

[← Back to Dashboard](#)

What's New With ColdBox 3.0.0

Introduction

ColdBox 3.0.0 is an entirely new release and you must take care when updating from older releases as there might be some compatibility issues that can be found here: [Compatibility Guide for the 3.0.0](#). Below is the new configuration file XSD documentation:

- [Configuration File Schemas Documentation](#)

Training, Presentations, More

- [ColdBox Connection: Introducing ColdBox 3](#)

Bug Fixes

You can see our defect report for ColdBox 3.0.0 at our [Assembly workspace](#)

Application Template Updates

- New application templates
- All application generators updated also.

Major New Features

This section lists the major updates to the ColdBox 3.0.0 Platform

Programmatic Configuration

Look Ma! No more XML! That's right, you can leave your XML worries behind and configure the entire application using a simple configuration CFC. Not only that, but we added environment detection right into the new programmatic CFC. To learn more about it, please visit our [configuration CFC page](#).

CacheBox: The Enterprise ColdFusion Cache Engine, Aggregator and API

We are opening our enterprise level cache as a standalone application but we are also enhancing it in order to create CacheBox. CacheBox is not only an enterprise level caching engine but also functions as a cache aggregator and API. You can aggregate different caching engines or types of the same engine into one single umbrella. It also gives you built in logging, events, synchronization, shutdown/startup procedures and best of all a cache agnostic API. You can then build your applications based on an abstract API and then be able to configure the caches for your applications at runtime. This gives you greater flexibility and scalability when planning and writing your applications. Our caching engine has also been revamped with new events and we now have flexible and customizable object storages. From concurrent object stores to simple JDBC replicated storages. We also opened our storage and cache provider API so you can even build your own and extend the framework. Read our [CacheBox Documentation](#).

WireBox: The Enterprise Dependency Injection Framework

We are also opening up our internal DI/AOP framework for usage not only within ColdBox applications but for ANY ColdFusion application. WireBox is an enterprise dependency injection framework based on our amazing conventions and a binding DSL (Domain Specific Language) for object mapping. It is built on simplicity and highly leverages ColdFusion metadata to allow for a better workflow and introduction into dependency injection. We have also take it a step further and WireBox allows for the creation of several different target objects:

- ColdFusion Components
- Java Components
- RSS feeds
- Webservices
- Constant Values
- Object Factory Methods
- Annotation Factories

We have also created a simple mapping binding language (DSL) that will help you define object relationships and construction. No XML, purely programmatic. Read our [WireBox Documentation](#)

MockBox: The ColdBox Mocking/Stubbing Framework

In one of our most ambitious releases, we are also introducing one of the newest standalone services/frameworks that are now bundled with ColdBox. MockBox is a next generation mocking/stubbing framework that is a necessity when unit testing ColdFusion components. Read our [MockBox Documentation](#)

LogBox: The Enterprise ColdFusion Logging Library

We are opening up our internal logging framework for usage not only within ColdBox applications but for ANY ColdFusion application. It becomes another addition to our standalone services/frameworks that are bundled with the ColdBox Platform. LogBox is an enterprise ColdFusion logging library designed to give you flexibility, simplicity and power when logging or tracing is needed in your applications. LogBox allows you to easily build upon its logging framework in order to meet any logging or reporting needs your applications has. LogBox allows you to create multiple destinations for your loggings and even be able to configure them or change them at runtime. LogBox was inspired by the original logging capabilities in ColdBox and in the Log4j project. Read our [LogBox Documentation](#).

- All logging for internal ColdBox classes can now be controlled via your application's LogBox configuration.

ColdBox Modules

You can now build your ColdBox applications in a more modular and segregated approach. ColdBox Modules are self-contained subsets of a ColdBox application that can be dropped in to any ColdBox application and become alive as part of the host application. They will bring re-usability and extensibility to any ColdBox application, as now you can break them down further into a collection of modules. This concept has been around in software design for a long time as it is always essential to partition a system into manageable modules or parts.

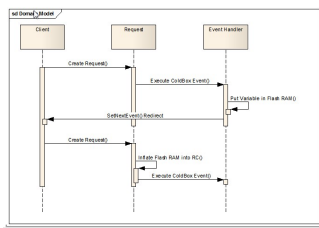
"In structured design and data-driven design, a module is a generic term used to describe a named and addressable group of program statements" by Craig Borysowich (Chief Technology Tactician)

ColdBox Modules will change the way you approach application development as you can now have a foundation architecture that can scale easily and provide you enough manageability to reduce maintenance and increase development. Such a design means that development, testing and maintenance becomes easier, faster and with a much lower cost. Welcome to a brave new world! Read our [Modules Documentation](#)

[ColdBox Modules Debugger Panel](#)

ColdBox Flash RAM

ColdBox has Flash RAM capabilities since version 2.0 (long long time ago). However, in this release we are taking a step further and enhancing the capabilities. Now every handler, plugin, interceptor, layout and view will have a **flash** object in their *variables* scope already configured for usage. The entire flash RAM capabilities are now encapsulated in a flash object that you can use in your entire ColdBox code. Not only that, we based our Flash object on an interface and now you can build your own Flash RAM implementations very easily. What this means is that you can build a Flash RAM implementation that can store the Flash variables wherever you want.



[ColdBox Flash RAM Sequence Diagram](#)

We also added the capabilities to maintain flows or conversations between requests by using our *discard()* and *keep()* methods in our Flash objects. This will continue to expand in our 3.X releases as we build our own conversations DSL to define programmatically wizard like or complex web conversations.

For those who do not know what Flash RAM does, please read our [Flash RAM Documentation](#). A brief introduction is that there are times where you need to store user related variables in some kind of permanent storage then relocate the user into another section of your application, be able to retrieve the data, use it and then clean it. All of these tedious operations are definitely doable by why reinvent the wheel if we can have the platform give us a tool for maintaining conversation variables across requests.

The included implementations for ColdBox 3.0.0 are:

- **Session** : Persists variables in session scope
- **Cluster** : Persists variables in cluster scope (Railo only)
- **Client** : Persists variables in client scope (Simple or complex, we serialize the data for you)
- **ColdBox Cache** : Persists variables in the ColdBox Cache
- **Mock** : Mocks the storage of Flash variables. Great for unit/integration testing.

