alternate processing. Using this method will prevent the error from making its way to the user. Inside the `cftry` tag you can put any ColdFusion logic, including calling objects and including files. If any errors occur from within these external resources, the error will still be caught, assuming those external files do not have their own error handling in them.

At the end of the `cftry` block, but before the closing tag, you will place at least one `cfcatch` tag. The `cfcatch` tag is where you will place your alternate processing. The `cfcatch` tag must provide a `type` attribute which specifies which type of exception it will catch. Multiple `cfcatch` tags can be used within a `cftry` tag as long as they all have different types specified. Providing different `cfcatch` tags allows you to handle the different exception types differently. If you do not want to handle different exception types differently you can use the exception type of `any`, which will catch all exception types.

Here is an example of a cftry/cfcatch:

```
<cftry>
    <!--- Your code --->

    <cfcatch type="database" >
        <!--- Code to handle a database exception --->
    </cfcatch>
    <cfcatch type="any">
        <!--- Code to handle all other exceptions --->
    </cfcatch>
</cftry>
```

If you prefer your code to be script based, here is a script based example:

```
try{
    //Your code
    }
catch(database exception){
    //Code to handle database exception
```

```
}
catch(any exception){
    //code to handle all other exception
}
```

Once the `cftry` has run and any exceptions have been dealt with or ignored if you so choose, the page will continue to process from the closing `cftry` tag. If, once the exception has been caught, you do not wish to handle the exception, but want to pass it off to the next level of error handling, you can use the `cfrethrow` tag. The `cfrethrow` tag will then bubble the error up the chain of Error Management. The page will then no longer continue to process.

Here is an example of the cfrethrow in action:

```
<cftry>
    <!--- Your code --->

    <cfcatch type="database" >
        <!--- Code to handle a database exception --->
    </cfcatch>
    <cfcatch type="any">
        <cfrethrow />
    </cfcatch>
</cftry>
```

And in script format:

```
try{
    //Your code
    }
catch(database exception){
    //Code to handle database exception
}
catch(any exception){
    //code to handle all other exception
```

```
    rethrow;
}
```

cfthrow

Sometimes when developing an application, it may be necessary to create your own error to be thrown. Perhaps you are making an HTTP request and you did not receive the expected 200 response. In those situations you can use the `cfthrow` tag to throw your own exception. That exception will then get picked up by the first level of your Error Management solution.

When using the `cfthrow` tag, you have a number of useful attributes at your disposal. The most useful attributes are the `type` and `message` attributes. The `type` attribute allows you to specify the type of error being thrown. This type does not need to be one of ColdFusion's predefined exception types and can be a custom type of your own choosing. This can be very useful when used within a `cftry/cfcatch` block, as your `cfcatch` type can also match that custom type. The `message` attribute is also very useful; it allows you to provide a message, or reason, for the error. This message could then be relayed back to you via a global part of your Error Management solution.

Here is an example of a `cfthrow` inside of a `cftry/cfcatch`:

```
<cftry>
    <!--- Your code --->

    <cfcatch type="database" >
        <!--- Code to handle a database exception --->
    </cfcatch>
    <cfcatch type="any">
        <cfthrow
            message="I am an Error"
            type="specialError"
            detail="This is where I put extra detail"
            />
    </cfcatch>
</cftry>
```

And in script format:

```
try{
    //Your code
    }
catch(database exception){
    //Code to handle database exception
}
catch(any exception){
    //code to handle all other exception
    throw(message='I am an error', type="specialError", detail=
"This is where I put extra detail");
}
```

cffinally

The `cffinally` tag is placed inside the `cftry` block of code after the `cfcatch` blocks. The `cffinally` tag will always execute, even if no errors occur. This tag can be useful when there is some functionality you always want to run, such as functionality that will free up resources.

Here is an example of `cffinally` in action:

```
<cftry>
    <!--- Your code --->

    <cfcatch type="database" >
        <!--- Code to handle a database exception --->
    </cfcatch>
    <cfcatch type="any">
        <!--- Code to handle all other exceptions --->
    </cfcatch>
    <cffinally>
        <!--- This code will always run --->
    </cffinally>
</cftry>
```

And in script format:

```
try{
    //Your code
    }
catch(database exception){
    //Code to handle database exception
}
catch(any exception){
    //code to handle all other exception
}
finally{
    //This code will always run
}
```

onMissingTemplate

Within the `Application.cfc` file you can specify a function called `onMissingTemplate`. This method will be called when a requested page does not exist. If this method is called, the standard `onApplicationStart` and `onRequestStart` methods are not called. If the `onMissingTemplate` function returns 'false', then the control is passed back to the servers 404 handler.

