

Sample Applications Methodology Guidelines

Purpose

When programming, it is critical that developers are consistent in their coding practices. However, this does not mean that you must always build applications in the same way or that developers must be robots who do everything alike. Rather, each application that you build most likely has different requirements than other applications, and you must adapt your coding practices to each environment.

The methodology guidelines outlined in this document were built specifically for use with the two sample applications shipped with this Getting Started Experience. Although the applications contain many of the best practices recommended by Macromedia, they also emphasize the following:

- ◆ Educating the developer who is exploring the application and code
- ◆ Comments for teaching and reference
- ◆ Readability and consistency to reduce any potential confusion
- ◆ Ease of use to allow developers to take code and reuse it in their own applications

This document is intended to help you better understand how the sample applications were built.

Important: You should not consider this document to be an exact guide for building your own application. It was built for the sample applications with the preceding purposes in mind and should be considered a useful starting point for your own customized coding guidelines. This document is a simplification of the *ColdFusion MX Coding Guidelines* (http://livedocs.macromedia.com/wtg/public/coding_standards/contents.html).

The Guidelines

The following sections describe guidelines for naming, and formatting, and programming conventions in the ColdFusion sample applications.

Naming conventions

The following naming conventions apply to variables:

- ◆ Names should be readable English words or phrases.
- ◆ Names will be mixed-case, starting with lowercase (for example, `firstName`).
- ◆ Underscore characters should not be used.
- ◆ In general, names should not be abbreviated, including for URL variables (for example, use `customerService` instead of `custServ`). An exception to this rule would be “identification” (for example, `userID`).
- ◆ Boolean values should start with `is` if it improves readability (for example, `isEmployee`).
- ◆ Other prefixes on variable names should not be used because they detract from readability (for example, do not use: `aEmployees`, `stEmployees`, `qEmployees`).

The following naming conventions apply to databases and SQL:

- ◆ All names used in the database and SQL statements should be in all uppercase, for consistency and portability with case-sensitive databases.
- ◆ Underscores are permitted to separate words if that improves readability. Do not use underscores outside of databases or SQL; use mixed case.
- ◆ The table name should be plural and descriptive, unless the table contains only one row of data, in which case it should be singular and descriptive (for example, `EMPLOYEES`).
- ◆ Column names should be singular and descriptive (for example, `FIRSTNAME` or `FIRST_NAME`).
- ◆ The table name should not be repeated in the column name unless it makes the column name more readable.
- ◆ Primary keys should be the singular form of the table name followed by “ID”. (for example, `EMPLOYEEID` or `EMPLOYEE_ID`)
- ◆ Names should not be abbreviated.

The following conventions apply to filename:

- ◆ HTML files end in `.html`.
- ◆ CFML files end in `.cfm`.
- ◆ ColdFusion component (CFC) files end in `.cfc`.
- ◆ ColdFusion custom tags end in `.cfm`.
- ◆ XML files end in `.xml`.
- ◆ All filenames are lowercase, with words separated by underscore characters. The exceptions to this rule are that `Application.cfm` and `OnRequestEnd.cfm` must be named with this exact case.
- ◆ Custom tags should be named using lowercase letters and underscore characters.

The following conventions apply to naming ColdFusion components:

- ◆ Component names should be mixed-case, starting with an uppercase letter (for example, `ShoppingCart`).
- ◆ Methods and properties should be named with mixed-case, starting with a lowercase letter (for example, `checkout`).
- ◆ Paths to CFC methods should exactly match the case of the filename and methods (for example, `ShoppingCart.checkout`).

HTML and CFML formatting conventions

The following conventions apply to formatting HTML and CFML:

- ◆ Tags and their attributes should be written in lowercase letters.
- ◆ Do not wrap spaces around equal signs (=).
- ◆ Always use double-quotation marks around attribute values.
- ◆ If an attribute value contains double-quotation marks, you can use single-quotation marks around it. The reverse is also true.
- ◆ Attribute values should be written in lowercase letters unless specific rules, like filenames, override this requirement.
- ◆ Hexidecimal colors should be used with pound signs (#), escaping them when necessary. The following code below shows the hexadecimal color used in the table row (`tr`) tag and proper formatting, with indentation, for a table:

```
<table border="0" cellpadding="0" cellspacing="0">
<tr bgcolor="#999999">
    <td valign="top" align="right">
        
    </td>
</tr>
</table>
```

- ◆ ColdFusion functions should be written in mixed-case, with the first letter in lowercase.
- ◆ Built-in ColdFusion operators (`is`, `and`, `or`, `not`) should be written in lowercase.
- ◆ Number signs (#) should not be used around variables within ColdFusion tags or functions unless inside an attribute value. The following code shows various ColdFusion code formatting:

```
<cfif compareNoCase(variables.companyName, "Macromedia")>
    ...
</cfif>

<cfset firstName=queryname.firstName />

<cfquery name="employeeSelect" datasource="#request.dsn#">
    ...
</cfquery>
```

