



Alaska Department of Natural Resources
Division of Support Services
Information Resource Mangement
550 W. 7th Ave. Suite 706
Anchorage, AK 99501

ColdFusion Coding Standards

For DNR Web Applications

Table of Contents

INTRODUCTION	2
COMMON CODE LOCATIONS	2
<i>Java Class files</i>	2
<i>Java Jar files</i>	2
<i>Custom Tags</i>	2
<i>Server-wide files</i>	2
CODE REPOSITORY	2
DOCUMENTING CODE AND APPLICATIONS	3
<i>General Guidelines</i>	3
<i>CFC/Template Document Header</i>	3
APPLICATION.CFC FILES	4
<i>Server Level Variables</i>	4
ERROR HANDLING AT THE CODE LEVEL	5
<i>Using cftry/cfcatch</i>	5
<i>onError() Method</i>	6
USE OF FRAMEWORKS	7
CODING FOR MULTIPLE ENVIRONMENTS	7
WHEN TO USE THE SECURE SERVER	8
NAMING & CONVENTIONS	8
<i>Abbreviations</i>	8
<i>Acronyms</i>	8
<i>Package Names</i>	8
<i>Class/Component/Interface Names</i>	9
<i>Methods</i>	9
<i>Type Names</i>	9
<i>CFML Tags, Custom Tags and Attributes</i>	10
<i>Arguments and Variables</i>	10
<i>Constants or Static Variables</i>	11
USE CFLOCK ON SHARED RESOURCES	11
<i>Race Conditions</i>	12
VARIABLE SCOPING	12
DO NOT ABUSE POUND SIGNS	13
USE CONSISTENT CODE FORMATTING	13
COLDFUSION CODE PROTECTION BEST PRACTICES	14
<i>Introduction</i>	14
<i>Golden Rule of Web Applications</i>	14
<i>Understanding HTTP Methods</i>	14
<i>Protecting Your Code</i>	15
<i>OWASP Data Validation and Interpreter Injection</i>	15
SQL Injection	15
LDAP Injection	16
XML Injection	16
Event Gateway, IM, and SMS Injection	16
Best Practices	16
Best Practice in Action	17
ADDITIONAL RESOURCES	18

Introduction

This document describes the coding practices for new ColdFusion applications within DNR. The purpose of these coding practices is to ensure some level of coding consistency, to allow code to be reused whenever possible and to allow other developers to more quickly understand and work with your code. Additionally it provides information on how to help prevent negatively impacting the server environment.

These guidelines keep in mind that many of our ColdFusion applications are not purely ColdFusion coding. They may also leverage the best technologies for the job, such as Java or XML, with ColdFusion tying the various technologies together.

These guidelines are based on Macromedia's ColdFusion MX Coding Guidelines available online at: http://livedocs.adobe.com/wtg/public/coding_standards/ and on ColdFusion Standards & Best Practices: <http://ortus.svnrepository.com/coldbox/trac.cgi/wiki/cbDevelopmentBestPractices>

Common Code Locations

Java Class files

Compiled Java class files are stored in `/shared/classes/`.

The corresponding `.java` files should be stored in the code repository with the project they were created for.

Java Jar files

Java Jar files are stored in `/shared/lib/`.

Custom Tags

ColdFusion custom tags can be stored either in the directory in which they are called. If being called by multiple applications, use `/shared/customtags`. Make sure there are no naming conflicts with existing files before placing new custom tags within the directory.

Server-wide files

Server-wide files for things like a generic email processing form are in directories under the `/shared` directory.

Code Repository

It is important to have past versions of an application available should there be a need to rollback to a prior version. Before you make a change to an application, copy the existing application to the subversion repository.

Documenting Code and Applications

This section provides guidelines on commenting your source code. In general, we should comment code to assist other developers who work on it in the future. We do not want our comments to be visible to the public so we do not want to generate HTML comments from CFML - we use `<!-- ... -->` in CFML which does not get published into the HTML. Comments are there to be read - consider your audience!

General Guidelines

Write CFML style `<!-- ... -->` comments, for all **important** entities, that describe **what** code does and **why** - document the **how** if it is not obvious.