When the `onMissingTemplate` function is called, it is up to you how you handle the issue. The most common thing to do is to include a site specific custom 404 page and notify the user that the page is missing. As the method receives the path of the file that was requested, you also have the ability to perform an action based on that information. For example, if you have a file that is often mistyped, you could look at the provided information, decide what page they really wanted, and redirect them to that page.

onError

If you wanted to catch all errors within a specific site, you can use the `onError` method inside of the `Application.cfc`. The `onError` method will be fired by any error that occurs in your site that is not caught inside of a `cftry/cfcatch` block. The `onError` method will catch all exception types, although it will not catch exceptions thrown due to syntax errors such as malformed tags. Those errors will bubble up to the next level of Error Management.

The `onError` method is the most common Error Management method. Using this method allows you to track all errors from your site and handle them accordingly. The most common method for handling these errors is to display a 'Sorry' page to your users and notifying one of your development staff of the information.

The `onError` method receives 2 arguments, the `Exception` and the `EventName`. The exception

is a structure that contains all the information about the error. The information that will be included in this structure was discussed previously in this chapter. If you choose to have the `onError` method send an email to one of your developers, then this information can be included in the email to provide specific information related to the issue and should help them resolve the issue.

Here is an example of an `onError` method in script format. This code will capture the error, include a 'Sorry' page, and email the information to a developer.

```
public void function onError(required any exception, required string
eventname){
    include "sorry.cfm";

    var errorEmail = new mail();
    errorEmail.setTo(application.developerEmail);
    errorEmail.setFrom(application.systemEmail);
    errorEmail.setSubject('An Error has Occured');
    errorEmail.setBody('
        Message: #arguments.exception.message#<br />
        Details: #arguments.exception.detail#<br />
        Type: #arguments.exception.type#<br />
    ');
    errorEmail.setType('html');
    errorEmail.send();
}
```

Site-wide Error Handling

The last possible level of Error Management you can create, prior to having the default ColdFusion Error displayed to the user, is to create a Site-wide Error page. The Site-wide Error page is not actually site specific. A better name for this page would be a Server-wide Error page, as only 1 can be set on the server and will be displayed to all websites on the server. It is important to remember this when it comes to styling this page, for if you have multiple sites on the server, you do not want it branded for one specific site. Even though the page is not specific to a site on the server, it is a useful back up to have in place as it will catch ALL errors, including tag syntax errors, which `onError` will not catch.

To specify a Site-wide Error Handler, you need to open up your ColdFusion Administrator and navigate to Server Settings > Settings. On this page, under the Error Handlers heading, you will see a Site-wide Error Handler box. In this box you will enter the location of the file you wish to be displayed.

The path that you use should be relative to the server root. For example if you had a folder called `serverWideFiles` in the root of the server (remember, server, not site), then it might look like `/serverWideFiles/globalErrorHandler.cfm`.

Multiple Error Handling Strategy

When creating Error Handlers for your web site, it is often best to have multiple handlers in place. The main goal for any Error Handling Strategy is to prevent the generic ColdFusion error page from displaying to your user. Not only does this not look professional, it also shows users sensitive information about your server, such as file location paths. It is also important to remember that not all the Error Handlers catch all the errors, except the Site-wide error handler (which is the least accommodating handler).

When an error is thrown, it will bubble up the application until the first Error Handler catches it. The order in which the handlers will be called is:

1. cfcatch
2. onError
3. Site-wide Handler
4. ColdFusion Generic Handler

Things To Remember

When looking at Error Handling, there are a few things you should remember:

- While in development, there is no need for error handling. You will want to see any errors immediately so that you can resolve them right away. Place checks in your Error Handlers that can tell if you are in a development or production environment. The most common method for this is to check if the `CGI.remote_addr` is 127.0.0.1. If it is, then that means you are running the code on your local machine.
- When displaying a 'Sorry page', use as little ColdFusion as possible and do not include any files. The sorry page is being displayed because there is a problem in the code. Imagine if the problem is an issue within the header of your web site and you include the header. The 'Sorry' page will now throw an error. If your 'Sorry' page throws an error, it is possible that the 'Sorry' page will get called again, and again, and again. The next thing you know, you will have created an infinite loop that could take down your server. If you want to have the site header and footer on the 'Sorry' page, then place the generated HTML in the 'Sorry' page so that you know the page will not throw any errors.

Debugging

Fixing problems within your code can sometimes be very difficult. If the problem you are experiencing is an error, then the chances are it is pretty easy to fix as you already know the file and line number of the issue. If the issue you are trying to fix is that a piece of functionality is not acting as you expected, then the problem is a bit harder to resolve. There is no sure way to find these problems and fix them, but there are a few debugging methods that can often help you find the problem area.