- ◆ CFCs should be formatted similarly:

```

<cfcomponent hint="...">

<cffunction name="checkOut" returnType="boolean" hint="...">

    <cfargument name="cart" type="struct" hint="..." />

    <cfset var cartInsert="" />
    <cfset var isSuccessful=true />

    <cftry>
        <cfquery name="cartInsert" datasource="#request.dsn#">
            ...
        </cfquery>

        <cfcatch type="database">
            <cfset isSuccessful=false />
        </cfcatch>
    </cftry>

    <cfreturn isSuccessful />

</cffunction>

</cfcomponent>

```

◆ SQL queries should be formatted with the following styles and cases:

```

SELECT FIRSTNAME, LASTNAME
FROM EMPLOYEES
WHERE EMPLOYEEID =
    <cfqueryparam cfsqltype="cf_sql_numeric" value="#url.employeeID#">
AND DEPARTMENTID = <cfqueryparam cfsqltype="cf_sql_numeric"
value="#url.departmentID#">
INSERT INTO EMPLOYEES
    (FIRSTNAME, LASTNAME, DEPARTMENTID)
VALUES
    (
        <cfqueryparam cfsqltype="cf_sql_varchar" value="#form.firstName#">
        , <cfqueryparam cfsqltype="cf_sql_varchar" value="#form.lastName#">
        , <cfqueryparam cfsqltype="cf_sql_numeric" value="#form.departmentID#">
    )
UPDATE EMPLOYEES
SET
    FIRSTNAME =
        <cfqueryparam cfsqltype="cf_sql_varchar" value="#form.firstName#">
    , LASTNAME =
        <cfqueryparam cfsqltype="cf_sql_varchar" value="#form.lastName#">
    , DEPARTMENTID =
        <cfqueryparam cfsqltype="cf_sql_numeric" value="#form.departmentID#">
WHERE EMPLOYEEID =
    <cfqueryparam cfsqltype="cf_sql_numeric" value="#form.employeeID#">

DELETE FROM EMPLOYEES
WHERE EMPLOYEEID =
    <cfqueryparam cfsqltype="cf_sql_numeric" value="#form.employeeID#">

```

The following are exceptions to the formatting conventions:

- ◆ The `cfset` tag attribute values do not need to be surrounded by double-quotation marks, because they are often variables. If they are strings, double-quotation marks should be used.

File structure conventions

- ◆ The main application should be stored in the following file structure. Notice that `appname` represents the name of your application and `cfmxroot` refers to the root ColdFusion directory (often: `c:\cfusionmx`).

```
{cfmxroot}/wwwroot/appname/
{cfmxroot}/wwwroot/appname/db
{cfmxroot}/wwwroot/appname/extensions
{cfmxroot}/wwwroot/appname/extensions/components
{cfmxroot}/wwwroot/appname/extensions/customtags
{cfmxroot}/wwwroot/appname/extensions/includes
{cfmxroot}/wwwroot/appname/webroot
{cfmxroot}/wwwroot/appname/webroot/images
{cfmxroot}/wwwroot/appname/webroot/shared
{cfmxroot}/wwwroot/appname/webroot/shared/css
{cfmxroot}/wwwroot/appname/webroot/shared/js
```

There are `wwwroot` and `webroot` directories. Server administrators often redirect domain names on a server to point to different applications. In this case, the domain name would point to the `webroot`, where all the CFM pages and images are stored. The benefit of this method is that your extensions (components, custom tags, includes) and database are more secure, because they are not in the `webroot` directory and are less prone to outside attacks.

- ◆ A custom tag path must be registered in the ColdFusion MX 7 Administrator:

```
{cfmxroot}/wwwroot/appname/extensions/components
```

- ◆ Create a ColdFusion mapping called `appname_includes` that points to the following:

```
{cfmxroot}/wwwroot/appname/extensions/includes
```

- ◆ The database files for each application should be placed in the following folder:

```
{cfmxroot}/wwwroot/appname/db
```

- ◆ Application subdirectories should be placed in the application directory:

```
{cfmxroot}/wwwroot/appname/webroot/reports/
```

- ◆ Shared resources should be placed in a `shared` directory:

```
{cfmxroot}/wwwroot/appname/webroot/shared/images/
{cfmxroot}/wwwroot/appname/webroot/shared/documents/
{cfmxroot}/wwwroot/appname/webroot/shared/css/
{cfmxroot}/wwwroot/appname/webroot/shared/js/
```

- ◆ CFCs accessible as web services or those that use Macromedia Flash Remoting (for example, the `cffunction` tag specifies `access="remote"`) should be stored in the shared resources:

```
{cfmxroot}/wwwroot/appname/webroot/shared/services/
```

The virtual mapping to the `webroot` directory defined previously is used to access this path.