When you make a change, comment it. Identify the change with the date and your user name:

```
<!-- 2010-11-26 fmlast Expanded the Comments section -->
```

When you want to leave a note about a bug to be fixed or functionality to be added, put **TODO:** in front of the actual comment so developers can easily search for them:

```
<!-- 2010-11-26 fmlast TODO: Incorporate everyone's feedback -->
```

Additional standard search keywords can be added after **TODO:**, **FIXME:**, **NOTE:** - this is very important as it helps your audience, other developers. Furthermore, standard tags like this can be read by code editors such as Eclipse to create a "task list" whenever you're working on a file.

```
<!-- 2010-11-26 fmlast TODO: BUG: Fails on Fridays -->
```

CFC/Template Document Header

Each CFML file should begin with an CFML style `<!-- ... -->` comment. This ensures the filename and application information are not rendered to HTML and viewable via the web browser. This comment contains the filename and a standard copyright message followed by an explanation of the file and then, optionally, its modification history:

```
<!--
filename.cfm
Copyright (c) yyyy State of Alaska Department of Natural Resources
Application:      Name_of_application
Purpose:          Short purpose of this file
Documentation:    Location of external documentation
-----
```

```

Author:           Author's Name
Description:      Brief description of file's functionality
Parameters:      Expected parameters passed into this file
                  Parameters passed from this file to the next file
Dependencies:    Things know to depend on or use this file
                  Things this file depends on

Revision History:
Date             Author  Comments
MM/DD/YYYY      FMLast  Summary of changes
-----
-->

```

Application.cfc Files

When creating applications that connect to another system (Oracle database, mainframe, Stellent server, etc), you need to use an Application.cfc file. If Application.cfc as well as Application.cfm and/or OnRequestEnd.cfm are all present, Application.cfm and OnRequestEnd.cfm are ignored by the application.

Application.cfc introduced a number of built in methods that handle specific events. The Application.cfc is the best location to include error checking for remote server availability. Failure to check the status of remote servers can cause your application to hang and can eventually bring down ColdFusion if left unchecked.

Server Level Variables

The DNR web servers have ColdFusion scheduled tasks that run every 5 minutes to check the status of backend systems. These scheduled tasks set or create server level Boolean variables for programmers to check within their Application.cfc files. This allows for error handling at the application, session or request levels if the backend systems are not available.

The server level boolean variables are:

Server Variable Name	Tests
server.isMainframeAlive	Can we reach the mainframe?
server.isBrokerAliveRO	Is the RO broker job running?
server.isBrokerAliveLAS	Is the LAS broker job running?
server.isTaminoAlive	Can we reach the Tamino XML database?
server.isDatabaseAlive	Can we reach the DNR Oracle databases?
server.isEftAlive	Can we reach the Electronic Funds Transfer server?
server.isStellentAlive	Can we reach the Stellent server?

These variables can be checked for within different areas of your Application.cfc to verify if the needed service is reachable or running. If it is not and the server variable is false, you will want to prevent the application from running. You can display a custom error message or possibly use a generic site-wide error message when the service is unavailable.

For more information on Application.cfc files, see the document “ColdFusion Application.cfc Files” under the ColdFusion section of the webmaster’s web site: <http://int.dnr.alaska.gov/shared/webmasters> . Also see Chapter 6, “Defining the application and its event handlers in Application.cfc” section in the Developing ColdFusion 9 Applications guide at http://int.dnr.alaska.gov/shared/webmasters/coldfusion9/coldfusion_9_dev.pdf .

Error Handling at the Code Level

Error handling at code level allows the programmer to

- control the exception error handling
- keep an application running by recovering gracefully from an exception error
- keep an exception from negatively impacting server performance
- give a customized error message to the user
- retry an operation in case of a temporary problem and
- create error handling exceptions that are custom to the application (like InvalidLASCode, etc.).

Code level error handling in ColdFusion is done with the `cftry/cfcatch` tags. The `cftry/cfcatch` tags should be used with all

- calls to a database
- file handling operations
- calls to other services (web services, broker calls, etc) not controlled by the application.

Using `cftry/cfcatch`

From the ColdFusion 8 documentation on using `cftry/cfcatch` tags:

“The `cftry` tag lets you go beyond reporting error data to the user:

- You can include code that recovers from errors so your application can continue processing without alerting the user.
- You can create customized error messages that apply to the specific code that causes the error.