Request Debugging Output

When trying to debug a problem within your application, it can sometimes be a very easy task or sometimes it can be a very arduous task. When an error is thrown, you often know the file and line number of the problem and you can fix it very easily. Sometimes the problem is that the application is not doing what you expected it to do and often this is a much harder problem to resolve. During these situations, the more information you have, the better. This is where the Request Debugging Output can help you.

On every page request, there is a lot of information that ColdFusion has about the request that you do not usually see. By default, this output is turned off,as it is not recommended that this be turned on in a production environment as it can slow down the page load times. If it were enabled on a production server, the information would not be displayed to everyone, as it is only shown to approved IP addresses. To turn on the Request Debugging information on your local server, log in to the ColdFusion Administrator and go to Debugging & Logging > Debug Output Settings; select the Enable Request Debugging Output checkbox. Once you click save, you should now see the debugging information at the bottom of every ColdFusion page that runs on your server. If you go to a ColdFusion page and you do not see that information, log back in to the ColdFusion Administrator and go to Debugging & Logging > Debugging IP Addresses. On this screen you will see a list of all IP addresses that are able to see the debugging information. Click on the Add Current button. This will add your local IP address to the list (127.0.0.1 and 0:0:0:0:0:0:0:1 are added automatically) and you should now see debugging information on every ColdFusion request.

The debugging information offers a lot of request information that can be very useful when debugging an issue. The output is comprised of the following sections:

- **Debugging Information** - This section gives you information about where the requested file is located, date and time information as well as some other server related information.
- **Execution Time** - This section gives you a list of all files that were executed in the request, in the order in which they were called, and how long they took to run. This information is very useful when debugging slow page loads or when needing to see all the files that were called.
- **SQL Queries** - This section displays all the SQL queries that were run on the page along with the SQL executed, how many records were returned, and how long the query took.
- **Trace Points** - This section works with the cftrace tag. This will be discussed later in this chapter.
- **Scope Variables** - This section is where the data that is stored in different scopes are output. Which scopes are output is controlled in the ColdFusion Administrator on the same page where you turned on this feature.

Using this information, you can often find where the problem might be occurring.

cftrace

The `cftrace` tag allows you to record the state of variables within your application at a specific time. It tracks the run-time logic flow, variables values, and execution time. The `cftrace` tag can be placed anywhere inside your code any number of times. You can specify a variable that you wish to trace, some text to associate with the trace, if you wish the trace to be inline or not, and if you wish it to stop the processing of the page or not. The `cftrace` tag has two output options: inline or not inline. If the `inline` attribute is set to true, then the trace information will be displayed at the end of the page request as well as within the request debugging output. If the attribute is set to false, then the information will only be displayed in the debugging output. In addition to the information being displayed on the page, the same information is stored in the `cftrace.log` file in the ColdFusion Administrator.

The `cftrace` tag has an `abort` option, which, when set to yes, will halt the page processing right after the tag. If this is set to false, the page will continue to process. Often, the best way to use the `cftrace` tag is to not set the abort option to true, but to let the page run. Placing multiple `cftrace` tags within the site will allow you to see the value of a variable at different stages of the request cycle. This will often allow you to track down where unexpected anomalies might occur.

Here is an example of the `cftrace` tag. In this example we are tracing a variable called today:

```
<cftrace
   var = "today"
   text = "I am expecting this to be Friday"
   type = "information" />
```

Here is the output we received when the tag was run:

```
 [23:26:26.012 /Users/simon/Sites/learncfinaweek/frontend/views/main/
default.cfm @ line: 8] [33 ms (1st trace)] - [today = Sunday] I am
expecting this to be Friday
```

As you can see, the value of Today is Sunday. As you can tell from the text added, we expected the value to be Friday; these values do not match, so we know that we must be close to our problem.

cfdump

Another method to help debugging a problem is to use the `cfdump` tag. The `cfdump` tag will output any variable, even variables that are not a simple value. Dumping variables allows you to see the current state of the variable and any values it might hold. If the variable is more complex, such as an object or ORM entity, then the `cfdump` tag will show you all available properties and methods for that object. In its simplest use case, the `cfdump` tag is provided a variable to be dumped. For simple variables, this is fine, but for variables that are more complex, such as ORM Entities, this can cause for a lot of information to be output and possibly cause a Java Heap error. For more complex

variables, it is often necessary to use some additional attributes. The `top` attribute allows you to specify a number which represents the depth that you wish the dump to go. For example, if you specified a top value of 3 and the variable was a query, only the top 3 rows would be output.

