Module 3: CFCs - Building ColdFusion Components

In this section you will learn about:

- Reviewing ColdFusion Components
- Creating a static component
- Defining functions in components
- Invoking static CFC methods
- Using CFC self-generating documentation
- Creating instance based components
- Persisting CFC instances
- Inheriting methods, properties, and data between components
- Restricting access to component methods

ColdFusion MX introduced the ability to create ColdFusion Components (CFCs). These components can store User Defined Functions (UDFs) and can even be treated Objects must the same way Object-Oriented Programming languages do.

They are arguably by far the most important new addition in ColdFusion in a long time and may severely change the way many ColdFusion applications are written.

CFCs allow developers to store related functions together so that may be used as a set. On one level, a CFC may simply be seen as a storage place for functions; however, CFCs go way beyond acting as a storage location.

Benefits to using CFCs

CFCs offer a variety of benefits. CFCs allow for:

- **Separation of the programming logic from the design** – Today's complex web applications involve usage of high level programming and design principles. Business logic is often interspersed within HTML tags, Adobe Flash, images, forms, etc. For a non-ColdFusion developer, this programming code can be intimidating and can increase development time while s/he tip-toes around loops and function calls. CFCs allow for a neat separation of these two types of

code allowing a team of developers to work side-by-side each other without getting in each others way.

- **Reuse of code**– The functions stored in a CFC may be called from multiple pages. They may accept parameters and may return resulting values. This allows for reuse of commonly needed code without any recoding. These functions may even be accessed by Flash and other applications again without recoding.

- **Self-documentation** – CFCs have a self-documenting feature that allows them to be viewed directly in the browser. Doing so exposes information about required parameters, available functions, etc. This can help a team of developers who must rely on each others work.

- **An easy transition in into Web Services** – CFCs can (easily!) be exposed as Web Services. This will be explored in great detail the next section.

- **Object Oriented** – Use of CFCs can be done in a way that is consistent with Object-Oriented Programming (OOP). This is discussed later in this Module.

Creating Components

Components are saved as separate documents that have a **.cfc** extension. The component will contain one or more function. These are made available to other pages as methods. The functions are generally created with the <cffunction> tag. (*Note: Like other UDFs, functions may be created with the <cfscript> tag, however, these will be much more limited since UDFs created with <cfscript> tags will not allow any CFML tags in the function.*)

In a sense, you can think of the component as an include or a custom tag. These documents also can store reusable programming logic in an external file and are accessible by multiple CFML files. As you will see, however, CFCs capabilities exceed those of includes or custom tags.

The following component, saved as **CFCs/demos/component-date.cfc**, stores a simple function that determines today's date. The <cfcomponent> tag begins and ends the page, enclosing the UDF, which is wrapped in <cffunction> tags. Notice that this is not a complete CFML page. It will not be called by the user in the browser. It is merely a piece of code that will be called from another page. (Although, calling a CFC directly in a browser has a neat use that you will see later.)

```
<cfcomponent displayname="Display the Date" hint="This component displays
the date in the format ddd mmm, ss yyyy">
  <cffunction name="getTheDate" returntype="string">
    <cfset thedate="#DateFormat(now(), 'ddd mmm, dd yyyy')#">
    <cfreturn thedate>
  </cffunction>
</cfcomponent>
```

This <cfcomponent> tag has three optional attributes. The first, "extends," will be discussed in the object oriented section. The others, "displayname" and "hint," both help with the self-documenting features.

Some Definitions

Below are definitions of some of the common terms used in Component construction.

- **Component** – Think of the component as the .cfc page itself. It may hold multiple functions
- **Component Package** – A number of components may be stored in the same directory on the server. This is known as a package. This organization structure can ensure that names are unique. It will be discussed in depth a little later.
- **Method/Function** – Any UDF that is inside a CFC is called a function or a method. The term method will be familiar to anyone with OOP experience. These two words are used interchangeably.
- **Invoke** – When a function is called or executed, it is invoked. There are a number of ways to invoke a function.
- **Argument** – Sometimes called a parameter, this is a value that is passed into a function. Many functions have required or optional arguments.

New Tags

Most of the code in CFCs will use tags that you already know. CFCs use a shift in development strategy more than major change in the way that CFML works. The following four tags are new in CFMX and are important in CFC development.

