

[<< Back to Dashboard](#)

ColdFusion Standards & Best Practices

Introduction

This document is intended to be a concise summary of best practices for anyone building ColdFusion applications within our team. Several external resources used when creating this document. Please note that this is a guideline based on past development experience and industry standards. Please use common sense when applying them and note that this document is ever changing as development trends continue to change.

- <http://www.wikipedia.com>
- <http://www.developer.com>
- <http://www.Adobe.com>
- <http://cfdj.sys-con.com/read/41660.htm>
- Sean Corfield's CFC Best Practices
 - <http://www.corfield.org>
- ColdFusion Locking Best Practices
-

http://www.adobe.com/devnet/server_archive/articles/cf_locking_best_practices.html

Naming & Conventions

Use good names for components, methods, arguments and local variables. This can sometimes be a disaster if developers choose random names or non qualified names for 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. Here are some *examples* NOT RULES:

- **a**cc for accessibility, as in ButtonAccImpl
- **a**uto for automatic, as in autoLayout
- **e**val for evaluate, as in EvalBindingResponder
- **i**mpl for implementation, as in ButtonAccImpl
- **i**nfo for information, as in GridRowInfo
- **n**um for number of, as in numChildren
- **m**in for minimum, as in minWidth
- **m**ax for maximum, as in maxHeight
- **n**av for navigation, as in NavBar
- **r**egex for regular expression, as in RegexValidator
- **u**til for utility, as in StringUtil

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 hyphens if necessary. You can package your objects/files using two well known approaches:

Contents

- [ColdFusion Standards & Best Practices](#)
 - [Introduction](#)
 - [Naming & Conventions](#)
 - [Abbreviations](#)
 - [Acronyms](#)
 - [Package Names](#)
 - [Class/Component/Interface Names](#)
 - [Methods](#)
 - [Type Names](#)
 - [CFML Tags, Custom Tags and Attributes](#)
 - [Arguments and Variables](#)
 - [Constants or Static Variables](#)
 - [CFC Best Practices & Conventions](#)
 - [Component Scopes](#)
 - [Instance Scope](#)
 - [Encapsulation](#)
 - [Constructors](#)
 - [Var Scoping](#)
 - [Output From CFCs](#)
 - [Hints & Documentation](#)
 - [Return Types & Duck Typing](#)
 - [Referencing External Scopes From CFC's](#)
 - [Getters/Accessors](#)
 - [Setters/Mutators](#)
 - [Default Arguments](#)
 - [Inheritance & Composition](#)
 - [CFC/Template Document Header](#)
 - [Variable Scoping](#)
 - [Application.cfc Best Practices](#)
 - [Name Property](#)
 - [General](#)
 - [Session Scope Usage](#)
 - [Use cflock On Shared Resources](#)
 - [Race Conditions](#)
 - [Do Not Abuse Pound Signs](#)
 - [Use Consistent Code Formatting](#)
 - [General ColdFusion Best Practices & Conventions](#)
 - [Related Guides](#)

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';
```

CFC Best Practices & Conventions

This section indicates some best practices when creating ColdFusion components. It also introduces several areas for development conventions and guidelines.

Component Scopes

Components can have instance data that can be placed in two different visibility scopes: private and public. Private variables are declared in the *variables* scope and public variables in the *this* scope. This means that the variables in the *this* scope will be available for modification from the outside world, while the *variables* scope is not accessible from the outside world directly. The only way to manipulate these private variables would be through methods that your object will expose to the outside world. This is called data-hiding or encapsulation, which you can read in the following point.

```
<cfset address = CreateObject(component,"address").init(>
<cfset instance = structnew(>

<cfset instance.firstname = "Luis">
<cfset instance.lastname = "Majano">
```

Note: The *variables* scope is the default scope in ColdFusion and therefore it is implied, so do not write it out.

Be very careful of when to make internal properties public as you will be violating encapsulation (look at next point). One of the best reasons for making variables public is if they do not change and can act like static constants. If your variable does not meet this criteria, then DO NOT expose it as public.