You will find all of these implementations in the following directory: `coldbox.system.web.flash`. Please read our [Flash RAM Documentation](#).

```
// handler code
function saveForm(event){
// save post
flash.put('notice','Saved the form baby!')
// welcome to another event
setNextEvent({cer: 'show'})
}
function show(event){
// Nothing to do with flash, inflating by flash object automatically
event.setView({cer: 'show'})
}
// User/show.cfm template using if statements
<cfflash:exists?notice>
<div class=notice@flash.get('notice')&k/>
```

Contents

- [What's New With ColdBox 3.0.0](#)
- [Introduction](#)
- [Training, Presentations, More](#)
- [Bug Fixes](#)
- [Application Template Updates](#)
- [Major New Features](#)
- [Programmatic Configuration](#)
- [CacheBox - The Enterprise ColdFusion Cache Engine, Aggregator and API](#)
- [WireBox - The Enterprise Dependency Injection Framework](#)
- [MockBox - The ColdBox Mocking/Stubbing Framework](#)
- [LogBox - The Enterprise ColdFusion Logging Library](#)
- [ColdBox Modules](#)
- [ColdBox Flash RAM](#)
- [ColdBox Extensions](#)
- [Layouts Externalization](#)
- [ColdFusion ORM Interactions](#)
- [ColdBox Form Enhancements](#)
- [ColdBox Factory Enhancements](#)
- [ColdBox Proxy Enhancements](#)
- [Layout Enhancements](#)
- [No More Inheritance](#)
- [Persistence Metadata](#)
- [EJB Pass-Through Constructor](#)
- [Persistence Updates](#)
- [Remote Module Plugins](#)
- [New Object Properties](#)
- [New Methods](#)
- [JIT Plugin Enhancements](#)
- [New Class-Scoped Plugins](#)
- [New Validator Plugins](#)
- [New ORM Service Plugins](#)
- [Utilities Plugin Refactor](#)
- [New HTML Helper Plugins](#)
- [New MailService Plugin](#)
- [MessageBox Updates](#)
- [AntiSamy XSS Plugin Updated](#)
- [Security Plugin](#)
- [Implicit View Rendering](#)
- [View Editor Helper](#)
- [Layout Helper](#)
- [Local Generator](#)
- [Local Builder](#)
- [New Resource Bundle](#)
- [Layout and Enhancements](#)
- [XML Converter plugin](#)
- [Interceptor Enhancements](#)
- [No More Inheritance](#)
- [New Interception Points](#)
- [Extending The Debugger](#)
- [New Object Properties](#)
- [Access Pattern Annotation](#)
- [New Interceptor Handler Loader](#)
- [Autowire Interceptor Updates](#)
- [Access Interceptor Updates](#)
- [Logging Interceptor Updates](#)
- [JSP Interceptor Updates](#)
- [Event Handler Enhancements](#)
- [No More Inheritance](#)
- [Persistence Metadata](#)
- [New Object Properties](#)
- [New Event Interceptors Updated](#)
- [Extended method action methods](#)
- [New implicit handler methods aroundHandler\(\) aroundAction\(\)](#)
- [New implicit handler method onErrors\(\)](#)
- [Handler Action HTTP Security](#)
- [New Implicit Handler MissingTemplateHandler](#)
- [New Implicit Handler ApplicationErrorHandler](#)
- [Annotation Event Arguments](#)
- [Model Interceptor Enhancements \(WireBox\)](#)
- [New Interceptor Settings](#)
- [Dependency DSL Metadata](#)
- [Updated Dependency Injection DSL](#)
- [Business Context Enhancements](#)
- [New Methods](#)
- [Assemblies](#)
- [Event Subscriptions](#)
- [Event Handler Data Updated](#)
- [Parameters](#)
- [ColdBox Test Suite Enhancements](#)
- [ColdBox Meta Controller Enhancements](#)
- [Unit Integration Testing Enhancements](#)
- [New Methods](#)
- [New Targeted Plugin Testing](#)
- [New Targeted Handler Testing](#)
- [New Targeted Interceptor Testing](#)
- [New Targeted Model Testing](#)
- [Performance Updates](#)

```
</cfif>
// User/show.cfm using defaults
<div class="notice" #flash.get(name="notice,default")></div>
```

ColdBox Extensions

You can now very easily override the functionality of core plugins without touching them and you can also add more plugins to the core by using our new ColdBox Extensions. This is basically a folder in the core called `coldbox/system/extensions`. In this folder (as of now), you can drop in your plugins in the `plugins` directory. What this does is that the framework will look for core plugins in this location **FIRST** and then if not found look in the core plugins directory: `coldbox/system/plugins`. This is extremely useful if you want to override the behavior of internal plugins or just have a location for your own global core plugins.

You can also customize the location to ANYWHERE you like by adding a simple setting to your configuration: `ColdBoxExtensionsLocation`. This setting is a dot notation path or cf mapping to where your plugins and (*ahem*) future services (announced later), will be placed.

```
coldbox = {
  coldboxExtensionsLocation: shared.extensions*
};
// or
<setting name="ColdBoxExtensionsLocation" value="shared.extensions">
```

Layouts External Location

Long overdue external location setting that can act as a secondary lookup location for layouts.

ColdFusion ORM Integrations

We have created several classes that can give you greater support for ColdFusion ORM integrations via Hibernate. Read the documentation here: [CF ORM Extras](#).

- **EventHandler**: We added a base ORM Event Handler object that bridges your ColdBox applications to the Hibernate event model. It also allows you to inject entities with other objects using the ColdBox Autowire DSL and create more domain driven objects.
- **BaseORMService**: A base service layer object that can help you manage and work with entities using ORM capabilities. It can offer dynamic methods, query executions, introspections, hibernate transactions and much more.
- **VirtualEntityService**: We also created a virtual entity service that can help you create virtual service layers concentrated on specific entities.
- **TransactionalAspect**: An interceptor that allows for AOP to be applied to model objects and handler methods. It can demarcate transactional methods by usage of annotations.

This is a great way to start off your ColdFusion ORM projects. We believe these base service layers can give you a good 85% of functionality right off the bat.

ColdBox Cache Enhancements

- Lots of concurrency and load testing enhancements
- New method `lookupMulti()` so you can do multiple key lookups
- New method `clearEventMulti()` so you can clear multiple event cachings at once

ColdBox Factory Enhancements

You can now use the following new functions in the ColdBox Factory

- `getRequestContext()`: Get the context object
- `getRequestCollection()`: Get a reference to the request collection.
- `getLogBox()`: Get a reference to logbox
- `getLogger(category)`: Get a logger object
- `getRootLogger()`: Get a root logger object

ColdBox Proxy Enhancements

You can now use the following new functions in the ColdBox Proxy

- `getRequestContext()`: Get the context object
- `getRequestCollection()`: Get a reference to the request collection.
- `getLogBox()`: Get a reference to logbox
- `getLogger(category)`: Get a logger object
- `getRootLogger()`: Get a root logger object
- `getCacheBox()`: Get a reference to CacheBox
- `getColdBoxOCM(cacheName)`: Get a named cache from CacheBox
- `getRemotingUtil()`: We have added a new utility object called `RemotingUtil.cfc` that contains some nice methods when working with remote calls.

New interception point: `preProxyResults`.