When `cfdump` is called, it outputs the provided variable and the page continues to process. In some scenarios, it is necessary for you to stop the page process right after the dump. If you are writing the `cfdump` tag, then you can simply add the word `abort` to your attribute list, and the page will stop processing after the tag. If you are using the script version,the `writedump` function, you do not have access to that feature and you must place an `abort` call after your `writedump` function call.

In situations where you have multiple `cfdumps` being called, it might get confusing which output relates to which tag in your files. For that purpose, there is a `label` attribute that allows you to specify some text which will be displayed with your dump.

Here is an example of a `cfdump` that dumps the session scope, only going 3 levels deep, and aborting once it has run:

```
<cfdump
    var="#session#"
    label="I am the session scope"
    top="3"
    abort />
```

And here is the script alternative:

```
writedump(var="#session#" top="3" label="I am the session scope");
abort;
```

One feature of the cfdump tag that can be very useful is its ability to write to a file rather than displaying to a page. This allows the page to be executed uninterrupted, but still allow you to get the data you need. To write to a file, just simply provide a file path to the `output` attribute.

Hands On 30

By Simon Free

In this hands on, you are going to create an error and view the error information in the log files.

Tags Used: <cfoutout>

1. To view the error information, we must first throw an error. Create a new file in the `/www/` folder called `throwError.cfm`.
2. Open up the `/www/throwError.cfm` file in your code editor.
3. Add the following lines of code:

```
<cfoutput>
    #foo#
</cfoutput>
```

4. Open the `/www/throwError.cfm` page in your browser. You will see an error that states `Variable FOO is undefined`.
5. Notice that the error is displayed but with not much additional information. This is not helpful when developing a web site. Open up your ColdFusion administrator and log in. The URL for the ColdFusion Administrator is most likely http://localhost:8500/CFIDE/administrator/.
6. Click on 'Debugging Output Settings' under 'Debugging and Logging'.
7. Check the 'Enable Robust Exception Information' option and click 'Submit Changes'.
8. Reload the `throwError.cfm` page in your browser. Notice that you now get the file name and line number the error is on.
9. Go back to the ColdFusion administrator.
10. Click on the 'Log Files' link under Debugging & Logging.
11. Click on the 'Search / View Log File' button (the first one) for the `application.log` file.
12. You will notice that the first item on the screen is details regarding the error that was just thrown. The application name column will show the name of the application which threw the error, in this case, `learncfinaweek`. Also, you will see details regarding the error that was just shown to you on the screen. You can see the error message, the file that threw the error, and the line number. When error handling is turned on, this information can be very useful when debugging errors.
13. Go back to the Log Files screen.
14. Click on the 'Delete Log File' icon (the 4th icon from the left) for the `exception.log` file. This will delete the exception log file.

15. Go back to the `throwError.cfm` page and refresh.
16. Go back to the Log Files screen and you will see the `exception.log` file is back.
17. Click on the 'Search / View Log File' icon (1st on the left) for the `exception.log` file.
18. The first item in the list will be the error that you generated. In this view, you will see a lot of additional information regarding the error, such as the full Java stack trace.

Hands On 31

By Simon Free

In this hands on, you are going to add an Error Handling solution to the web site.

Tags Used: <cfdump>
Functions Used: include, mail

1. Open up the `/www/Application.cfc` file in your code editor.
2. After the `onRequestStart` function, create a new function called `onError`. The function will accept 2 arguments. The first is of type `any` and is called `Exception`. The second is of type `String` and is called `EventName`. Your code should look similar to this:

```
function onError( any Exception, string EventName ) {


}
```

3. Inside the function, include the file `sorry.cfm`.
4. After the file, place the following code, which will email the error information to you:

```
var errorEmail = new mail();
errorEmail.setTo('you@domain.com');
errorEmail.setFrom('system@domain.com');
errorEmail.setSubject('An Error has Occured');
errorEmail.setBody('
    Message: #arguments.exception.message# <br />
    Details: #arguments.exception.detail# <br />
    Type: #arguments.exception.type# <br />
');
errorEmail.setType('html');
errorEmail.send();
```

5. Your function should look similar to this:

```
function onError( any Exception, string EventName ) {
    include 'sorry.cfm';
    var errorEmail = new mail();
    errorEmail.setTo('you@domain.com');
    errorEmail.setFrom('system@domain.com');
    errorEmail.setSubject('An Error has Occured');
    errorEmail.setBody('
        Message: #arguments.exception.message# <br />
        Details: #arguments.exception.detail# <br />
        Type: #arguments.exception.type# <br />
    ');
    errorEmail.setType('html');
    errorEmail.send();
}
```