- **<cfcomponent>** – holds all of the functions
- **<cffunction>** – acts as a wrapper for the function's code. These functions can be called from outside a CFC whenever the CFC is invoked
- **<cfreturn>** – used to return a value from a function to the place that it was called
- **<cfargument>** – allows the function to use incoming arguments or parameters

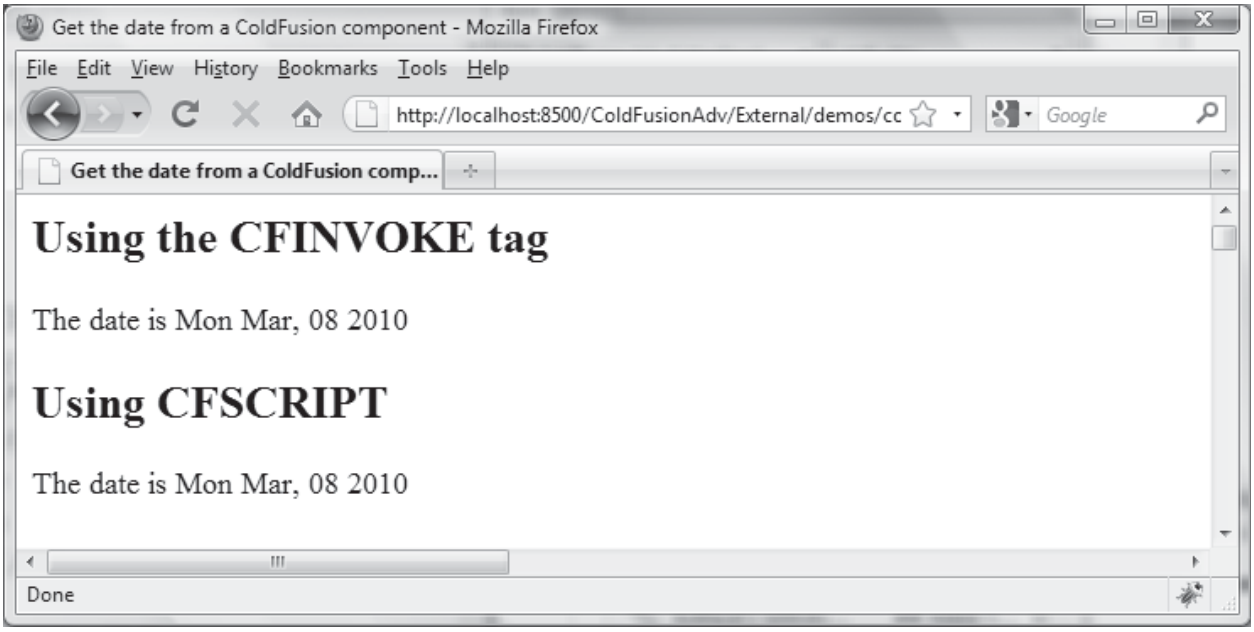
Access control

The "access" attribute of the <cffunction> tag is what determines who or what may access this file. There are four valid values. Read them carefully, they are not necessarily what they sound like:

- **Private –** This is the most secure access level for a function. It will only be available to other functions within the same CFC. It will not be accessible directly from any .cfm page even on your own server.
- **Package –** This type of function will only be accessible by CFCs in the same package (the same directory) as this function. As with "private," it will not be accessible directly from any .cfm page even on your own server.
- **Public –** If a function will be called from .cfm pages on your site, this is the required access level. For most sites, this will be a common access level. It will not be available from outside your server.
- **Remote –** Use "remote" only when you intent to open a particular function to the outside world.

Calling or Invoking Components

When our method named "getTheDate" is called, we will receive a formatted date that is displayed on the page. This demo common ways to call (or invoke) a method in a component:



Two common ways to call the functions that are stored in components.

1. With the <cfinvoke> tag
2. Using a <cfscript> block with the createObject method or the <cfobject> tag. (As the word "object" implies, this will be used in the OOP section.)

The following demo uses all three. It is saved as **CFCs/demos/component-date.cfm**.