Plugin Enhancements

No More Inheritance

The major change to plugins is that now plugins DO NOT need to inherit from the Core ColdBox Plugin. You can now build simple CFCs and they will be decorated at runtime and adapted for usage within ColdBox. Ahh, doesn't that feel great, less coupling! Not only that, we are still backwards compatible and if you prefer to do inheritance, then go for it, we are not prejudice to your decision. You can still use ALL of the methods you are used to when building plugins, we take care of the hard stuff to make your code look nicer and easier. Also, if you declare a constructor in your plugin, ColdBox will call it for you. We also create a virtual `super` scope for your inheritance components so you can still provide overrides. The name of the scope is called: `$super` so you can talk to the base class.

```
**
* My awesome plugin
* @singleton
*/
component {
  function init() {
    setPluginNameToCol_Awesome*
    setPluginAuthorToMl_MaJan*
    return this
  }
}
```

Persistence Metadata

You can now use the `singleton` annotation on the `cfcomponent` tag to denote that this object will live for the entire application. This is the same as adding `cache=true cacheTimeout=0` to the component tag:

```
**
* @singletontrue
*/
component {
}

// OR
<cfcomponent singleton=true</cfcomponent>
```

By pass Plugin Constructor

You can also bypass an automatic `init()` call on the plugin by passing a new argument to the `getMyPlugin()` or `getPlugin()` calls.

```
// get plugin with no constructor called, I will do it
oPlugin = getMyPlugin(pluginName="somePlugin" init=false, init("Yeaaaaa!"))
```

Persistence Updates

You can now declare a `singleton` annotation on the plugin declaration and it will be treated as a singleton.

```
**
* My awesome plugin
* @singleton
*/
component {
  function init() {
    return this
  }
}
```

Retrieve Module Plugins

You can now pass in a `module` argument to all plugin functions to retrieve a plugin directly from a module.

```
// Get the Smiley plugin from the fun module
smiley = getMyPlugin(module="smiley", module="fun");
```

New Object Properties

All plugins will have certain objects in the `variables` scope available to them for usage. The following is a table listing of all the objects you can use:

Property	New	Description
<code>logBox</code>	Yes	The reference to the LogBox library
<code>log</code>	Yes	A pre-configured <code>LogBox</code> Logger object for this specific class object
<code>flash</code>	Yes	A reference to the current configured Flash Object Implementation that inherits from the AbstractFlashScope (<code>coldbox.system.web.flash.AbstractFlashScope</code>)
<code>controller</code>	No	The main application ColdBox controller (<code>coldbox.system.web.Controller</code>)

New Methods

You now have a few extra methods when dealing with plugin development:

- `getRequestContext()`: Get the context object
- `getRequestCollection()`: Get a reference to the collection
- `getPluginAuthor()`: Get the plugin author
- `getPluginAuthorURL()`: Get the plugin author's url

IOC Plugin Enhancements

- The IOC plugin has been completely revamped to now leverage factory adapters. So now the IOC plugin can talk to any object factory that implements our adapter interface. We have updated it to work with ColdSpring, ColdSpring2, LightWire and WireBox or your very own object factory.

- You can also now declare parent factories for the main factory of your application if your adapter supports it. The IOC plugin will create the parent factory for you and wire it to the main one for you. Cool huh?

```
// CFC declaration
//IOC Integration
ioc = {
  framework "coldspring"
```

```

reload      true
objectCaching true
definitionFile "conf/coldspring.xml.zfm"
parentFactory = {
    framework "coldspring"
    definitionFile "conf/parent.xml.cfm"
}
};

// Custom factory
ioc = {
    // The framework is the type of the adapter to use.
    framework "com.model.MyBeanFactory"
    reload      true
    objectCaching true
    definitionFile "conf/coldspring.xml.cfm"
};

```

- The available adapters are:
 - **coldspring** : Builds a ColdSpring factory
 - **coldspring2** : Builds a ColdSpring 2 factory
 - **lightwire** : Builds a LightWire factory
 - **wirebox** : Builds a WireBox factory

If you would like to build your own adapter it must adhere to the following Base Adapter:

```
coldbox.system.ioc.AbstractIOAdapter
```

We also recommend looking at the current ones for inspiration.

New Cluster Scope Plugin

New plugin for using the cluster scope on rails enabled CFML engines.

New Validator Plugin

A plugin that helps you do any kind of data validation, check its API out: <http://www.coldbox.org/api> (coldbox.system.plugins.Validator)

New ORMService Plugin

We added a new plugin that will act as a generic ORM service layer for your applications. This generic service layer will give you a nice way for you to build application using a simple service layer already pre-coded. This service layer is based on our [BaseORMService](#) class.

Utilities Plugin Refactored

The *Utilities* plugin has been refactored into several new plugins:

- **DateUtils** : Helps all your date time necessities
- **FileUtils** : Helps all your file necessities
- **JVMutils** : Be cool and talk to the JVM
- **Utilities** : Generic utility methods

New HTMLHelper Plugin

This cool new little plugin will help you out when dealing with HTML. It can help you keep track of js or css assets and make sure they get loaded only once, render strict html css or javascript, render lists from arrays or queries, render tables from arrays or queries, create cool dynamic meta tags, create doctype headers and so much more. Please read [HTMLHelper Documentation](#) for more information and awesomeness!

New MailService Plugin

You can now be cool and send emails via our very own MailService plugin. Use it in script and in any CFML engine. Not only that but this service can actually do token replacements for you at the time of sending emails. So what is this? Well, once you tell the mail service to give you a new mail object, you can set attachments on it, headers, etc but you can also pass in a structure of tokens. Once the email is about to be sent, the MailService will do token replacements on the body of the email against these tokens.

- Each token is delimited by the following @token@

```

// Get new mail object
mail = getPluginMailService("newMail").config({from:"@coldbox.org",
    to:"some@coldbox.org",
    type:"html",
    subject:"Hi For You Captain Awesome"});
// create some tokens
tokens = {name:"Mia",time:dateformat(now,{full})};
mail.setBodyTokens(tokens);

// Set some body text
mail.setBody("<h1>Hello @name@, how are you today?</h1> <p>Today is the <b>time</b>.</p> <br/>@www.coldbox.org GoldBox Rules!</p>");

// Send the mail
results = getPluginMailService().send(mail);

```

MessageBox Updates

Some new convenience methods:

- **info()** : To render info message directly
- **warning()** : To render a warning
- **error()** : To render an error

If you want to style your own messagebox you will need to create a setting called: **messagebox_style_override** and set it to true. Then make sure the css for the messagebox exists.

```

settings = {
    messagebox_style_override true
};

```

You can now also render messages a-la-carte or on-demand by using the *renderMessage* method:

```

#getPluginMessageBox().renderMessage@info:"Info Message"
#getPluginMessageBox().renderMessage@warning:"Flash.getNotice"

```

AntiSamy XSS Plugin Updated

The AntiSamy XSS plugin has been updated from the latest AntiSamy release project: <http://www.oswsp.org/index.php/Category:OWASP-AntiSamy-Project>

Yes, ColdBox has XSS Cleanup built in since version 2.0

- You can now create custom policies for this plugin in three steps:

1. Create a policy file based on the ones shipped and store it wherever you like in your application.
2. Create a custom setting in your application with the path to this custom file: **AntiSamy_Custom_Plugin**

```

// custom settings
settings = {
    AntiSamy_Custom_Plugin = "expandPath(mapping['/includes/MyAntiSamy.xml'])
};

```

3. You can now call the AntiSamy's *HTMLSanitizer()* method with **custom** as the policy to use.

```
clean = getPluginAntiSamy().HTMLSanitizer(rc.text@atom?);
```

Renderer Plugin

- All rendering methods updated to support module renderings.
- **renderView()** now has some new optional arguments:
 - **cacheSuffix**: A suffix to add to the caching keys for the view
 - **module**: Will render the view in the specified module
 - **renderLayout()** can now be used a-la-carte and with some cool optional arguments
 - **layout**: The layout to render
 - **view**: The view argument to pass into the layout
 - **module**: Will render the layout in the specified module

Implicit View Rendering

You can now render views without the need of creating events for them. If you are building a prototype or migrating from legacy code into the framework, this will be a great asset. So let's say you want to execute the event: **site.contact** this should map to a *site.cfc* and *contact()* method. However, if they do not exist, the framework will look for a view called: *views/site/contact.cfm* and if it finds it, it will execute it!

View Folder Helper

We also introduce view folder helpers. We already have view helpers by conventions, basically if you have a view called *home.cfm* and you create alongside it a file called *homeHelper.cfm* that helper's UDFs will be injected at runtime into the *home.cfm* and you can use those UDFs. For 3.0.0 we expanded on this (thanks to an awesome student at one of our trainings) and we introduce a **view folder helper**. What this means is that you can create a convention based on the folder name and **all** the views within that folder will get to use the helper UDFs. So if I have the following layout:

```

/users
  index.cfm
  editor.cfm
  permissions.cfm
  listings.cfm
  usersHelper.cfm *

```

You can see that all the views are in the *users* folder, and I created a *usersHelper.cfm*. Basically: *[name of folder]Helper.cfm*. The Renderer now sees a folder helper and will inject it for all the renderings within this folder ONLY.

Query Helper

- Addition of *getCSV()* to convert a query to a CSV string
- Sorting can now also be case-insensitive

Feed Generator

- Additional support formats like iTunes/Atom and so much more
- Fully documented with new sample applications

Feed Reader

- Added multiple new XML supports
- Added more types of generated content from reading feeds
- Fully documented with new sample applications

i18n & Resource Bundle

Both the *i18n* and *ResourceBundle* plugins have been completely reworked and reconstructed for this release. They are now faster, leaner and meaner! Their performance is substantially faster and more stable under race conditions as lots of thread conditions have been eliminated. Below are some of the major enhancements apart from performance and reworking.

- If the translation requested is not found in the localized resource bundle, the mechanisms will look for it in the default language bundles
- `getResource(resource, [default], [locale])` is now the new signature for this method, which enables you to pass in a default value to return if the resource is not located in the user's locale language. You can also pass in an explicit *locale* and the resource will be retrieved from that locale instead of detecting the user's set locale.
- `getRString(rFile, rKey, rLocale, default)` is also updated to include locale and default arguments.
- When resources are retrieved using the `getResource()` method and it does not exist, the plugin will now also show you which key was not found.
- `getfLocale()` and `setfLocale()` are 3 times faster than before.

```
// show a resource and if it does not exist, return a value
#getResource(name.button.Click)#

// Show the resource from the spanish locale, I am building a spanish email ala carte
<div>
#getResource(resource.locale.es)# Francisco,
</div>

Gracias!
</div>
```

JavaLoader Enhancements

- You can now call a method on the plugin called `getLoadedURLs()` so it can give you an array of all the paths that have been loaded by the JavaLoader library.
- Added the ability to load more jars or class files to an already running JavaLoader instance by using the method `appendPaths(dirPath, filter)`
- You can have a custom setting called `javaLoader.libpath` in your configuration file that can be a single location or an array of locations the java loader library will look for jar files to load.

XMLConverter plugin

You now also have our very own `XMLConverter` library that will transform ANY object into XML representation. And when we mean ANY object, we mean ANY object including ColdFusion 9 ORM entities. The `event.renderData()` method also leverages this converter in order to marshal data to XML. Also, this XML is waaaaay nicer than WDDX.

Interceptor Enhancements

- The methods `setNextEvent()` and `setNextRoute()` now execute a `postProcess` interception point. It can also be deactivated if need be.
- When you declare interceptors you can add a `name` attribute which uniquely identifies the declared interceptor. This allows you to declare more than once instance of an interceptor in different configurations. If no `name` is defined, then the name will be the name of the component.

```
// Declare interceptor
Interceptors = {
{name=AwesomeInterceptorClass=coldbox.system.interceptors.Awesome*
{name=MoreAwesomeInterceptorClass=coldbox.system.interceptors.Awesome*
properties={ awesomeCheck: configLoading: false }
}
}

<!-- OR XML-->
<Interceptor>
<InterceptName=AwesomeInterceptorClass=coldbox.system.interceptors.Awesome*
<InterceptName=MoreAwesomeInterceptorClass=coldbox.system.interceptors.Awesome*
<Propertyname=AwesomeCheck/Property>
<Propertyname=ConfigLoading/false/Property>
</Interceptors>
</Interceptors>
```

No More Inheritance

The major change to interceptors is that now they DO NOT need to inherit from the Core ColdBox Interceptor class if you like. You can now build simple CFCs and they will be decorated at runtime and adapted for usage within ColdBox. Ahh, doesn't that feel great. less coupling! Not only that, we are still backwards compatible and if you prefer to do inheritance, then go for it, we are not prejudice to your decision. You can still use ALL of the methods you are used to when building interceptors, we take care of the hard stuff to make your code look nicer and easier. We also create a virtual *super* scope for your inheritless components so you can still provide overrides. The name of the scope is called: `$$super` so you can talk to the base class.

```
..
* My awesome Interceptor
*/
component {
//Autowire Dependencies
property name=ServiceInject:

function configure(){
// configure my interceptor
}
function preProcess(event, interceptData){
}
}
```