6. Open up the `/www/throwError.cfm` file in your browser. You should now see a 'sorry' page.
7. Open up your ColdFusion Administrator and login. The URL for the ColdFusion Administrator is most likely http://localhost:8500/CFIDE/administrator/.
8. Click on 'Mail' under Server Settings.
9. Click on the 'View Undeliverable Mail'.
10. The first item in the list should be your error email. Review the contents of the email that is sent when an email is thrown.
11. Open up the `/www/throwError.cfm` file in your code editor.
12. Change the contents of the file to the following code:

```
<cfdump value="2" />
```

13. Open up the `/www/throwError.cfm` file in your browser and notice that an error is thrown. `onError` does not catch tag syntax errors. For that, we must rely on the site-wide error handler.
14. Create a new file called `sitewideErrorHandler.cfm` in the `/` folder.
15. Open the `/sitewideErrorHandler.cfm` file in your code editor.
16. Add the following line of text to the file:

```
An error has occurred!
```

17. Go back to your ColdFusion Administrator.
18. Click on 'Settings' under Server Settings.
19. Go to the Site-wide Error Handler input and enter the following value:

```
/learncfinaweek/sitewideErrorHandler.cfm
```

20. Go back to the `/www/throwError.cfm` page in your browser and refresh.
21. You will now see the message 'An error has occurred!'. The site wide error handler has caught the exception and displayed a friendlier message to the user. You can now find details about the specific error from the server log files.

14

i18n

By Paul Hastings

About Paul Hastings

Paul Hastings, who after over 25 years of IT work is now a completely fossilized geologist, is CTO at Sustainable GIS, a consulting firm specializing in Geographic Information Systems (GIS), Flex, and ColdFusion applications for the natural resource market, and of course ColdFusion globalization. Paul is based in Bangkok, Thailand, but says that it's not nearly as exciting as it sounds but sometimes can be. He can be reached at paul@sustainableGIS.com

What is Globalization?

The process of making an application ready for global usage is globalization, or G11N (for the 11 letters between the "g" and the "n" in globalization). Globalization consists of two steps: internationalization, or I18N (for the 18 letters between the "i" and "n" in internationalization), and localization or L10N (for the 10 letters between "l" and "n" in localization—if you're sensing a pattern here, yes there is, people working in this field are particularly fond of numeronyms). The atomic units for globalization are locales. Locales are the most important piece of G11N.

Locales

Locales are languages and calendars; date, number, and currency formatting; spelling; writing system direction; etc., that are specific to a geographic region. For instance, the English (color) and date formats (month/day/year) used in Brooklyn are not exactly the same as the English (colour) and date formats (day/month/year) used in Perth.

Since version 7, ColdFusion locales are based on core Java locales, which means the locale information concerning formatting, calendars, etc. are an integral part of the underlying Java platform and vetted by the Unicode Consortium—you do not have to research these yourself. Locale resources are accessed via locale identifiers in the form of language_Country, such as en_US for the English language used in the United States, or fr_CA for the French language as used in Canada. Take note of the case, as in all things related to core Java, it matters.

Internationalization (I18N)

The process of I18N is the first step in making your application globalized. It consists of stripping away any hard-coded text and date, time and number formatting that tie it to a specific locale and replacing these with variables or functions that can dynamically return locale appropriate content or formatting. This allows for the application to be adapted to various locales **without major engineering changes**. Considering the amount of work this can involve, it is perhaps best done from the outset rather after the fact (though its quite common for applications to be undergo G11N well after development has completed).

Localization (L10N)

Once an application has undergone I18N, the next step is making it ready for use in a given locale. This is done by translating all of the application's text into the various languages that your application will support as well as using appropriate locale-based functions to format dates, numbers, currency, etc. These function are usually prefixed by "LS" such as `LSdatetimeFormat`. You can find a reference for all ColdFusion functions and tags related to globalization here: http://adobe.ly/UrOesD

Note that during the translation process, locales need to be considered as well. It helps fine tune the content to a specific region.

Simple I18N Example

Lets say you have an original web application dealing with appointments and had something like the following block of code:

```
You have the following appointments for #dateFormat(now())#:
<ul>
     <cfoutput query="todaysAppointments">
          <li>#appointmentWith#: #timeFormat(appointment)#</li>
     </cfoutput>
</ul>
```

After I18N that would look like the following:

```
<cfprocessingdirective pageencoding="UTF-8">
<cfset setLocale( session.locale) />
#appointmentsRB[session.locale].appointmentsText
#LsdateFormat(now(),"FULL")#:
<ul>
     <cfoutput query="todaysAppointments">
     <li>#appointmentWith#: #LstimeFormat(appointment,"MEDIUM")#</li>
     </cfoutput>
</ul>
```

where

- `cfprocessingdirective pageencoding="UTF-8"` identifies the character encoding used on this page, UTF-8.
- `session.locale` is a session variable holding the user's preferred locale, usually set at login or application initialization.
- `setLocale( session.locale)` tells ColdFusion what locale this page will be using.
- `appointmentsRB` is a resource bundle (see below) as a ColdFusion structure holding the application's translated text.
- `appointmentsText` is the structure value holding the translated text for "You have the following appointments for".