For example, you can use `cftry` to catch errors in code that enters data from a user registration form to a database. The `cfcatch` code could do the following:

1. Retry the query, so the operation succeeds if the resource was only temporarily unavailable.
2. If the retries fail:
 - Display a custom message to the user
 - Post the data to an email address so the data could be entered by company staff after the problem has been solved.”

Additionally there are standard `cfcatch` and `onError` variables that allow you to obtain error information.

The method `onError()` in `Application.cfc` will allow an application to handle errors that are not handled by individual `try/catch` blocks.

onError() Method

The onError() method in ColdFusion provides a way to handle any exceptions that are not handled by individual try/catch blocks. It is placed in the Application.cfc

```
<cffunction name="onError" access="public" returntype="void">
</cffunction>
```

The onError() method takes two arguments. EventName, which is a string and Exception, which is a structure. The EventName will be one of the following:

```
onApplicationStart
onSessionStart
onRequestStart
onRequest
onRequestEnd
onApplicationEnd
onSessionEnd
```

With the onApplicationEnd and onSessionEnd events the application is not able to display an error message to the user, however the application is able to log errors that occur during these events.

The method should look like the following:

```
<cffunction name="onError" access="public" returntype="void">
<cfargument name="Exception" required="true" type="struct" />
<cfargument name="EventName" required="true" type="string" />
</cffunction>
```

The application can send an email or create an entry in a log when an error occurs as well as show a message to the user.

In order to display a simple error message to the user the application should check to make sure the event is not onSessionEnd or onApplicationEnd.

Once this is done an error message can be coded in the function or included using <cfinclude>.

The final onError() method should look as follows:

```
<cffunction name="onError" access="public" returntype="void">
<cfargument name="Exception" required="true" type="struct" />
<cfargument name="EventName" required="true" type="string" />

<cfif NOT (Arguments.EventName IS "onSessionEnd") OR
(Arguments.EventName IS "onApplicationEnd")>
<cfinclude template="globalErrorTemplate.cfm" />
</cfif>

</cffunction>
```

ColdFusion has made implementing a global error handler easy with the use of the onError() method.

For more information on error handling see Chapter 6, “Handling Errors” section of the Developing ColdFusion 9 Applications guide at http://int.dnr.alaska.gov/shared/webmasters/coldfusion9/coldfusion_9_dev.pdf .

Use of Frameworks

The use of frameworks for coding ColdFusion applications is strongly encouraged. It assists in productivity and maintainability of code. Most ColdFusion applications within DNR have utilized the FuseBox framework. Information on FuseBox is available at <http://fusebox.org>.

If FuseBox does not work for your application, consider another Model-View-Controller framework. A number of frameworks are reviewed in this Adobe Developer’s Network article, An introduction to ColdFusion Networks (http://www.adobe.com/devnet/coldfusion/articles/frameworks_intro.html).

Coding for Multiple Environments

Our applications often connect to databases or mainframes with their own development/test/production environments. Best practices would have our test environment connect to the database test environment. By coding your application to “sniff” out the current environment, you can have your database or mainframe connections automatically select the correct environment to connect to.

Using the ColdFusion variable CGI.SERVER_NAME (or the older CGI.HTTP_HOST) you can test to see which environment you are in and set up the appropriate environment-related variables.

For example, to set the variable isTest based on which system it is on, the code might look like this:

```
<cfswitch expression="#CGI.HTTP_HOST#">
  <cfcase value="intdev.dnr.alaska.gov">
    <cfset isTest = "true"/>
  </cfcase>
  <cfcase value="int.dnr.alaska.gov">
    <cfset isTest = "false"/>
  </cfcase>
  <cfcase value="dev.dnr.alaska.gov">
    <cfset isTest = " true "/>
  </cfcase>
  <cfcase value="test.dnr.alaska.gov">
    <cfset isTest = " true "/>
  </cfcase>
  <cfcase value="dnr.alaska.gov">
    <cfset isTest = " false "/>
  </cfcase>
  <cfdefaultcase>
    <cfset isTest = " false "/>
  </cfdefaultcase>
</cfswitch>
```

When to use the Secure Server

In conjunction with the standard public Internet web site, is a secure, or encrypted, web server. It has a separate document home from the public Internet web site (see the handout on File Organization in the DNR Webmaster's Center).