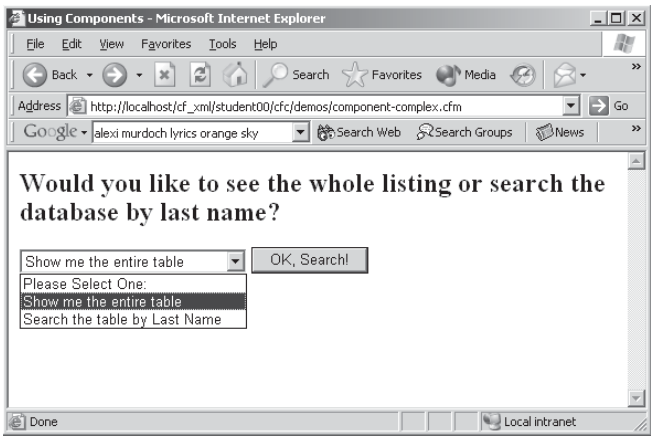
```
<html>
<head>
   <title>Get the date from a ColdFusion component</title>
</head>
<body>

<h1 align="center">Three ways to access components</h1>

<h2>Using the CFINVOKE tag</h2>
   <cfinvoke component="component-date" method="getTheDate"
returnvariable="datereceived">
   <p>The date is <cfoutput>#datereceived#</cfoutput></p>

<h2>Using CFSCRIPT</h2>
   <cfscript>
      datewithscript = createObject("component","component-date");
      newdatereceived = datewithscript.getTheDate();
   </cfscript>
   <p>The date is <cfoutput>#newdatereceived#</cfoutput></p>

<h2>Remote Access</h2>
<p>Functions inside components can be set for remote access.  Click on
the link in the next paragraph</p>
      <p>This is an <a
href="http://<<server_name>>/ColdFusionAdv/CFCs/demos/component-date-
external.cfc?method=getTheDate">external function</a></p>
</body>
</html>
```

Passing Arguments to Components

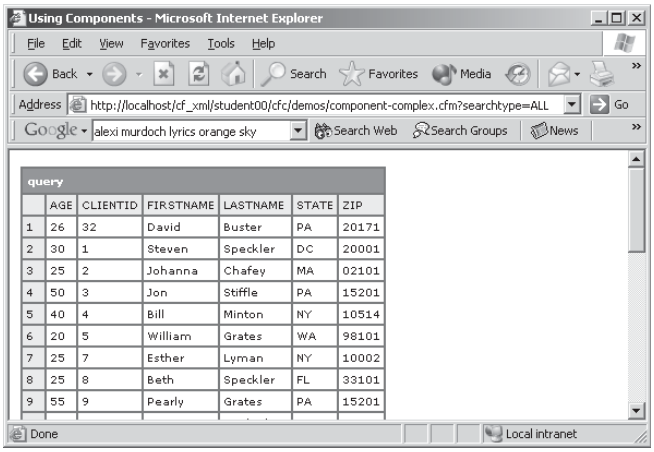
As mentioned above, functions are able to take arguments or parameters. This means that we can change the way they behave by passing in a value. In the following example, we will examine two functions. The first one queries the database for an entire table. The second is expecting to receive a value for the last name. It will then search the database for records that match that last name.

In our example, users will begin with the page called **CFCs/demos/component-complex.cfm**. Note the .cfm extension. This is not the component, but the CFML page that calls the component.

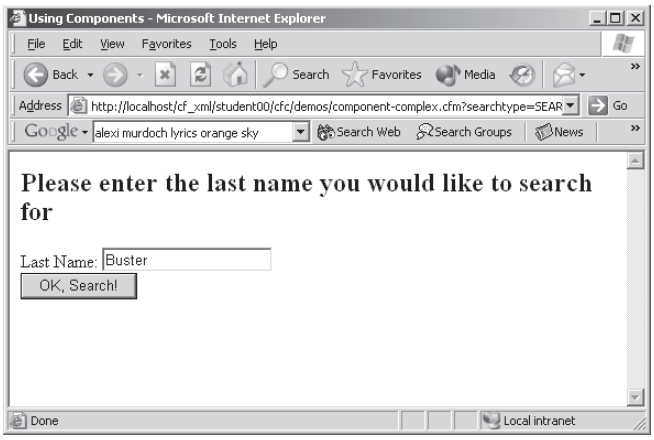
In “Phase One” of this page, the user is asked to select from a drop down list:



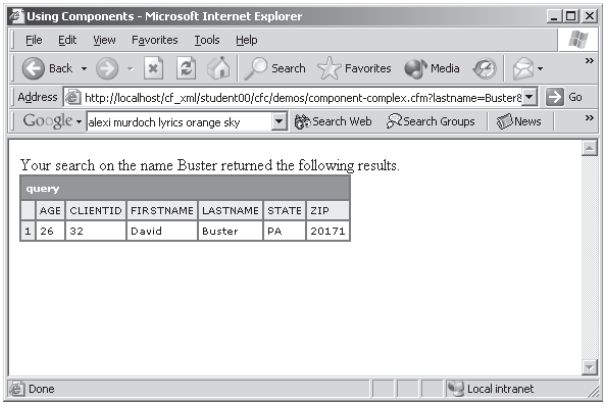
In "Phase Two," if the user selects "Show me the entire table," the form submits a variable to itself called searchtype which has a value of "ALL." This triggers the invocation of a component method:



If, instead, the user selected the option to search by last name, a form is displayed. Again, notice the URL. The value for searchtype is now "SEARCH." This is "Phase Three":



If the user enters a value of "Buster," we will see "Phase Four"



The Component Code

Let's examine the code for the two methods in the component. It is saved as **CFCs/demos/component-query.cfc**:

```
<cfcomponent>

   <!--- This function will return all clients --->
   <cffunction name="getAllRecords">
      <cfquery name="getTheRecords" datasource="courses">
         SELECT * FROM clients
      </cfquery>
      <cfdump var="#getTheRecords#">
   </cffunction>


   <!--- This function will expect a parameter -
      a value for lastname --->
   <cffunction name="searchRecords">
      <cfargument name="lastname" required="true">
      <cfquery name="getSomeRecords" datasource="courses">
         SELECT * FROM clients
```

```
         WHERE lastname = '#arguments.lastname#'
         ORDER BY lastname, firstname
      </cfquery>
      <cfoutput>Your search on the name #arguments.lastname# returned the
following results.</cfoutput>
      <cfdump var="#getSomeRecords#">
   </cffunction>

</cfcomponent>
```

The first function will run a query of the table named “clients” with no WHERE clause, thus returning all of the records. The second function, however, is expecting to receive a parameter. The <cfargument> tag tells the function that it should look for the parameter when it is invoked.

The main CFML page

The page the stores all four phases of this application is stored as **CFCs/demos/component-complex.cfm**. Here is the code from the page. Note the section in “Phase Four” which invokes the function while passing the value for lastname.

```
<html>
<head><title>Using Components</title></head>
<body>
<!---
   This is phase one: determine whether the user will search
   by last name or see the whole table
--->
<cfif NOT isDefined("searchtype") OR URL.searchtype IS "">
   <h2>Would you like to see the whole listing or search the database by
last name?</h2>
   <form method="GET">
      <select name="searchtype" >
         <option value="">Please Select One:</option>
         <option value="ALL">Show me the entire table</option>
         <option value="SEARCH">Search the table by Last Name</option>
      </select>
      <input type="submit" value="OK, Search!">
   </form>
<!---
   This is phase two: If the user selects "all," invoke
   the "getAllRecords" method of the component named
   "component-query"
--->
<cfelseif URL.searchtype IS "ALL">
   <cfinvoke component="component-query" method="getAllRecords">
<!---
   This is phase three: If the user selects "search,"
   provide a form for the entry of last name
--->
```

```
<cfelseif URL.searchtype IS "SEARCH">
   <h2>Please enter the last name you would like to search for</h2>
   <form method="GET">
      Last Name: <input type="text" name="lastname"><br>
      <input type="submit" value="OK, Search!">
      <input type="hidden" name="searchtype" value="GO">
   </form>
<!---
   This is phase four:  invoke the "searchRecords" method
   of the component named "component-query" passing the
   entered value for last name
--->
<cfelseif URL.searchtype IS "GO">
   <cfinvoke component="component-query" method="searchRecords"
lastname="#URL.lastname#">
</cfif>
</body>
</html>
```

Three Ways to Pass Arguments

The example above shows one technique in passing variables to a function in a CFC. There are, however, others ways.

Passing Arguments – Method 1 – as an attribute of <cfinvoke>

Firstly, this is the technique used above. The arguments may be passed as attributes of the <cfinvoke> tag. This is a convenient method but could prove to be problematic since the attributes of this tag will always be changing. It might be confusing if used on a team of developers who may expect a tag to have a consistent structure from application to application.

```
<cfinvoke component="component-query" method="searchRecords"
lastname="#URL.lastname#">
```

Passing Arguments – Method 2 - <cfinvokeargument>

The <cfinvokeargument> tag allows for arguments to be explicitly passed in a tag of their own.

```
<cfinvoke component="component-query" method="searchRecords">
   <cfinvokeargument name="lastname" value="#URL.lastname#">
</cfinvoke>
```

Passing Arguments – Method 3 – as a structure

It is also possible to pass a structure using the "argumentcollection" attribute of the <cfinvoke tag>. Since collections such as the #FORM# collection are structures, they may also be passed in this fashion!

```
<cfset thestructure = StructNew()>
   <cfset thestructure.lastname ="#URL.lastname#">
   <cfset thestructure.address ="...">
   etc.

<cfinvoke component="component-query" method="searchRecords"
argumentcollection = "#thestructure#">
</cfif>
```

Where to save CFCs

CFCs must be saved in a location that is accessible by the calling page. This means that they may be saved in the same places as custom tags – the root directory, the same directory as the calling page or the Custom Tags directory. Additionally, they may be saved in folders that have a mapping set in the ColdFusion Administrator. (They may also exist on another server if that are exposed as Web Services. See the next section for details on Web Services.)