Some Things to Consider when Building a Globalized Application

Locales

Since your entire globalized application flows from a user's locale it is critical that this be captured and stored for use throughout the application. The easiest way of capturing a user's locale is simply to ask for it. For example, if your application requires registration, ask for it then. As a tip, it's usually a good idea to display locale choices in that language: French in French, Thai in Thai, etc.

If asking isn't possible, a stealthier approach is to examine the user's browser-provided data such as `http_accept_language` which is accessible via the `CGI` scope. You can also supplement this by examining the user's IP address and looking up their country from one of the many online geolocation services. Note that it's a good practice, however, to also provide a way for a user to manually change their locale.

It's important that the user's locale be persisted and used throughout the application. A session-scoped variable is often the best choice for this. The relevant ColdFusion function for handling locales are:

- setLocale, which tells ColdFusion to use the supplied locale identifier for the current page.
- getLocale which returns the locale ColdFusion is currently using.

Character Encoding/Writing Systems

A character encoding is a map for each character in a language to a numeric code that can be represented in a computer. For a variety of reasons, there are often more than one encoding for a language. Japanese, for example, is represented by Shift-JIS, EUC-JP, and ISO-2022-JP encodings. Since each encoding is basically the same set of numbers, data and application content couldn't be dynamic. For instance, in a web application, each web page had to be encoded in that language's code page--developers therefore had to develop and maintain one page per encoding the application needed to support. Data for each encoded web page had to be stored in its own column in a table or its own table in a database. Prior to the creation of Unicode in 1987, developers always had to climb this encoding Tower of Babel, making globalized applications extremely costly and very limited in scope. Unicode allows for a single, standard encoding scheme for much of the world's languages. Since all modern databases now support it, it's the logical choice for character encoding for globalized applications. In short, **just use Unicode.**

A language's writing system dictates the way a language is written. There are basically three flavors:

- Left-to-right (LTR) -used in Western languages such as English, French, German, etc.
- Right-to-left (RTL)- used in Middle Eastern languages such as Hebrew and Arabic.
- Vertical- used in some Asian languages (traditional Japanese uses tategaki , top-to-bottom, right to left format as does traditional Chinese from which it was derived—modern forms of Japanese and Chinese follow a LTR format, some print formats still use a vertical layout).

Writing systems have an impact on an application's layout and display. It's especially important for RTL languages. Not only is the text RTL, but the whole application layout needs to be RTL—visual attention is no longer in the upper-left corner but instead in the upper-right.

Resource Bundles

As mentioned above, one common way to handle localized text at run time is via resource bundles. All static text in the application is replaced by variables that hold that specific text translated into the different languages that need to be supported. A resource bundle can be a simple ColdFusion structure:

```
appointments={};
appointments.en_US.greeting="Hello"; (American English)
appointments.fr_FR.greeting="Bonjour"; (French)
appointments.de_DE.greeting="Hallo"; (German)
appointments.sv_SE.greeting="Hallå" (Swedish)
appointments.th_TH.greeting="สวัสดี"; (Thai)
appointments.ja_JP.greeting="もしもし"; (Japanese)
```

where the various locales in this structure could be accessed at run time by supplying a locale key (en_US, fr_FR, etc.). This approach, however, is very cumbersome to maintain, especially as the application grows in complexity or the number of locales it supports. Another, perhaps better approach, is to use files or a database to hold the translated text, but this requires advanced methodologies beyond the scope of this course and will be addressed in week 2.

Dates, Times, Calendars and Time Zones

Date formats are particularly vexing to many developers. For instance, a date string of 1/2/2012 in en_US locale is `January 2, 2012`, while in en_AU it's `1 February 2012`; that's quite a difference if you're making an appointment. To ensure that date and time formats are correct for a user's locale, use the following functions:

- LStimeFormat
- LSdateFormat

to format dates and times.

Note that it's usually a good idea to format dates and times using one of the regular masks, FULL, LONG, MEDIUM, or SHORT to ensure your application isn't breaking some cultural rule as well as to make sure the dates and times the application displays to the user can also be parsed back to a ColdFusion date-time object via `LSParsedatetime` function.

ColdFusion dates are based on the Gregorian calendar which will suffice for many locales. While this is an advanced topic, it's important to note that there are several other calendars in common use globally:

- Islamic calendar
- Buddhist calendar
- Chinese calendar
- Indian calendar
- Persian calendar

Handling non-Gregorian calendars will be covered in the week 2. of this course.