The secure server site is used when handling confidential data such as credit card or social security numbers or when asking for username/password information and authenticating it against an ETS server.

Naming & Conventions

Use good names for components, methods, arguments and local variables. Naming is very important and will most of the time document your code. Always remember to use meaningful names and stay away from cryptic abbreviations or naming strategies.

Abbreviations

AVOID abbreviations if possible. For example, *calculateSalary()* is a better method name than *calcSalary()*. Although you can use well known abbreviations, please try to avoid them if possible.

Acronyms

Acronyms should be avoided in names, but if they must be used, then all acronyms must be capitalized no matter where they are located on a string name.

```
-- DO THIS --
URLScanner.cfc
parseHTTPString()

-- NOT THIS --
url-scanner.cfc
UrlScanner.cfc
parseHttpString()
ParseHttpString()
```

Package Names

Package names should be unique and in lowercase letters. Underscores may be used or hiphens if necessary. You can package your objects/files using two well known approaches:

- 1) By Functionality (Best Practice)
- 2) By object types

The best practice is to use packaging by functionality if at all possible. This creates better packaging layout and maintainability. Here is an example from an application's model or business layer folder:

```
+ model
  + security
  + remote-api
  + products
  + users
  + conversions
  + util
```

Class/Component/Interface Names

Class/Component/Interface names should be nouns, as they represent most likely things or objects. They should be written in camel case with only the first letter capitalized for each word. Use whole words and avoid acronyms and abbreviations if possible.

Examples:

```
-- DO THIS --
URLConverter
RSSReader
Serializable
ISearchEngine

-- NOT THIS --
urlConverter
rssreader
serializable
iSearchEngine
```

Methods

Methods should be verbs, in mixed camel case with the first letter lower cased and then each internal first letter of words capitalized. Examples:

```
-- DO THIS --
run()
doThis()
executeInBackground()
isLocated()

-- NOT THIS --
RUN()
dothis()
executeINBackGround()
ISLocated()
```

Type Names

All ColdFusion type names in arguments, return types and the like should all be in lower case when they are native ColdFusion types. If they are components they should be the

EXACT name of the component. This is extremely important if for some reason the code executes in a case-sensitive system, then the code will not work. ALWAYS have the exact case of components and definitions.

```
-- DO THIS --
<cfargument name="paths" type="array" >
<cfargument name="user" type="model.users.User">
<cffunction name="getSecurityService"
returnType="model.security.SecurityService">

-- NOT THIS --
<cfargument name="paths" type="ARRAY" >
<cfargument name="user" type="model.users.user">
<cffunction name="getSecurityService"
returnType="model.security.SECURITYSERVICE">
```

CFML Tags, Custom Tags and Attributes

All CFML and custom tags should be writing in lower case form, just like HTML tags. Attributes for CFML tags should follow the same behavior as arguments and variables as seen below. If attributes can all be placed in one line, then do that. However, if they will span and cause breaks, consider breaking the attributes into multiple lines and aligning them to the first attribute.

```
-- DO THIS --
<cfhttp url="...">
<cfabort>
<cfdump var="#session#">
<cfhttp url="#urladdress#" method="GET"
resolveurl="Yes" throwOnError="Yes"/>

-- NOT THIS --
<CFHTTP>
<CFABORT>
<CFDump Var="#session#">

-- Unecessary Multi Line --
<cfhttp url="#urladdress#"
method="GET"
resolveurl="Yes"
throwOnError="Yes"/>
```

Arguments and Variables

They should be descriptive lowercase single words, acronyms or abbreviations. If multiple words are necessary they should follow camel case with first letter lowercase. Examples:

```
-- DO THIS --
niceLocation = "Miami";
results = "";
avgSalary = "323";
```

```
-- NOT THIS --
NICELOCATION = "Miami";
Results = "";
average-salary = "323";
```

Constants or Static Variables

They should all be in upper case separated by underscores "_". Examples:

```
-- DO THIS --
INTERCEPTOR_POINTS = "";
LINE_SEP = "-";
MAX = "123";

-- NOT THIS --
interceptor-points = "";
line_sep = "d";
max = "123";
```

Use *cflock* On Shared Resources

Use *cflock* whenever you need to make your code thread safe. This applies to variables in shared scopes such as: *server and application* scope. You sometimes want to even lock *session* scope if you are working with framesets, but usually locking *session* scope is not necessary anymore. Also remember to use *cflock* whenever you are accessing shared resources, such as file operations, cache operations, etc.