New Interception Points

Interception Point	Description
preLayout	Occurs before any rendering is done or determined. Remember that the preRender point has the entire rendered content passed into it already. So this happens before rendering of content
preViewRender	Occurs before ANY view is executed. Keys received in the interception: <code>renderedView.view.cache.cacheTimeout.cacheLastAccessTimeout.cacheSuffix.module</code>
postViewRender	Occurs after the view has been executed and is awaiting to be rendered. Receives all the view rendering arguments and the view content that will be rendered. Keys: <code>renderedView.view.cache.cacheTimeout.cacheLastAccessTimeout.cacheSuffix.module</code>
preProxyResults	Occurs before the results of a proxy call are sent back to the external caller. This can be used for marshalling or transformation of data, logging, etc. Keys received are: <code>proxyResults</code>
afterModelCreation	Occurs after ANY model object gets created and can be used to post-process objects after creation and dependency injections.
onReinit	Occurs right before the application will be reinitialized.
applicationStop	Fires whenever the ColdFusion application stops. A great place to shutdown services such as EHCACHE, etc.
beforeDebuggerPanel	Fires right before the debugger panels get rendered in debug mode
afterDebuggerPanel	Fires right after the last debugger panel gets rendered in debug mode.
onRequestCapture	Fires right after the request is captured by the framework but before the event is detected, flash ram inflated and event caching detected. A great way to add new elements to the request collection so event caching can have a depends on reaction.
onInvalidEvent	Fires whenever an invalid event is detected and before the <code>onInvalidEvent</code> handler is called (if any). This interception point receives the following: <code>[invalidEvent]</code>

Extending The Debugger

Thanks to two new interception points: `beforeDebuggerPanel`, `afterDebuggerPanel`, you can easily build interceptors that can act as debugger panels in your ColdBox applications. Time to start getting creative folks!

New Object Properties

All interceptors will have certain objects in the `variables` scope available to them for usage. The following is a table listing of all the objects you can use:

Property	New	Description
logBox	Yes	The reference to the LogBox library
log	Yes	A pre-configured LogBox Logger object for this specific class object
flash	Yes	A reference to the current configured Flash Object Implementation that inherits from the AbstractFlashScope (<code>coldbox.system.web.flash.AbstractFlashScope</code>)
controller	No	The main application ColdBox controller (<code>coldbox.system.web.Controller</code>)

eventPattern annotation

You can now add a new metadata argument (annotation) called `eventPattern` to any interceptor method in order to tell the framework that it should execute that interception point ONLY if the incoming `event` matches that regular expression. Let's say that you have an interceptor with a `preProcess` method and you would like to provide security with it, but only to events that are in the `admin` package. Then I can do this:

```
/**
 * @eventPattern ^admin
 */
function preProcess(event, interceptData){
}

//OR
<cffunction name=preProcess output=false eventPattern=^admin>
</cffunction>
```

The `eventPattern` annotation is a regular expression that MUST match the incoming `event` variable. If it does, then the interception point fires, else it is ignored.

New Interceptor: ReactorLoader

The reactor extras have been expanded and a new interceptor has been born called: `!ReactorLoader`. This nice interceptor will configure your application to use Reactor and cache it in the ColdBox cache. Easily configure your app for Reactor ORM

Interceptor Properties:

Property	Type	Required	Default	Description
<code>dsnAlias</code>	string	true	N/A	The datasource alias name to use, as defined in your datasources section of your config. Make sure you define the dbtype also
<code>pathToConfigXML</code>	string	true	N/A	The path of the Reactor config file
<code>project</code>	string	true	N/A	The name of the Reactor project
<code>mapping</code>	string	true	N/A	The relative path or mapping to the directory where reactor will write generated files
<code>mode</code>	string	true	N/A	The reactor mode: <i>always, development, production</i>
<code>ReactorCacheKey</code>	string	false	Reactor	The key used to store Reactor as a singleton in the coldbox cache. (case sensitive)
<code>ReactorConfigClassPath</code>	string	false	reactor.config.config	The class path override of the reactor configuration object
<code>ReactorFactoryClassPath</code>	string	false	reactor.reactorFactory	The class path override of the reactor factory object

So just remember to add the datasource element with a valid **dbType** as Reactor needs it; Options are:

- mssql - Microsoft SQL Server 2000 and 2005.
- mysql - MySQL 3
- mysql - MySQL 4
- postgresql - PostgreSQL 8
- db2 - IBM DB2
- oracle - Oracle 9i and 10g
- oraclerd - Oracle RDB (this is not officially supported)

Sample:

```
<DataSources>
<DataSource alias=blogDSN name=simpleblog dbtype=mysql/>
</DataSources>

<Interceptors>
<Interceptor class=coldbox.system.orm.reactor.ReactorLoader*
  <Property name=dmlia=blogDSN/Property>
  <Property name=patHTOConfig=ConfigReactor.xml.of/Property>
  <Property name=project=MyBlog/Property>
  <Property name=mapping=AppMappingMode/Property>
  <Property name=mode=Development/Property>
</Interceptor>
</Interceptors>
```

This declaration will create the following objects in the ColdBox cache:

Cache Key	Object
Reactor	The reactor factory object

So do you still think there is more? Well, there is no more, that's it, a few lines of code and you are cooking with Reactor.

Autowire Interceptor Updates

The autowire interceptor has been updated to allow for the configuration of dependency injection of ORM entities. This feature ONLY works if you are using our new ORM Event Handler to ColdBox bridge. If enabled you can configure the following settings:

Property	Type	Default	Description
entityInjection	Boolean	false	Enable the entity injection events
entityInclude	List	[empty]	A list of entity names to include in the injections ONLY!
entityExclude	List	[empty]	A list of entity names to exclude from dependency injection!

Security Interceptor Updates

- The property **useRoutes** has now been deprecated as it is automatically calculated.
- You can now add a new element called **useSSL** to the XML or query rules so the interceptor will use SSL or not to redirect or secure the rule.

```
<rule>
<whitelist user\.login,user\.logout,.*></whitelist>
<securelist user\.*,.*></securelist>
<role admin/roles>
<permissions permissions>
<redirect user\.login/redirect>
<useSSL false/useSSL>
</rule>
```

- You can now also add ANY element to a rule and it will be read and passed to the validators for usage:

```
<rule>
<whitelist user\.login,user\.logout,.*></whitelist>
<securelist user\.*,.*></securelist>
<role admin/roles>
<permissions permissions>
<redirect user\.login/redirect>
<useSSL false/useSSL>
<!--Custom Elements-->
<filter math,cookie,if/filters>
<CustomCheck math/CustomChecks>
</rule>
```

Deploy Interceptor Updates

- New property: **deployCommandModel** that points to a model alias (object) that will be used as the deployment command object. This is the object that gets fired when a new deployment gets detected.