```
<cfset this.OPTIONS = "add,remove">
<cfset this.NOT_FOUND = 'NOTFOUND_>
<cfset this.EVENT_CACHEKEY_PREFIX = "cboxevent_event->
```

Note: Even though you are treating these variables like final static variables they CAN still be modified as ColdFusion does not support *final* or *static* variables. It is more of a convention and agreement to use this approach.

Instance Scope

For instance data, create a "virtual scope" inside the *variables* scope. For example, in the first line of your *init()* method you might have:

```
<cfset instance = structNew(>
```

You can then reference your instance data as "instance.foo". The benefit is that it separates your stateful instance data from the *variables* scope, which also contains references to all methods (public AND private) as well as a reference to "THIS" -- this becomes really handy if you need to do things like return a [memento](#) of your instance or do global operations on all instance data. It's also very handy if and when you need to clone a CFC, since you can move state data in one chunk rather than worrying about which keys in *variables* are your instance data and which are built-in keys. You would likely still use the *variables* scope for non-stateful instance data such as references to other CFC's your instance uses. ColdFusion 9 has the potential to change this best practice as they allow for the creation of instance data via the *cfproperty* tag and placing them in the *variables* scope. They also create implicit getters/setters for these properties, which saves time. Therefore, this best practice could potentially be removed in later editions.

Below is a way to implement the state pattern on an object to get and set an entire object's instance data.

```
<--- Getter/Setter memento --->
<cffunction name="getMemento" access="public" returnType="struct" output="false" hint="Get the memento">
  <cfreturn variables.instance>
</cffunction>
<cffunction name="setMemento" access="public" returnType="void" output="false" hint="Set the memento">
  <cfargument name="memento" type="struct" required="true">
  <cfset variables.instance = arguments.memento>
</cffunction>
```

Encapsulation

Encapsulation provides the basis for modularity by hiding information from unwanted outside access and attaching that information to only methods that need access to it. This binds data and operations tightly together and separates them from external access that may corrupt/change them intentionally or unintentionally. Encapsulation is achieved by declaring variables as private in a CFC (*variables* scope). This gives access to data to only public/package member functions of the CFC. You can then create their mutators (setters) and accessors (getters) via public methods. Some benefits of encapsulation are:

- You can keep the exposed API the same while changing how your CFC works internally without breaking any code that uses your CFC
- Prevents the developer from getting in trouble by violating your internal assumptions about how the instance data works
- It hides your implementation from the outside world

```
-- THESE ARE GOOD --
<cfset stuff = myCFC.getStuff(>
<cfset myCFC.setStuff(stuff)>

-- THESE ARE BAD --
<cfset stuff = myCFC.stuff
```

```
<cfset myCFC.stuff = stuff
```

Constructors

Always have an *init()* method that acts as your constructor and returns *this* (unless you are building a web services/flash remoting facade). Even if the method has a simple return statement, it is always best practice that every object have a constructor method.

```
<cfcomponent name="Converter" output="false">
  <cffunction name="init" access="public" output="false" returnType="Converter" hint="Constructor">
    <cfreturn this>
  </cffunction>
</cfcomponent>
```

If you are using inheritance then you must call the parent constructor by accessing the *super* scope.

```
<cfcomponent name="Converter" output="false" extends="BaseConverter">
  <cffunction name="init" access="public" output="false" returnType="Converter" hint="Constructor">
    <cfset super.init()>
    <cfreturn this>
  </cffunction>
</cfcomponent>
```

Var Scoping

Always, always, always use "var" for local variables inside your methods, including ALL loop counters, temporary variables, queries, etc. This is called "*var scoping*". If you do not do this, your component will not be thread-safe. This means that if somebody persists (stores) this component in memory, succinct calls can and will override variables and create all sorts of memory problems. There is an open source project called [varscoper](#) that can check all of your components for var scoping issues, even if they are using cfscrip. **ALWAYS VAR SCOPE.**

Var scoping applies to methods inside components and also to UDF's in order to comply with best practices.