- Always use a *timeout* and *throwOnTimeout* attributes on the *cflock* tag.
- If you use **exclusive** locks on a resource, make sure that you also provide **readonly** locks when trying to read from such resources.
- Use named locks for locking resources that do not apply to scopes such as *server, application, session*. However, please understand that the name of the lock is on a **per server** basis. So make sure the name is unique enough so other applications running on the same server do not collide with it. If they do, you will be providing unnecessary bottlenecks as named locks are global.
- Good locking article:
http://www.adobe.com/devnet/server_archive/articles/cf_locking_best_practices.html
- Do not overinflate the code within lock tags. Locking code should only occur on small bits of code and when you are accessing the shared resource. Of course, there are special occasions to do more than just saving in shared scope, but use it as a rule of thumb.

Race Conditions

There will be cases where you need to do a double test in order to avoid race conditions on shared resources. This strategy can be applied when you need to test, for example, if a resource is created, an object is configured, etc. What this strategy does is provide two if statement criterias that can verify behavior on the resource, squished between a cflock tag. This prevents threads that have already entered the locking stage and are waiting execution, to re-execute the locked code.

```
-- DO THIS --
<cfif structKeyExists(application,"controller")>
  <cflock name="mainControllerCreation" timeout="20"
throwOnTimeout="true" type="exclusive">
<cfif structKeyExists(application,"controller")>
  <cfset application.controller =
createObject("component","coldbox.MainController").in
it()>
</cfif>
</cflock>
</cfif>

-- NOT THIS --
<cfif structKeyExists(application,"controller")>
<cflock name="mainControllerCreation" timeout="20"
throwOnTimeout="true" type="exclusive">
  <cfset application.controller =
createObject("component","coldbox.MainController").in
it()>
</cflock>
</cfif>
```

As you can see from the previous code snippet, if you do not have the double if statements, then code that is waiting on the lock, will re-execute the creation of the controller object. Therefore, since we can test the resource state, we can provide a multi-thread safety net.

Variable Scoping

ColdFusion variables must be scoped according to where they are created and located unless for good dynamic reasons. This will improve performance and readability when diagnostics are needed. The default scope that can be omitted is the *variables* scope, which is by default implied. Even scoping variables/columns in queries is mandatory to avoid collisions.

```
-- DO THIS --
<cfoutput>#url.name#</cfoutput>
<cfoutput query="qCountries">
  <li>#qCountries.name#</li>
</cfoutput>
```

```
-- NOT THIS --
<cfoutput>#name#</cfoutput>
<cfoutput query="qCountries">
  <li>#name#</li>
</cfoutput>
```

Do Not Abuse Pound Signs

Pound signs are most often used to output variables to their set values or evaluate them. There are many places where you DO NOT need to place hash signs. This only delays the evaluation and is not best practice. Most likely you will only need to use pound signs when using *cfoutput* or when dealing with certain tag attributes that require the evaluation of a variable.

```
-- DO THIS --
<cfset name = request.firstname>
<cfif isValid></cfif>
<cfset SomeVar = Var1 + Max(Var2, 10 * Var3) + Var4>

-- NOT THIS --
<cfset name = #request.firstname#>
<cfif #isValid#></cfif>
<cfset #SomeVar# = #Var1# + #Max(Var2, 10 * Var3)# +
#Var4#>
```

Use Consistent Code Formatting

Try to always use tabs and spacing correctly when spacing code and formatting it. Always indent your tags when they are nested, it provides readability and consistency.

```
-- DO THIS --
<cfif isValid>
  <cfset test = luis>
</cfif>
  <cffunction name="getValue"
              access="public"
              returnType="any"
              output="false">
    <cfreturn test>
  </cffunction>

-- NOT THIS --
<cfif isValid>
<cfset test = luis>
</cfif>

<cffunction name="getValue" access="public"
returnType="any" output="false">
<cfreturn test>
```

```
</cffunction>
```

ColdFusion Code Protection Best Practices

Introduction

This section delineates some best practices when dealing with SQL injection attempts or just plain old URL/FORM variable manipulations, when building ColdFusion web applications.