Time zones are another issue to many developers, especially if users are in one time zone while the ColdFusion server is in another. Again, this is an advanced topic but developers should take note of

the following points:

- ColdFusion considers all date-times to be in the server's time zone; ColdFusion doesn't care about your intentions, just the server's time zone.
- If ColdFusion handles any date-times, these will be converted to the server's time zone.
- The simplest option for handling time zone issues is not to store date-time objects but instead store epoch offsets such as Java's (milliseconds since 1-jan-1970) .
- Basic timezone handling is done via the getTimeZoneInfo, which returns a structure holding the server's time zone information.

Numbers/Currencies

Similar to date and time formats, it's important to display numbers and currencies in the locale's correct format. For example, 123456789.10 in en_US locale would be understandably displayed as 123,456,789 to Americans. The same number would be displayed as 123 456 789 in fr_FR locale and unfamiliar to most Americans. The relevant ColdFusion functions for formatting numbers and currencies are:

- LSnumberFormat
- LScurrencyFormat

Note that the masks used in these function will map the dollar sign ($), decimal (.) and comma (,) to the appropriate locale symbols. Also note that the `LScurrencyFormat` function will not convert between currencies; it's **not** a function for handling exchange rates.

Databases

As mentioned above, modern databases are all Unicode-capable and shouldn't be an issue when considering which one to deploy. It's only important that any database-specific setup, data types, collations, etc. be followed to ensure your application's database is able to fully use Unicode and perform sorting in a locale specific way.

Hands On 32

By Simon Free

In this hands on we will change our locale and update the date and time outputs to change the display based on our locale.

Functions Used: LsdateFormat, LstimeFormat, setLocale

1. To be able to change the date and time formats based on our locale, you need to update the functions used for date and time output. Open up the `/www/blogpost.cfm` file in your code editor.
2. Locate the `blogPost.datePosted` variable output on or around line 52.
3. Replace the `dateFormat` code with the following line of code:

```
#LsdateFormat(blogPost.datePosted,'short')#
```

4. Locate the `comment.createdDateTime` variable on or around line 72.
5. Change the `dateformat` code with the following code:

```
#LsdateFormat(comment.createdDateTime,'short')#
```

6. Change the `timeFormat` code with the following code:

```
#LstimeFormat(comment.createdDateTime,'medium')#
```

7. Open up the `/www/blog.cfm` page in your browser.
8. Navigate to a blog post and review the date and time output. If you are based in the U.S., you will probably not notice any difference. If you are outside of the U.S., then you will probably see a different date and time format.
9. To allow everyone to see a change in the formatting, let's create a temporary page which will change our locale. Create a file called `updateLocale.cfm` in the `/www/` folder.
10. Open up the /`www/updateLocale.cfm` file in your code editor.
11. Add the following line of code:

```
<cfset setLocale('English (UK)') />
```

12. Open up the `/www/updateLocale.cfm` page in your browser. You will not see any output, but the new locale has been set (unless you are in the UK; in that case, you should try a different

locale such as English (US)).

13. Navigate back to the /www/blog.cfm page and click on a blog post. You should now notice that the date and time formats have changed. The changes might be slight, but they are there.

What to do Next

Now that you have finished Learn CF in a Week, you should have a good understanding of ColdFusion. While you might not be ready to go out and lead a team of developers in building a highly complex and scalable application, you should feel confident enough in creating a few basic applications while you get your feet wet.

If you have been following along with the Hands On chapters, you should also have a completed web site. If you do not already have a personal web site, feel free to find some hosting and launch the web site. Having a personal website is a great way of advertising and showcasing your skills to others.

With the completion of this course, you are probably wondering what you should do next. There are a few things that you can do to help improve your ColdFusion knowledge:

Read Articles and Blogs

There are many articles and blog posts published on a regular basis that will help you advance your knowledge. Often, these resources will be on specific ColdFusion functionality or development problems that you might come across while you are developing. These resources are jam packed with information from developers who have come across the problem and found a good solution. Here are a few useful resources:

- **Adobe Developer Connection**: http://www.adobe.com/devnet/coldfusion.html
  The Adobe Developer Connection has weekly updates to its already extensive pool of articles. Not only will you find ColdFusion articles, you will also find articles on topics such as mobile development and jQuery that you might also find interesting.

- **Adobe TV**: http://tv.adobe.com/show/adc-presents/
  Adobe TV has a number of training videos specific to ColdFusion. These videos demonstrate new features of ColdFusion, as well as demonstrate how to use some of the IDE's such as ColdFusion Builder.