```
-- DO THIS --
<cffunction name="myFunction" access="public" returnType="void" output="false" hint="This methods does nothing">
  <cfset var i = 0>
  <cfset var qGet = "">
  <cfquery name="qGet">
  </cfquery>
  <cfloop from="1" to ="20" index="i">
  </cfloop>
</cffunction>

-- NOT THIS --
<cffunction name="myFunction" access="public" returnType="void" output="false" hint="This methods does nothing">
  <cfquery name="qGet">
  </cfquery>
  <cfloop from="1" to ="20" index="i">
  </cfloop>
</cffunction>
```

Output From CFCs

Always (**with rare exceptions**) use `output="false"` in your `cffunction` and `cfcomponent` tags. Do not output directly to the buffer inside a CFC method; instead return a string from the method. The main reason is that you don't want to break encapsulation. By outputting directly to the output stream you assume knowledge of the external environment of the CFC. However, if you return a string then you get the exact same behavior when you do `#myCFC.someHTMLGeneratingMethod()` but you gain the advantage of not assuming that's how your method will be used. For instance, what if the method that returns the string is used inside of a big `cfscrip` block where someone is building a string via concatenation? Everything will break.

```
<cfcomponent output="false">
  <cffunction name="getContent" output="false" access="public" returnType="string">
    <cfreturn "This is my content">
  </cffunction>
</cfcomponent>
```

Hints & Documentation

Document your component, methods and arguments by using the `hint` attribute in those tags. This will help fellow developers and even you, when determining what a method, argument or component does, can do, etc. You can also use several tools to create cfc documentation according to your component metadata. This should be done for the following tags:

- `cfcomponent`
- `cffunction`
- `cfproperty`
- `cfargument`

Return Types & Duck Typing

Use the `returnType` attribute of the `cffunction` tag and the `type` attribute of the `cfargument` tag to add documentation and for runtime type checking. Also remember that `void` is the return type when your method call does not return anything.

Duck Typing is when you use the return type or type of *any* in a *cffunction* or *cfargument* tag. This is a useful technique when dealing with a dynamic language such as ColdFusion. This means that the argument or object returned can be ANYTHING, which then your caller needs to determine what to do with it and what it is based on pre-determined conventions. A side effect of not using a strong type is a speed enhancement, since ColdFusion does not check the validity of the types. This side effect should not be used to get more performance, unless absolutely necessary.

This dynamic nature of arguments and return types brings forth great power in a dynamic language, but it also opens holes for runtime exceptions. However, thanks to unit testing, these runtime exceptions should be minimized. So as a followup guideline to duck typing is that you must have unit tests for these components.

Referencing External Scopes From CFC's

Do not directly reference external scopes, i.e.: session/application/client/server/request variables, inside a CFC. If you reference external scopes you will be breaking the encapsulation and cohesiveness of the component at hand. You have now binded the component to an external scope that must exist in order for this component to work. This also provides difficulties when unit testing.

However, the one exception to referencing external scopes is when building *facades*, especially for web services/flash remoting, in which case ALL references to shared scope variables should be encapsulated within the facade. This means, for instance, passing in the dsn instead of referring to application.dsn when doing database queries. If you do not know what a facade is, then please search for *facade pattern* to learn more about it. It basically encapsulates a shared scope such as application,session, etc into a CFC.

Note: This is not a golden rule, but try to adhere to it

Getters/Accessors

Getters or accessors are simple methods in a CFC that can access instance data. These are very simple methods that basically just retrieve data and follow a convention:*get{property name}()*

```
// Property name is firstname
<cffunction name="getFirstName" output="false" access="public" returnType="string">
  <cfreturn instance.firstName
</cffunction>
```

Setters/Mutators

Setters or mutators are simple methods in a CFC that can modify instance data. These are very simple methods that basically just set data and follow a convention:*set{property name}(value)*. They have one argument with the same name as the property or a generic name, i.e. *value*. The return type for such methods is *void*

```
// Property name is firstname
<cffunction name="setFirstName" output="false" access="public" returnType="void">
  <cfargument name="value" type="string">
  <cfset instance.firstName = arguments.value
</cffunction>
```

Default Arguments

In general, non-required arguments of a CFC method should have a default value specified, unless you will be programmatically checking for existence using *structKeyExists(arguments, "key")*.