Golden Rule of Web Applications

Never ever ever trust the incoming data. It is YOUR responsibility to protect your code.

Understanding HTTP Methods

First of all, you also need to understand what a POST, GET, DELETE, PUT are used for. The method attribute of the FORM element specifies the HTTP method used to send the form to the processing agent. This attribute may take the following values:

GET: With the HTTP "get" method, the form data set is appended to the URI specified by the action attribute (with a question-mark ("?") as separator) and this new URI is sent to the processing agent.

POST,PUT,DELETE: The form data set is included in the body of the form and sent to the processing agent, with expectations of either a save, update or delete.

The *GET* method should be used when the form is idempotent (i.e., causes no side-effects). Many database searches have no visible side effects and make ideal applications for the *GET* method. *If the service associated with the processing of a form causes side effects (for example, if the form modifies a database or subscription to a service), the "post, put or delete" method should be used.*

The most important fact is that the *GET* method should be used for idempotent transactions. Here is the definition for idempotent:

"Idempotent operation means that it can be repeated without causing any errors or inconsistencies if the operation is carried out once or many times".

Thanks to Roger Benningfield:

"Allowing GET requests to change the state of server resources can be a very dangerous game, without so much as a whiff of malicious behavior. An app that allows clients to change or delete data just by fetching a URI is asking for trouble in 2006."

There is a time for using *GET* and a time for using *POST,PUT,DELETE* and **ALL OF THEM should not trust the client data.**

Protecting Your Code

As for security, form variables are just as easy to modify as URL variables. However, there are several ways to protect from attacks, SQL injection or plain mischief:

- 1) <CFPARAM> tag with strict data-type and scaling will prevent URL and FORM variable abuse. This tag binds incoming FORM or URL variables to specific types and even create default values for them.
- 2) Use of <CFQUERYPARAM> is the most common form of stopping URL/FORM variable abuse with direct SQL and “it makes your queries faster as well (If your db supports it). Use this tag to protect any part of your SQL statements that come from an external source, even ORDER BY or GROUP BY statements.
- 3) Get into the habit of adding the maxlength attribute to cfqueryparam’s, to limit the number of characters/digits to bind a column with.
- 4) Use of Stored Procedures will prevent SQL injection as all variables are bounded by the use of <CFPROCparam>.
- 5) Try to always use a custom error page that cannot display to the user the entire error message. This will hide your internal information and encapsulate your errors.
- 6) It is the role of the developer to protect the database and its contents, no matter if you are using a POST or a GET.
- 7) Do not trust the client on the incoming data; always do data type checking and authorization/authentication checking on the server-side.

OWASP Data Validation and Interpreter Injection

This section is taken from the Open Web Application Security project (OWASP) Data Validation at http://www.owasp.org/index.php/Data_Validation .

This section focuses on preventing injection in ColdFusion. Interpreter Injection involves manipulating application parameters to execute malicious code on the system. The most prevalent of these is SQL injection but it also includes other injection techniques, including LDAP, ORM, User Agent, XML, etc. As a developer you should assume that all input is malicious. Before processing any input coming from a user, data source, component, or data service it should be validated for type, length, and/or range. ColdFusion includes support for Regular Expressions and CFML tags that can be used to validate input.

SQL Injection

SQL Injection involves sending extraneous SQL queries as variables. ColdFusion provides the <cfqueryparam> and <cfprocparam> tags for validating database parameters. These tags nests inside <cfquery> and <cfstoredproc>, respectively. For dynamic SQL submitted in <cfquery>, use the CFSQLTYPE attribute of the

<cfqueryparam> to validate variables against the expected database datatype. Similarly, use the CFSQLTYPE attribute of <cfproccparam> to validate the datatypes of stored procedure parameters passed through <cfstoredproc>.

LDAP Injection

LDAP injection is an attack used to exploit web based applications that construct LDAP statements based on user input. ColdFusion uses the <cfldap> tag to communicate with LDAP servers. This tag has an ACTION attribute which dictates the query performed against the LDAP. The valid values for this attribute are: add, delete, query (default), modify, and modifyDN. <cfldap> calls are turned into JNDI (Java Naming And Directory Interface) lookups. However, because <cfldap> wraps the calls, it will throw syntax errors if native JNDI code is passed to its attributes making LDAP injection more difficult.