- ◆ All other CFCs should be stored outside of the web root directory:

```
{cfmxroot}/wwwroot/appname/extensions/components
```

Components in this folder should be stored in subfolders, as appropriate.

Coding conventions

The following conventions apply to programming with CFML:

- ◆ All variables should be scoped and the scope should be in lowercase (ex: `url.variableName`).
- ◆ Global variables should be set in the `Application.cfm` file as request-scoped variables.

```
<!---//-----
Request-scoped variables will persist through the entire page process; therefore
they are a good choice for "global" application variables. Store them in the
Application.cfm file to ensure that every page process can see them.
-----//---->
<!--- set datasource name created in the ColdFusion Administrator --->
<cfset request.dsn="HumanResources">

<!--- set absolute path to documents directory --->
<cfset request.documentPath="c:\cfusionmx\wwwroot\intranet\shared\documents">
```

- ◆ Boolean values should always be denoted as `true` or `false`, never 1 or 0.
- ◆ Boolean values should never be tested against another value.

```
<cfif isEmployee>
    ...
</cfif>

NOT

<cfif isEmployee is "true">
    ...
</cfif>
```

- ◆ Use the `cfswitch` tag instead of the `cfif` tag when evaluating one expression against more than two values.
- ◆ Always include error trapping around mission-critical code and code that calls external services (like a database, mail server, and so on).
- ◆ Only use the `cflock` tag to avoid race conditions (where two concurrent requests could access and/or update the same data).
- ◆ For CFCs, always specify the `type` attribute for the `cfargument` tag. Avoid using the value `any`.
- ◆ All CFCs should define a method called `init()` that initializes the instance. The method should have a return type that matches the component and it should return `this` rather than `void`. The following example is for a CFC named `cart.cfc`:

```
<!---//-----
All components should have an init() method even if it does nothing more than
return this. Doing so ensures that you are following good coding practices
established in other languages, like Java. Research "constructors" for more
information.
-----//---->
<cfcomponent hint="Shopping cart component">
    <cffunction name="init" access="public" returnType="cart" output="false">
        ... arguments as necessary ...
        ... perform initialization ...

        <cfreturn this />
    </cffunction>
</cfcomponent>
```

- ◆ Choose one way to call a CFC:

```
<!---//-----
```

There are multiple ways to call your components. One way would be to use the `cfobject` and `cfinvoke` tags as done in the following code. You might also see the object creation and the invoking of the `init()` method rolled into one statement, like this:

```
<cfset cfcCart = createObject("component","cart").init() />
-----//---->

<cfobject component="cart" name="cfcCart" />

<cfinvoke component="#cfcCart#" method="init" returnvariable="cfcCart" />

<cfinvoke component="#cfcCart#" method="update" returnvariable="isSuccessful">
    ...
</cfinvoke>
```

- ◆ All custom tags should anticipate being executed in both start and end modes.

```
<cfswitch expression="#thisTag.executionMode#">
<cfcase value="start">
    ...
</cfcase>
<cfcase value="end">
    ...
</cfcase>
```

- ◆ Use query caching to improve performance but only in the more complex areas of the sample applications. Keep queries simple for beginners.
- ◆ Use the `cfqueryparam` tag to increase query performance.

```
<cfquery name="employeeSelect" datasource="#request.dsn#">
SELECT FIRSTNAME, LASTNAME
FROM EMPLOYEES
WHERE EMPLOYEEID =
    <cfqueryparam cfsqltype="cf_sql_numeric" value="#form.employeeID#">
</cfquery>
```

- ◆ For large query results, use the `blockFactor` attribute of the `cfquery` tag to increase query performance, if appropriate.
- ◆ Use the `compareNoCase()` function to compare two values. Remember that this function returns 0 if the values match, so it corresponds to `is not`.

```
<cfif compareNoCase(url.employeeID, 10)>
```

- ◆ Use the `listFindNoCase()` function to compare one item to a list.

```
<cfif listFindNoCase(employeeList, "10")>
```

- ◆ Use arrays instead of lists, but do not bother converting lists to arrays for simple comparisons.
- ◆ Do not use the `cfmodule` tag to call custom tags. Instead, use the filename of the custom tag without the file extension and prefixed with `cf_` (for example, `cf_tagname`) to access custom tags that are stored in the custom tag path, as defined previously.
- ◆ If the sample applications use the same custom tag names, use the `cfimport` tag and a relative path to call them.
- ◆ Try to avoid the use of the `evaluate()` function.

```
caller["var_" & i]
variables["var_" & i]
```

- ◆ Always use `structure.key` or `structure[key]` instead of `structFind(structure, key)`.
- ◆ Always use `x=x+1` instead of `x=incrementValue(x)`.