SES Interceptor Updates

- **addCourse()** is now deprecated and replaced by **addRoute()** as the standard method to define URL mapping routes
- New **view dispatcher**: You can now declare views to be dispatched with no need of running events in SES mode. This is accomplished via our new **view** argument to the **addRoute()** method. You can even use the **viewNoLayout** to tell the route to skip a layout, else the argument is always false.

```
// view dispatch, no event execution
addRoute(pattern=about,views=about,viewNoLayout=true);
```

- Ability to turn on/off convention name value pairs on a per route basis by adding a new argument to the **addRoute()** method: **valuePairTranslation** (boolean), which defaults to **true**.
- You can now declare a **route** pattern that indicates the default event instead of defining one in the configuration file.
- Performance updates are now even visible due to the evaluation of regex patterns at application startup instead of at runtime and for each route permutation.
- New method to allow for routes to be reloaded in every request. Great for development or on the fly routes: **setAutoReload(true)**
- You can now build **alpha** only placeholders by using **-alpha** on them.

```
addRoute(pattern=name-alpha#handler=show);
```

- Ability to do REST URLs based on HTTP Methods. You can do this by passing a JSON string or structure to the **action** argument

```
// Easily split actions on a handler according to HTTP methods detected.
addRoute(pattern=api/user/:username*
  handler=rest.User*
  actions={
    GET #show;
    DELETE #remove;
    POST #create;
    PUT #update;
  });
```

- You can now make the placeholders adhere to your own regular expressions by using the **constraints** argument which is a JSON structure or CF structure of name value pairs that match the placeholder name.

```
// route with custom constraints
addRoute(pattern=api/:format/*
  handler=api;
  actions=execute;
  constraints={
    format={xml|json}*
  });
```

Important: The value of the constraint MUST be enclosed in parenthesis ()

- New ses regex dsl for placeholders: **/regex:(full regex here)/**

This gives the user the ability to use ANY regular expression for any placeholder or for the entire URL string by just writing the namespace: **regex:()** and then placing the regular expression within the parenthesis. This is similar to the constraints functionality but targeted directly to a placeholder reference within the pattern. This is more readable and directed but it does not set the regular expression to a variable. It just identifies and matches.

```
// route with custom constraints
addRoute(pattern=api/regex:{"(xml|json)"}/*
  handler=api;
  actions=execute);
```

- SES interceptor now by default does **URL extension detection** via the incoming routed URL. You can turn on/off by using the new **setExtensionDetection()** method. So if an incoming URL is:

```
http://myapp.com/users/john.xml
or
http://myapp.com/users/john.json
```

The interceptor will see that an extension is requested and actually create a request collection variable for you called **format** with the value of the extension: **format=xml, format=json**. The interceptor also allows for you to setup the valid extensions it should recognize. By default the interceptor recognizes the following extension list:

```
xml,json,jsont,html,htm,rsx
```

You can change this list by using the new method: **setValidExtensions()**

```
// auto detect extensions, you don't have to do this as the default is true.
setExtensionDetect=true;

// Set xml and json as the only valid extensions for routing.
setValidExtensions(xml,json);
```

If an invalid extension is detected, the interceptor will throw a **403 Invalid Format Exception**, so make sure you can trap it and account for it.

- The default route of **handler/action?** now supports package resolution for modules. Before if we wanted a module call we would do something like:

```
http://mysite.com/module:handler/action
or
http://mysite.com/module:package.handler/action
```

The usage of the **:** to demarcate modules is ok, but not URL friendly or nice, so we updated the package resolver in the SES interceptor and automatically we can now do this

```
http://mysite.com/module/handler/action
or
http://mysite.com/module/package/handler/action
```

It will discover the right paths according to what is sent in the URL.

- You can now modify the URI used by the SES interceptor before the SES interceptor tries to identify a route.

This feature can be useful to set flags for each request based on a URL and then clean or parse the URL to a more generic form to allow for simple route declarations. Uses may include internationalization (i18n) and supporting multiple experiences based on devices such as Desktop, Tablet, Mobile and TV.

To modify the URI used by the SES interceptor before route detection occurs simply follow the convention of adding a UDF called `PathInfoProvider` to your `Routes.cfm` file. The `PathInfoProvider()` UDF is responsible for returning the string used to match a route. Without this UDF the SES interceptor will simply obtain the URI from the `CGLPATH_INFO` variable.

```
// Example PathInfoProvider for detecting a mobile request
function PathInfoProvider(Event){
    var rc = Event.getCollection();
    var prc = Event.getCollectionProperty('prc');

    local.URI = CGI.PATH_INFO;

    if (reFindNoCase(/m/,local.URI) == 0)
    {
        // Does not look like this could be a mobile request...
        return local.URI;
    }

    // Mobile Request? Let's find out.

    // If the URI is "/m" it is easy to determine that this is a
    // request for the Mobile Homepage.
    if (len(local.URI) == 2)
    {
        prc.mobile = true;
        // Simply return "/" since they want the mobile homepage
        return "/";
    }

    // Only continue with our mobile evaluation if we have a slash after
    // our "/m". Without a slash following the /m the route is something
    // else like coldbox.org/makes/cool/stuff
    if (reFindNoCase(/m/,local.URI) == 1)
    {
        // Looks like we are mobile!
        prc.mobile = true;

        // Remove our "/m/" determination and continue
        // processing for languages...
        local.URI = REReplaceNoCase(local.URI, "/");
    }

    // The URI starts with an "m" but does not look like
    // a mobile request. So, simply return the URI for normal
    // route detection...
    return local.URI;
}
```

• `addRoute(ssl=Boolean)` has been added to enable the restrictions of certain routes to SSL ONLY. By default, SSL is disabled in all routes. If you pass `ssl=true` on a route, then that route is only accessible via SSL. If the route is matched and not in SSL, then the interceptor will redirect itself back but in SSL.

Event Handler Enhancements

No More Inheritance

The major change to handlers is that now they DO NOT need to inherit from the Core ColdBox Event Handler if you like. You can now build simple CFCs and they will be decorated at runtime and adapted for usage within ColdBox. Ahh, doesn't that feel great, less coupling! Not only that, we are still backwards compatible and if you prefer to do inheritance, then go for it, we are not prejudice to your decision. You can still use ALL of the methods you are used to when building event handlers, we take care of the hard stuff to make your code look nicer and easier. Also, if you declare a constructor in your handler, ColdBox will call it for you. We also create a virtual *super* scope for your inheritless components so you can still provide overrides. The name of the scope is called: `Super` so you can talk to the base class.

```
**
* My awesome Handler
*/
component {
    //Autowire Dependencies
    property name=servicesInject;

    function init(){
        // do some init stuff
        return this;
    }
}
```

Persistence Metadata

You can now use the `singleton` annotation on the `cfc` component tag to denote that this object will live for the entire application. This is the same as adding `cache=true cacheTimeout=0` to the component tag:

```
**
* @singleton=true
*/
component {
}

// OR
<<cfc component singleton=true />
```

New Object Properties

All handlers will have certain objects in the `variables` scope available to them for usage. The following is a table listing of all the objects you can use:

Property	New	Description
<code>logBox</code>	Yes	The reference to the LogBox library
<code>log</code>	Yes	A pre-configured LogBox Logger object for this specific class object
<code>flash</code>	Yes	A reference to the current configured Flash Object Implementation that inherits from the AbstractFlashScope (<code>coldbox.system.web.flash.AbstractFlashScope</code>)
<code>controller</code>	No	The main application ColdBox controller (<code>coldbox.system.web.Controller</code>)