XML Injection

Two parsers exist for XML data – SAX and DOM. ColdFusion uses DOM which reads the entire XML document into the server's memory. This requires the administrator to restrict the size of the JVM containing ColdFusion. ColdFusion is built on Java therefore by default, entity references are expanded during parsing. To prevent unbounded entity expansion, before a string is converted to an XML DOM, filter out DOCTYPE elements.

After the DOM has been read, to reduce the risk of XML Injection use the ColdFusion XML decision functions: isXML(), isXmlAttribute(), isXmlElement(), isXmlNode(), and isXmlRoot(). The isXML() function determines if a string is well-formed XML. The other functions determine whether or not the passed parameter is a valid part of an XML document. Use the xmlValidate() function to validate external XML documents against a Document Type Definition (DTD) or XML Schema.

Event Gateway, IM, and SMS Injection

ColdFusion MX 7 enables Event Gateways, instant messaging (IM), and SMS (short message service) for interacting with external systems. Event Gateways are ColdFusion components that respond asynchronously to non-HTTP requests – e.g. instant messages, SMS text from wireless devices, etc. ColdFusion provides Lotus Sametime and XMPP (Extensible Messaging and Presence Protocol) gateways for instant messaging. It also provides an event gateway for interacting with SMS text messages.

Injection along these gateways can happen when end users (and/or systems) send malicious code to execute on the server. These gateways all utilize ColdFusion Components (CFCs) for processing. Use standard ColdFusion functions, tags, and validation techniques to protect against malicious code injection. Sanitize all input strings and do not allow un-validated code to access backend systems.

Best Practices

- Use the XML functions to validate XML input.
- Before performing XPath searches and transformations in ColdFusion, validate the source before executing.
- Use ColdFusion validation techniques to sanitize strings passed to xmlSearch for performing XPath queries.
- When performing XML transformations only use a trusted source for the XSL stylesheet.

- Remove DOCTYPE elements from the XML string before converting it to an XML object.
- Use <cfparam> or <cfargument> to instantiate variables in ColdFusion. Use this tag with the name and type attributes. If the value is not of the specified type, ColdFusion returns an error.
- To handle untyped variables use IsValid() to validate its value against any legal object type that ColdFusion supports.
- Use <cfqueryparam> and <cfpropparam> to valid dynamic SQL variables against database datatypes.
- Use CFLDAP for accessing LDAP servers. Avoid allowing native JNDI calls to connect to LDAP.

Best Practice in Action

The sample code below shows a database authentication function using some of the input validation techniques discussed in this section.

```
<cffunction name="dblogin" access="private"
output="false" returntype="struct">

<cfargument name="strUserName" required="true"
type="string">

<cfargument name="strPassword" required="true"
type="string">

<cfset var retargs = StructNew()>

<cfif IsValid("regex", uUserName, "[A-Za-z0-9]*")
AND IsValid("regex", uPassword, "[A-Za-z0-9]*")>

<cfquery name="loginQuery"
dataSource="#Application.DB#" >

SELECT hashed_password, salt

FROM UserTable

WHERE UserName =

<cfqueryparam value="#strUserName#"
cfsqltype="CF_SQL_VARCHAR" maxlength="25">

</cfquery>

<cfif loginQuery.hashed_password EQ Hash(strPassword
& loginQuery.salt, "SHA-256" )>

<cfset retargs.authenticated="YES">

<cfset Session.UserName = strUserName>

<!-- Add code to get roles from database -->
```

```
<cfelse>
<cfset retargs.authenticated="NO">
</cfif>
<cfelse>
<cfset retargs.authenticated="NO">
</cfif>
<cfreturn retargs>
</cffunction>
```

Additional Resources

CFC Best Practices: http://www.adobe.com/devnet/coldfusion/articles/cfc_practices.html

ColdFusion Best Practices for Oracle Databases:
http://www.adobe.com/devnet/server_archive/articles/cf_best_practices_oracle.html

Improving ColdFusion Application Performance:
<http://ria.dzone.com/articles/improving-coldfusion-performance>

Performance Tuning for ColdFusion Applications
http://www.adobe.com/devnet/coldfusion/articles/coldfusion_performance_04.html

ColdFusion Standards & Best Practices:
<http://ortus.svnrepository.com/coldbox/trac.cgi/wiki/cbDevelopmentBestPractices>