Commenting conventions

The following conventions apply to writing comments in your code:

- ◆ Use ColdFusion comments whenever possible.
- ◆ Each file must begin with a section of comments about the page; for example:

```
<!---//-----
Author: Emily Kim (emily@trilemetry.com)
Creation Date: 06/03/04
Copyright: (c) 2004 Macromedia, Inc.

Purpose: to display one Employee's detailed information.

Input:

employeeID (required)

Revision Log:

6/04/04 - ekim - modified display formatting.
-----//-->
```

- ◆ For the purposes of this Experience, you should over-comment code with explanations, because these sample applications will be used as a learning guide.
- ◆ Comment for formatting clarity, even if logical clarification is not necessary.

```
<html>

<head>
  <title>Getting Started Experience</title>
</head>

<body>

<!--- start logo table --->
<table border="0" cellpadding="5" cellspacing="0">
<tr>
  <td>
    
  </td>
</tr>
</table>
<!--- end logo table --->

<!--- start main content table --->
<table border="0" cellpadding="5" cellspacing="0">
<tr>
  <td>
    Some text here.
  </td>
  <td>
    <!--- start nested table for form --->
    <form action="form_action.cfm" method="post">
    <table border="0" cellpadding="5" cellspacing="0">
    <tr>
      <td align="right">
        First Name:
```

```

                </td>
                <td>
                    <input type="text" name="firstName" size="20" />
                </td>
            </tr>
            <tr>
                <td align="right">
                    Last Name:
                </td>
                <td>
                    <input type="text" name="lastName" size="20" />
                </td>
            </tr>
        </table>
    </form>
    <!-- end nested table for form -->
</td>
</tr>
</table>
<!-- end main content table -->

</body>

</html>

```

- ◆ Comments should be liberally added for conditional logic explanations.
- ◆ The `hint` attributes should be completed for all ColdFusion `cfcomponent`, `cffunction` and `cfargument` tags.

XHTML compliance

Both ColdFusion and HTML tags should follow these rules whenever possible. The `cfelse`, `cfset`, and `cfmodule` tags are exceptions.

- ◆ Element and attribute names must be lowercase.
- ◆ For nonempty elements, end tags are required (for example, `<p>paragraph text</p>`).
- ◆ Empty elements must have an end tag or the start tag must end with `/>`. Ensure that you place an extra space before the forward slash (for example, `
`).
- ◆ Attribute values must always be enclosed in double-quotation marks.

508 compliance

The following guidelines apply to writing accessibly sample applications:

- ◆ All generated HTML must pass Section 508 accessibility guidelines (http://www.macromedia.com/macromedia/accessibility/508_guidelines.html and <http://www.section508.gov/index.cfm?FuseAction=Content&ID=12#Web>).
- ◆ A text equivalent for every nontext element must be provided using `alt`, `longdesc`, or in element content.
- ◆ Equivalent alternatives for any multimedia presentation must be synchronized with the presentation.
- ◆ Web pages shall be designed so that all information conveyed with color is also available without color (for example, from context or markup).
- ◆ Documents shall be organized so they are readable without requiring an associated style sheet.
- ◆ Redundant text links shall be provided for each active region of a server-side image map.
- ◆ Client-side image maps shall be provided instead of server-side image maps except where the regions cannot be defined with an available geometric shape.
- ◆ Row and column headers shall be identified for data tables.
- ◆ Markup shall be used to associate data cells and header cells for data tables that have two or more logical levels of row or column headers.

- ◆ Frames shall be titled with text that facilitates frame identification and navigation.
- ◆ Pages shall be designed to avoid causing the screen to flicker with a frequency greater than 2 Hz and lower than 55 Hz.
- ◆ A text-only page, with equivalent information or functionality, shall be provided to make a website comply with the provisions of this part, when compliance cannot be accomplished in any other way. The content of the text-only page shall be updated whenever the primary page changes.
- ◆ When pages utilize scripting languages to display content, or to create interface elements, the information provided by the script shall be identified with functional text that can be read by assistive technology.
- ◆ When a web page requires that an applet, plug-in, or other application be present on the client system to interpret page content, the page must provide a link to a plug-in or applet that complies with §1194.21(a) through (l).
- ◆ When electronic forms are designed to be completed online, the form shall allow people using assistive technology to access the information, field elements, and functionality required for completion and submission of the form, including all directions and cues.
- ◆ A method shall be provided that permits users to skip repetitive navigation links.
- ◆ When a timed response is required, the user shall be alerted and given sufficient time to indicate that more time is required.