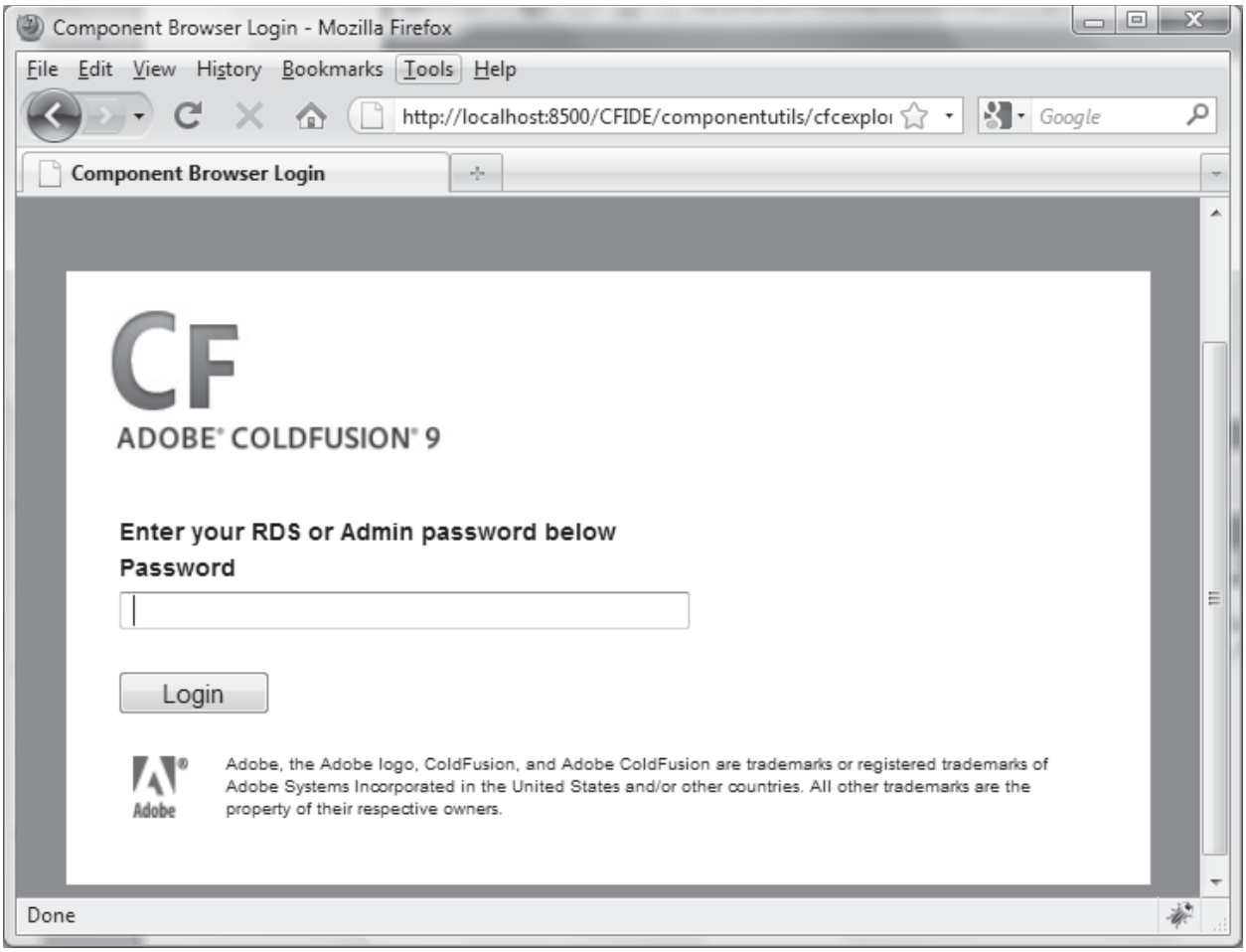
Packages

Especially if you expect to have a large number of components, it is a good idea to organize them in "packages." This is really a fancy name for a directory or file on your server. All components stored in the "rootdir\mystuff\comp\" folder will be accessible using the syntax "mystuff.comp" or if one component is saved as "rootdir\mystuff\comp\coolStuff.cfc" it will be known as "mystuff.comp.coolStuff".