```
<cfargument name="isReadOnly" type="boolean" default="false" required="false">
<cfargument name="maxRows" type="numeric" default="10" required="false">
```

Inheritance & Composition

Use inheritance only when describing an "is-a" relationship, not for a "has-a" relationship (composition) or for code reuse only. For a nice summary, visit <http://cnx.rice.edu/content/ml1709/latest/>

Do not use a component as a huge glorified set of methods and call that code reuse. Components are synonymous to objects, they should have an identity upon themselves. Put in practice your [Ontology](#) skills and define what the CFC will do for you and what is their identity.

Always prefer object composition over inheritance. This is where another component is created or injected as a property of the object at hand. There are several reasons of why to choose composition over inheritance in order to make your designs more flexible and not coupled at compile time, which inheritance does. Composition brings in functionality at runtime as you can switch implementations, etc. Some resources are:

- <http://brighton.ncsa.uiuc.edu/prajlich/T/node14.html>
- <http://www.artima.com/lejava/articles/designprinciples4.html>
- <http://guidewiredevelopment.wordpress.com/2008/02/05/favoring-composition-over-inheritance/>

CFC/Template Document Header

It would be of best practice to add a document header that can follow the following standard or something similar to any CFC or template:

```
<----->
Author      : Luis Majano
Date       : 3/13/2009
Description :
  This is my component that does fantastic stuff!
<----->
```

Variable Scoping

ColdFusion variables must be scope 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>
```

Application.cfc Best Practices

Name Property

ColdFusion treats separate applications by looking at the *application.cfc* name property. This has to be unique, especially when dealing with multiple applications in a server. If collisions occur, then scopes will be shared and nasty things could happen. It is imperative to distinguish applications uniquely. This will even be more of a thing to watch out for in future versions of CF. Therefore, try the following technique:

```
this.name = "MyApp_" & hash(getCurrentTemplatePath());
```

This will provide you with a unique hash for your application according to where it is located on the server. This way, no collisions will apply.

General

- Avoid using *Application.cfm* and move to *Application.cfc*. Not only will you get more functionality, but it is a standard. Also, **always** have an *Application.cfc* for your application. If not, CF will search for one and use it's settings and you do not want that.
- Only create listener methods that you will use, do not create them as empty functions unnecessarily.

Session Scope Usage

Be very careful when using *session* scoped variables in your applications, especially if used under heavy load or high traffic. Remember that *session* variables are per user and consume memory. If they are misused or you issue *session* variables randomly, your RAM usage will increase exponentially. Consider always having low timeouts for your *session* scopes and always disallowing bots to have *session* variables.

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.

```
-- DO THIS --
<cflock name="FileOperation" timeout="20" throwOnTimeout=true>
  <cffile action="write" file="#filePath#" output="#content#">
</cflock>

<--- application scope is exclusively locked on the cache --->
<cflock type="readonly" scope="application" timeout="10" throwOnTimeout=true>
  <cfset myVar = application.cache.getValue(#k#)>
</cflock>

<cfquery name="variables.qUser" datasource=#request.dsn#>
  SELECT FirstName, LastName
  FROM Users
  WHERE UserID = #request.UserID#
</cfquery>
<cflock scope="application" timeout="2" type="exclusive">
  <cfset application.qUser=variables.qUser
</cflock>

-- NOT THIS --
<cflock name="FileOperation">
  <cffile action="write" file="#filePath#" output="#content#">
</cflock>

<cfset myVar = application.cache.getValue(#k#)>

<cflock scope="application" timeout="2" type="exclusive">
<cfquery name="application.qUser" datasource=#request.dsn#>
  SELECT FirstName, LastName
  FROM Users
  WHERE UserID = #request.UserID#
</cfquery>
</cflock>
```

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").init(>
  </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").init(>
  </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.