Pre/Post Interceptors Updated

These local interceptors now take in the current executing action they are intercepting as a new argument called: `action` and `eventArguments` (if passed)

```
<!-- preHandler -->
<cfunction name=preHandler returnType=void output=false hint=Executes before any event in this handler*
<cfargument name=event*required=true
<cfargument name=action*hint=The intercepted action*
<cfargument name=eventArguments*hint=The called runEvent() event arguments*
</cfscript>
var rc = event.getCollection();
</cfscript>
</cfunction>

<!-- postHandler -->
<cfunction name=postHandler returnType=void output=false hint=Executes after any event in this handler*
<cfargument name=event*required=true
<cfargument name=action*hint=The intercepted action*
<cfargument name=eventArguments*hint=The called runEvent() event arguments*
</cfscript>
var rc = event.getCollection();
</cfscript>
</cfunction>
```

Targeted pre/post action Methods

You can now declare targeted pre/post action methods for any action in the same event handler. This gives you the ability to target a specific action's before and after interception points. Let's work on the premise of an action function called `listUsers`:

```
function listUsers(event){
    //list users
}
```

Now let's target its before and after intercepting points

```
function preListUsers(event){
    // executes before the "listUsers" method
}

function listUsers(event){
    //list users
}

function postListUsers(event){
    // executes after the "listUsers" method
}
```

These are a great way to target individual action's before and after intercepting points.

New implicit handler methods: `aroundHandler()`, `around(Action)`

We had a way to intercept before and after a handler action, but what about the whole action or around the action. What this means is that you will create a method in your handler called `aroundHandler()`. If that method is detected, then the framework will call it instead of ANY action in your handler and it will be YOUR responsibility to delegate the action to execute or totally ignore the execution of such action. This is a concept called an *around advice* (AOP).

"Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception." - by Spring

In other words we can get very creative now and completely change the behavior of actions without actually touching them. Let's say I wanted to create an around handler that could provide safe transactions around the entire method. How would I do that? What if I wanted to have a try/catch block for each action? Let's say a handler has 20 actions, that's 20 try/catch blocks? These are all cross-cutting concerns that could be achieved cleanly by decorating the method targeted for execution, thus *aroundHandler*. Let's give the argument list first and then let's see how it works.

Argument	Required	Type	Default	Description
<code>event</code>	true	RequestContext	---	The request context
<code>targetAction</code>	true	UDF	---	The UDF that was requested for execution

```

eventArguments false struct {} The event arguments structure (if passed)

```

Let's build an `aroundHandler` that provides transactions, also remember that event handler calls can return data, so check for it also.

```

function aroundHandler(event, targetAction, eventArguments){
    // start transaction
    transaction{
        // execute the targeted action
        results = arguments.targetAction( arguments.event );
    }
    // check for results and send back if found
    if( NOT isNull(results) ){
        return results;
    }
}

```

That's it, I just transactioned all of my event action calls. What if I wanted consisted exception handling:

```

function aroundHandler(event, targetAction, eventArguments){
    try{
        // execute the targeted action
        results = arguments.targetAction( arguments.event );
    }
    catchAny e{
        //log it
        log_errorError executing #getMetadata(targetAction).name#: #e.detail#, #e.message#
        // place exception on request collection for display
        rc.exception = e;
        // Take them to an error page screen for nice display.
        event.setView#exceptions/generik#
    }
    // check for results and send back if found
    if( NOT isNull(results) ){
        return results;
    }
}

```

WOW! Powerful huh!! You can even use it on targeted action methods, basically `around(action)`.

```

function aroundListUsers(event, targetAction, eventArguments){
    try{
        // execute the targeted action
        results = arguments.targetAction( arguments.event );
    }
    catchAny e{
        //log it
        log_errorError executing ListUsers: #e.detail#, #e.message#
        // place exception on request collection for display
        rc.exception = e;
        // Take them to an error page screen for nice display.
        event.setView#exceptions/generik#
    }
    // check for results and send back if found
    if( NOT isNull(results) ){
        return results;
    }
}

```

We also introduced two handler properties that should seem very familiar as they are very similar to the `except` and `only` lists for pre/post handlers:

- `this.aroundHandler_only`: A list of actions to advice ONLY
- `this.aroundHandler_except`: A list of actions to NOT advice

Let's say you wanted an around advice for only the actions called: `save, delete, commit`:

```
this.aroundHandler_only=save,delete,commit"
```

We hope you enjoy our around advices as much as we do.

New implicit handler method: `onError()`

You can now create another implicit method on your event handlers with the following signature:

```
function onError(event, faultAction, exception){
}
```

This method will be executed if there is ANY exception executing the current event handler. This can give you very nice controlled exceptions at a handler or base handler level.

```

// Implicit Error Handler
function onError(event, action, exception){
    var validator = getInterceptor#onValidator#
    var append = structnew();

    // Log Error
    append.exception = arguments.exception;
    logger.fatal#Error executing resource: #event.getResource#. Error: #arguments.exception.message# validator.getLogJSON(append);

    // Send error to screen
    validator.renderError(message=arguments.exception.message# #arguments.exception.detail#,
        code=500,
        format=arguments.event.getValue#,
        event=arguments.event);
}

```

Handler Action HTTP Security

All event handlers can now declare a public variable called `allowedMethods` (struct) that contains a map of allowed methods and their appropriate HTTP methods. This has been done to provide a nice way to secure RESTful API's built on ColdBox. You can now control what HTTP method operations can be done on specific handler actions. Let's say we want to protect our actions on our handler, `delete()`, `add()` and `index()` with different http methods.

Action	Allowed Methods
delete	delete
add	get,post
index	get

```

//public property to allow method executions
this.allowedMethods = {
    delete:delete,
    add:get,post,
    index:get
};

```

New Implicit Handler: `MissingTemplateHandler`

You can now declare a coldbox setting called: `missingTemplateHandler` that points to an event. This event will be fired whenever the framework detects a missing template call in your application, that is for ANY cfm template. The request context will have the missing template value in a variable called `missingTemplate`.

```

// Declare the missing template handler in your Coldbox.cfc or your coldbox.xml.cfm
coldbox = {
    missingTemplateHandler=main.missingTemplate
};
// XML
<Settings>
<Settingname=missingTemplateHandlervalue=main.missingTemplate#
</Settings>

// The handler event
function missingTemplate(event){
    var rc = event.getCollection();

    rc.page = pageService.findByName( rc.missingTemplate );

    if( rc.page.isValid() ){
        // Render the page
        event.setView#page.show#;
    }
    else{
        // invalid page
        setNextEvent#page.invalid#;
    }
}

```

New Implicit Handler: `applicationEndHandler`

You can now declare a coldbox setting called: `applicationEndHandler` that points to an event. This event will be fired whenever the ColdFusion Application expires or stops.

```

// Declare the missing template handler in your Coldbox.cfc or your coldbox.xml.cfm
coldbox = {
    applicationEndHandler=main.onAppEnd
};
// XML
<Settings>
<Settingname=applicationEndHandlervalue=main.onAppEnd#
</Settings>

// The handler event
function onAppEnd(event){
    var rc = event.getCollection();

    //maybe shutdown things
    getMode#EHCACHE#.shutdown();
}

```

}

runEvent() Event Arguments

The `runEvent()` method has been added a new argument called `EventArguments`, which is a structure of arguments that it will passthrough to the event executed. This is a great way to pass arguments to these event methods without affecting other callers or the request collections. A great use case is for building and rendering viewlets. You can pass direct arguments to these viewlets events that will only be encapsulated for that `runEvent()` call.