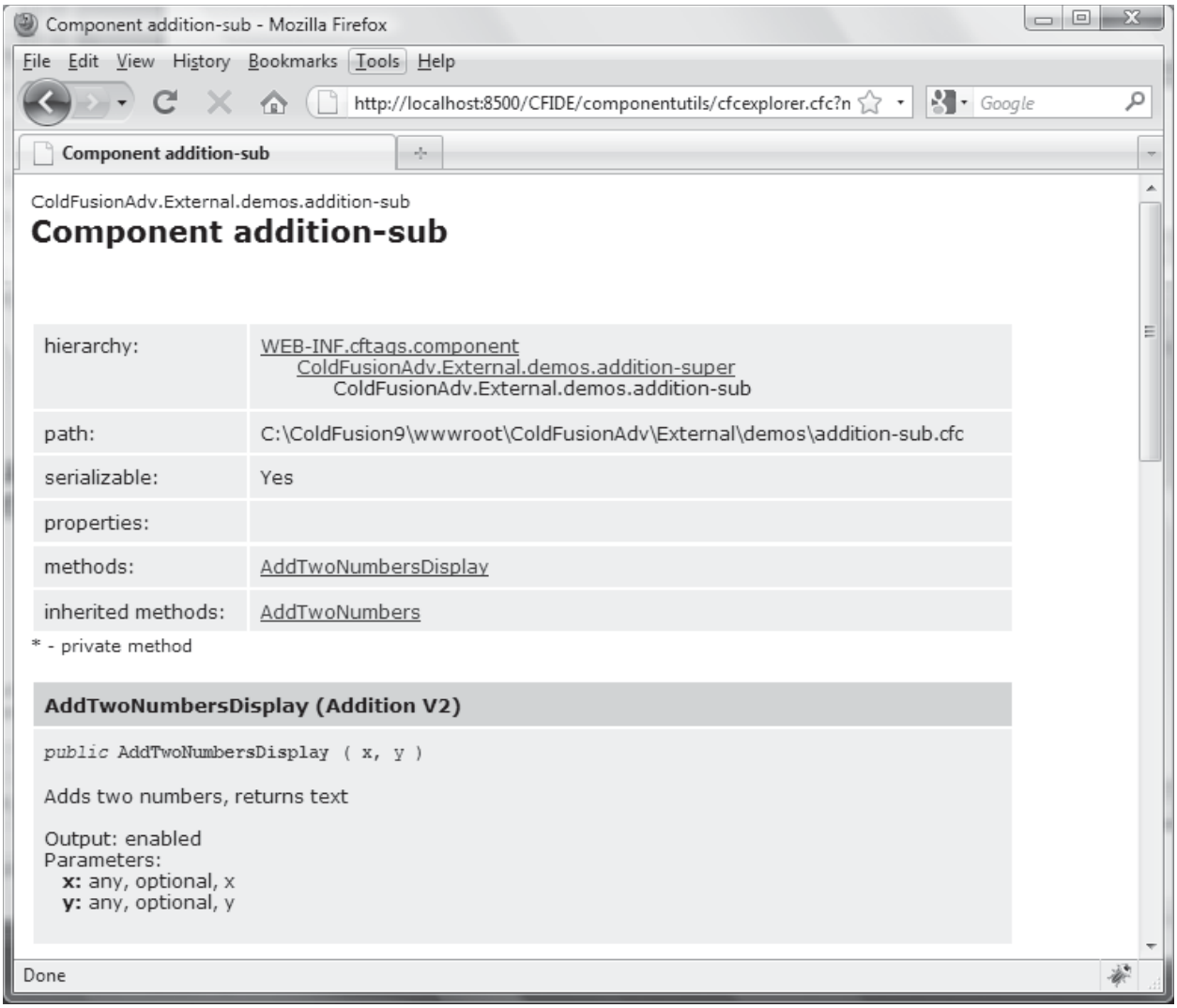
This means that you can be sure not to have an unexpected naming conflict if, for instance, you have two functions with methods named "getstuff." You should be sure which component's method is being called. It can be confusing (not to mention inefficient!) to store multiple copies of the components in the same directory as any page that calls it.