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>

```

General ColdFusion Best Practices & Conventions

- Components are supposed to be objects and have an identity. Always ask yourself what this component's responsibilities are and how will it interact with its surroundings.
- Variables pass in and out of components by reference or by value based on the same rules as the rest of CFML. For instance, strings, arrays, numbers, and dates all pass by value, but structures, queries, and all other "complex" objects (including CFC instances) pass by reference.
- Arrays in ColdFusion pass by value in Adobe ColdFusion, so beware of this behavior as it is not the same as in Java or other CFML engines.
- *Duplicate()* and *CFWDDX* do not work on CFC instances (ColdFusion 7 and below). CFC's can only be serialized in ColdFusion 8 and with several restrictions. Be careful when serializing objects as the entire object graphs have the potential of being serialized. ColdFusion 9 presents mechanisms to restrict component serializations which helps incredibly.
- When extending a component outside the base component's package, the sub-component does not inherit *package* permissions -- thus, you cannot call *package* methods on other CFCs in the package of the base component from the sub-component.
- You can have a method that has a *returnType* or an argument that has a *type* of a base component and return any component that extends that base component. For example, if *methodA* takes an argument *foo* of type *motorVehicle* and you pass *foo* as an instance of *car*, which extends *motorVehicle* then *methodA* will honor that *car* is a *motorVehicle* when doing type checking on the argument *foo*.
- The previous concept applies to interfaces and inheritance.
- Use interfaces when you want to provide clear API definitions that need to be implemented. They can be good documentation tools and provide compile time checks on your code.
- Use *structKeyExists* instead of *isDefined* when checking for existence.

```

-- DO THIS --
<cfif structKeyExists(arguments,"car")>
</cfif>

-- NOT THIS --
<cfif isDefined("arguments.car")>
</cfif>

```

- Use *cfswitch* instead of *cfif* if you have a specific expression that you can evaluate against and if you will have more than 2 *cfelseif* clauses. Not only does it provide more readability, but your code will make more sense.
- Avoid usage of *if* if at all possible as it is documented to be slower. However, sometimes it can prove handy.
- Avoid usage of *evaluate()* expressions. They have to be evaluated by the ColdFusion engine and will always run slower. There are times when you will have to use them, especially when doing dynamic concatenations, but try to avoid them at all possible.

```

-- DO THIS --
<cfset value = form["field#i#"]>

```

```
-- NOT THIS --
<cfset value = evaluate(form.field#i#)>
```

- Use boolean evaluations

```
-- DO THIS --
<cfif len(firstName)></cfif>
<cfif NOT obj.isEmpty()></cfif>
<cfif query.recordcount></cfif>
<cfif arrayLen(myArray)></cfif>
```

```
-- NOT THIS --
<cfif firstName eq ""></cfif>
<cfif obj.isEmpty() eq false></cfif>
<cfif query.recordcountgt 0></cfif>
<cfif arrayLen(myArray)gt 0></cfif>
```

- When you are creating view templates, try to always surround it with 1 *cfoutput* tag, instead of nesting them all over the place.

```
-- DO THIS --
<cfoutput>
<html>
  <head>
    #head#
  </head>
  <body>
    #leftBar#
    #content#
    #footer#
  </body>
</html>
</cfoutput>
```

```
-- NOT THIS --
<html>
  <head>
    <cfoutput#head#</cfoutput>
  </head>
  <body>
    <cfoutput#leftBar#</cfoutput>
    <cfoutput#body#</cfoutput>
    <cfoutput#footer#</cfoutput>
  </body>
</html>
```

- Code for portability. Avoid at all costs on hardcoding paths, urls, file locations, etc. If you are using a framework, which you should, they usually provide a way to setup application global variables. If not within a framework context, try to set global variables in a shared scope such as *application* scope once when your application loads and then just grab settings from it. Always believe that your application locations can change.

Related Guides

- [Model-View-Controller Demystified](#)
- [ColdFusion SQL Injection Protection Best Practices](#)
- [Database Naming Conventions](#)