- **The Blog of Ben Nadel**: http://www.bennadel.com/
  Ben Nadel is a famous figure in ColdFusion. His blog posts go into a depth that none can compete with. His posts are informative and can explain it in a way everyone can understand.

- **cfbloggers.org**: http://www.cfbloggers.org
  cfbloggers.org is a ColdFusion blog aggregator. Anyone who is anyone in the ColdFusion world has their blog aggregated by cfbloggers. The feed is constantly updating with new blog posts from people around the globe ,allowing you to keep up with all the new developments in the ColdFusion world.

Attend User Group Meetings

Throughout the world, ColdFusion user groups are meeting up on a monthly basis to discuss all facets of ColdFusion. User groups are a free event where like minded developers meet to learn new things. Each meeting will have a different presenter who speaks on a different topic each month. To find your nearest ColdFusion user group near you, visit http://groups.adobe.com/.

Attend Conferences

There are a number of conferences throughout the year that offer presentations on ColdFusion topics. These conferences provide an opportunity to attend a wide variety of presentations on numerous topics, some which you might have experience with and some you might never have come across before. These conferences give you the chance to broaden your knowledge and directly interact with the speakers. Also, you will meet and socialize with other ColdFusion developers. It is not uncommon for people to leave a conference filled with new ideas that they wish to try out.

Always keep your eyes and ears open for new conferences that might start up, but for now, here is a list of established conferences that are held each year:

- **cfobjective**: http://www.cfobjective.com/
  cfobjective is a 3 day ColdFusion conference in Bloomington, Minnesota. The conference has a number of advanced level ColdFusion topics. It is attended by many of the leading ColdFusion community members and is packed with great information.

- **RIA Con**: https://www.riacon.com/
  RIA Con is a 2 day conference in the Washington, D.C. area. The conference covers many ColdFusion and non-ColdFusion topics.

- **NCDevCon**: http://www.ncdevcon.com/
  NCDevCon is a 2 day conference in Raleigh, North Carolina. The conference covers a number of topics from ColdFusion to HTML5. As well as having regular sessions, NCDevCon also has a number of Hands On sessions. Priced as one of the least expensive ColdFusion conferences out there, you definitely get the value for your money.

- **Scotch on the Rocks**: http://www.sotr.eu/
  Scotch on the Rocks is a European ColdFusion conference. The conference is based out of Edinburgh, but has been known to travel around; locations have ranged from Edinburgh to London to numerous places in Europe. The conference offers a wide variety of ColdFusion topics presented by some of the leading ColdFusion experts in Europe.

Listen to Podcasts

Podcasts are a great way to learn what is going on in the ColdFusion world as you get to listen to them at your own leisure. Maybe it is when you are on your commute into work, or maybe when you are hitting the gym. Whenever it is, podcasts are packed with lots of information about what is going on in the ColdFusion community, any new information that Adobe has released about the next version of ColdFusion, and interviews with members of the ColdFusion community.

Here are a few of the current ColdFusion podcasts:

- **<cfhour>**: http://www.cfhour.com/
  Hosted by Dave Ferguson and Scott Stroz, <cfhour> provides you with updates about ColdFusion and interviews with members of the ColdFusion community, as well as a few well timed jokes.

- **2 Devs from Down Under**: http://www.2ddu.com/
  Join Mark Mandel and Kai Koenig for their podcast that covers all things ColdFusion and all things development.

- **Bolt Talks**: http://cfmumbojumbo.com/cf/index.cfm/bolttalks/
  Tim Cunningham holds conversations with some of the leading minds in the ColdFusion world. Filled with fun interviews and opinions, these podcasts will help you understand what makes a ColdFusion developer tick.

Get Certified

Besides building some fantastic applications, one way to demonstrate to people how great your ColdFusion knowledge is to get ColdFusion Certified. Adobe offers a ColdFusion certification test that anyone can take. There is a fee involved, but once passed, you will have the ability to call yourself an Adobe Certified Expert (ACE). Not only is this a great thing to show your employer or future employer, but you also get to display a special logo on your web site!

To find out more about the Adobe ColdFusion Certification, visit: http://training.adobe.com/certification/exams.html#p=1&product=adobe-cold-fusion. Currently the only certification available is the ColdFusion 9 certification. The ColdFusion 10 certification will be released soon.

Sign Up for our Mailing List

This isn't a plug just to get your email address, I promise! There are plans for Learn CF in a Week to be expanded with additional weeks. Week 1, which you just completed, was covering the basics of ColdFusion. With week 2, we plan on covering more advanced topics. We even have initial plans for week 3 and possibly even a week 4. If you sign up for the mailing list, we can notify you when the new updates have been released.