Self-documenting - View a Component directly in the browser

If you have the administrator password for your server, you will be able to view the component directly in the browser.



When you do so, ColdFusion will display some information about each of the functions in the component including the names of all functions and any parameters:

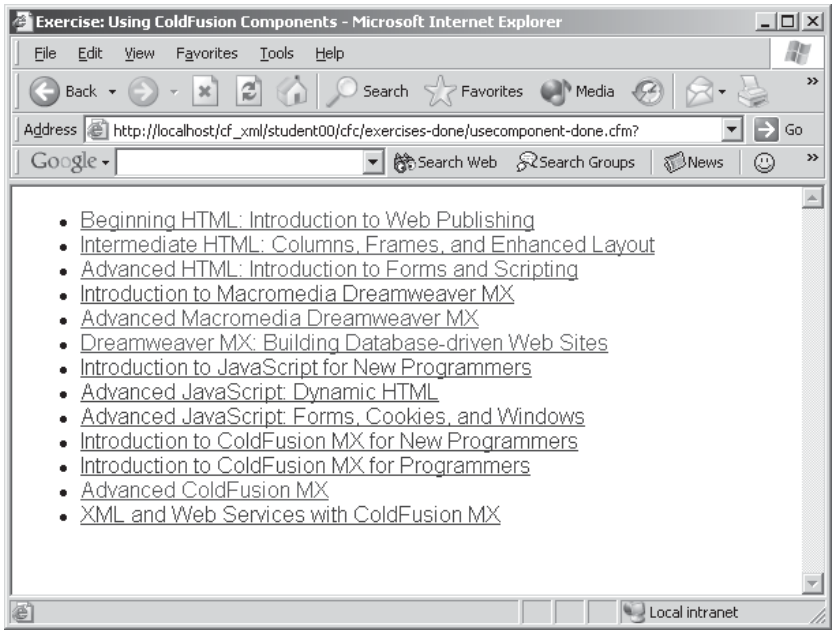

Component addition-sub - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8500/CFIDE/componentutils/cfcexplorer.cfc?n
Google
Component addition-sub
ColdFusionAdv.External.demos.addition-sub
Component addition-sub
hierarchy:
WEB-INF.cftags.component
ColdFusionAdv.External.demos.addition-super
ColdFusionAdv.External.demos.addition-sub
path:
C:\ColdFusion9\wwwroot\ColdFusionAdv\External\demos\addition-sub.cfc
serializable:
Yes
properties:
methods:
AddTwoNumbersDisplay
inherited methods:
AddTwoNumbers
* - private method
AddTwoNumbersDisplay (Addition V2)
public AddTwoNumbersDisplay (x, y)
Adds two numbers, returns text
Output: enabled
Parameters:
x: any, optional, x
y: any, optional, y
Done

Exercise 6: Creating ColdFusion Components with Arguments

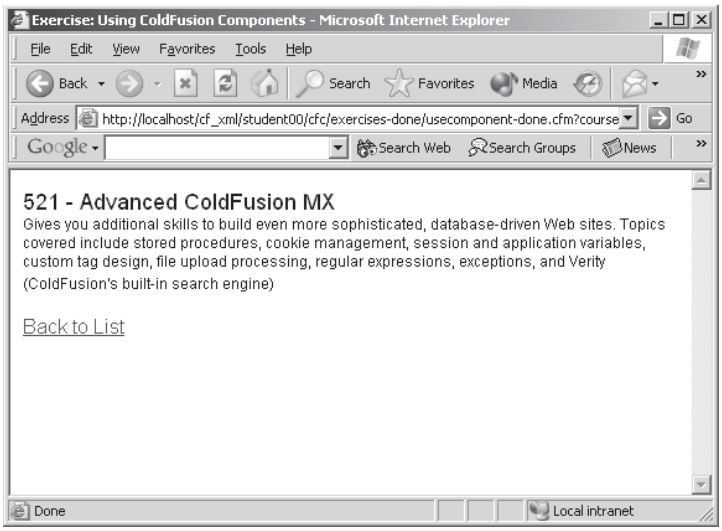
20 to 30 minutes

Build a CFC that stores two functions. Both will access the "courses" table in our database. Then, create a main CFML page that will invoke the appropriate function and pass the appropriate value.

In Phase 1 – The main .cfm page (not the CFC!) will invoke a function in the CFC named "getCourses" which will return a query that includes a list of all courses. The course titles will be links that will pass the course number to "Phase 2" of our page. When Phase 1 is completed, it may look something like this:



In Phase 2 – The main page will now invoke a function named "courseDetail." This function should require a parameter – the course number (named "coursenum"). This course number will be used in a query to return detail information about one particular course. When Phase 2 is completed, it may look something like this:



For this exercise, complete the following:

1. Open **CFCs/exercises/courseinfo-temp.cfc** and you will find the following comment:

```
<!---
Create a cfcomponent tag that include two functions:

1 - getCourses will include the following query and will return the
resulting
   recordset to the calling page
      <cfquery name="getTheRecords" datasource="courses">
         SELECT coursenum, name
         FROM courses
      </cfquery>
2 - courseDetail will expect to receive an argument named coursenum and
will
   include the following query:
      <cfquery name="getOneRecord" datasource="courses">
         SELECT coursenum, name, description
         FROM courses
         WHERE coursenum = #arguments.coursenum#
      </cfquery>
   It will return the resulting recordset
--->
```

2. Create a component that includes two functions and outlined in the comment above.

3. Open **CFCs/exercises/usecomponent-temp.cfm** and you will find the following code:

```
<html>
<head>
   <title>Exercise: Using ColdFusion Components</title>
</head>
<body style="font-family:Arial">
<cfif NOT isdefined("coursenum")>
```

```
   <!---
   Use the cfinvoke tag to call the component named courseinfo-temp
   and its function getCourses
    - the returnvariable should be called "courseList"
   --->

   <ul>
      <cfoutput query="courseList">
         <li><a href="?coursenum=#coursenum#">#name#</a></li>
      </cfoutput>
   </ul>
<cfelse>
   <!---
   Use the cfinvoke tag to call the component named courseinfo-temp
   and its function courseDetail - passing the coursenum with
   a cfinvokeargument tag
    - the returnvariable should be called "oneCourse"
   --->

   <cfoutput query="oneCourse">
      <p style="font-family:Arial">
         <span style="font-size:18px;font-style:bold">#coursenum# -
#name#</span>
         <br>
         <span style="font-size:12px">#description#</span>
         <br><br>
         <a href="?">Back to List</a>
      </p>
   </cfoutput>
</cfif>
</body>
</html>
```

4. Add two cfinvoke tags that conform to the comments in **CFCs/exercises/usecomponent-temp.cfc.**

Challenge

- Create an additional .cfm page that will access the same CFC but will display the data differently.

A Possible Solution to Exercise 6

As contained in **CFCs/solutions/usecomponent-done.cfm**:

```
<html>
<head>
   <title>Exercise: Using ColdFusion Components</title>
</head>
<body style="font-family:Arial">
<cfif NOT isdefined("coursenum")>
   <cfinvoke component="courseinfo-done" method="getCourses"
returnvariable="courseList" />
   <ul>
      <cfoutput query="courseList">
         <li><a href="?coursenum=#coursenum#">#name#</a></li>
      </cfoutput>
   </ul>
<cfelse>
   <cfinvoke component="courseinfo-done" method="courseDetail"
returnvariable="oneCourse">
      <cfinvokeargument name="coursenum" value="#URL.coursenum#">
   </cfinvoke>
   <cfoutput query="oneCourse">
      <p style="font-family:Arial">
         <span style="font-size:18px;font-style:bold">#coursenum# -
#name#</span>
         <br>
         <span style="font-size:12px">#description#</span>
         <br><br>
         <a href="?">Back to List</a>
      </p>
   </cfoutput>
</cfif>
</body>
</html>
```

As contained in **CFCs/solutions/courseinfo-done.cfc**:

```
<cfcomponent displayname="Course Information" hint="This component
provides functions that ">

   <!--- This function will return all courses --->
   <cffunction name="getCourses" access="public" returntype="query"
displayname="Deliver all the courses" hint="This function displays all of
the courses in the table 'courses'">
      <cfquery name="getTheRecords" datasource="courses">
         SELECT coursenum, name
         FROM courses
      </cfquery>
      <cfreturn getTheRecords>
   </cffunction>
```

```
   <!--- This function has one required argument: coursenum --->
   <cffunction name="courseDetail" access="public" returntype="query"
displayname="Course Detail" hint="This function returns the description
for one course">
      <cfargument name="coursenum" required="true" access="public">
      <cfquery name="getOneRecord" datasource="courses">
         SELECT coursenum, name, description
         FROM courses
         WHERE coursenum = #arguments.coursenum#
      </cfquery>
      <cfreturn getOneRecord>
   </cffunction>

</cfcomponent>
```