Here are some examples: **Execution Calls:**

```
<div#runEvent(event=viewlets.userInfoEventArguments={format:compact$false})#</div>
<div#runEvent(event=viewlets.userInfoEventArguments={compact:true})#</div>
```

As you can see, I make the same `runEvent()` call to the same event: `viewlets.userInfo` but with different event arguments. This way, the calls are unique and traceable.

Receiving Calls

```
function userInfo(any event, any format):boolean compact$false {
  // do something, but it can use more arguments now
}
```

The event action method: `userInfo` now takes in not only the required `event` argument, but a la-carte arguments. Just make sure you can setup defaults for these arguments or check their existence.

Model Integration Enhancements (WireBox)

- The model integration framework in ColdBox is now called **WireBox** and it will be standalone for use outside of ColdBox if you so wanted to.
- You can now use the keyword `singleton` in the `cfcomponent` tag to simulate a singleton instance.
- New interception point: `afterModelCreation` that can be used to post-process objects after creation and dependency injections.
- Factory now respects defaults on constructor arguments
- Factory now ignores objects that cannot be located or found but logs them
- Debug mode is now enhanced to reflect more data via `logInfo`
- `getModel()` has new arguments
 - `executeInit=true`: Whether to execute the constructor or not.
 - `id`: You can optionally pass in a DSL string instead of a name to create a domain object
- A backup namespace has been added to package scanning of path `/`. This means that WireBox can now by default locate ANY object in the server just like `createObject()` does but with autowiring goodness.

New/Updated Settings

The new/updated settings you can use when configuring the ColdBox Model Integration:

- **DefinitionFile**: By default points to `config/ModelMappings.cfm`. You can use this setting to tell WireBox where to find the configuration file.
- **ExternalLocation**: It is now a list of invocation paths that the WireBox bean factory will search CFCs for. Please remember that order of declaration is important as that will be the order of search.

```
// Programmatic approach
// Model Integration
models = {
  objectCaching:true
  definitionFile:"config/ModelMappings.cfm"
  externalLocation:wirebox.testing.testmodel,shared,transfer*
  setterInjection:false
  DICompleteUDF:"endIComplete"
  StopRecursion:"$"
```

```
// XML Approach
<Models>
  <DefinitionFile#config/ModelMappings.cfm#definitionFile>
  <SetterInjection#false#setterInjection>
  <ObjectCaching#true#objectCaching>
  <ExternalLocation#externalLocation>
  <DICompleteUDF#endICompleteUDF>
  <StopRecursion#stopRecursion>
</Models>
```

Dependency DSL Metadata

The dependency DSL metadata marker has been standardized to the keyword `inject`. You can now use this metadata annotation on

- `cfproperty` tags
- constructor arguments
- setter methods

```
// some examples
property name=userService inject model:
// name as inject by default: references a model object
property name=userService inject:

// setter method
/**
 * @inject Model
 */
function setUserService(userService) {
  variables.userService = arguments.userService;
}

// Constructor arguments
<cffunction name=init output=false returntype=any>
<cfargument name=userService type=any inject=model>
</cffunction>
```

Updated Dependency Injection DSL

More updates will be shown once the [WireBox Documentation](#) is finalized. Below are some new autowire DSL types:

Type	Description
cachebox	Get a reference to the application's CacheBox instance
cachebox:{name}	Get a reference to a named cache inside of CacheBox
cachebox:{name}:{objectKey}	Get an object from the named cache inside of CacheBox according to the <i>objectKey</i>
coldbox:moduleService	Get a reference to the ColdBox Module Service
coldbox:interceptor:{name}	Get a reference of a named interceptor
coldbox:fwConfigBean	Get a configuration bean object with ColdBox settings instead of Application settings
coldbox:fwSetting:{setting}	Get a setting from the ColdBox settings instead of the Application settings
javaLoader:{class}	Create an object from the JavaLoader plugin and its set of loaded java libraries
entityService	Inject a <i>BaseORMService</i> object for usage as a generic service layer
entityService:{entity}	Inject a <i>VirtualEntityService</i> object for usage as a service layer based off the name of the entity passed in.
logbox	Get a reference to the application's LogBox instance
logbox:root	Get a reference to the root logger
logbox:logger:{category name}	Get a reference to a named logger by its category name

```
// some examples
property name=cachebox inject=logbox
property name=rootLogger inject=logbox:root
property name=logger inject=logbox:logger:model.com.UserService
property name=moduleService inject=coldbox:moduleService
property name=producer inject=coldbox:interceptor:MessageProducer
property name=configBean inject=coldbox:fwConfigBean
property name=producer inject=interceptor:MessageProducer
property name=appPath inject=coldbox:fwSetting:ApplicationPath

// JavaLoader goodness
property name=BinaryHeap inject=javaLoader:org.apache.commons.collections.BinaryHeap
property name=mail inject=javaLoader:org.apache.commons.mail.SimpleEmail

// Generic ORM service layer
property name=genericService inject=entityService*
// Virtual service layer based on the User entity
property name=userService inject=entityService:User*
```

Request Context Enhancements

These are the updates on the request context object:

- `buildLink(linkTo, translate, ssl, baseURL, queryString)`: now supports a query string argument for building ses or non-ses query strings and also an ssl argument.
- The `setLayout(layout)` method can now accept the layout alias also instead of the file name. This is only valid if you have declared your layouts implicitly in your configuration file.
- The `setView()` now takes in two extra optional arguments:
 - `layout`: The layout to render the view in
 - `cacheSuffix`: You can now pass in a suffix to that will be added to the cache keys (maybe the host you are on?)

New Methods

- `getHTTPMethod()`: New method gives you the currently executing http method the request is coming from.
- `noExecution()`: New method that disables the execution of the main event.
- `getCurrentRoutedURL()`: Gives you the currently routed URL via SES
- `getCurrentModule()`: Gives you the name of the currently executing module (if any)
- `getCurrentRoute()`: Gives you the current matched SES URL Mapping route
- `getHTTPHeader(key, [default])`: Nice little tool to get you HTTP headers with default values
- `setHTTPHeader([status,code],[statusText],[name],[value],[charset])`: Nice little tool to set your HTTP headers, even mock them
- `isAjax()`: Reads the xmlhttp request header and determines if this is an Ajax request or not

Examples

```
// build link with query string
<a href=#event.buildLink(linkTo='users.list', queryString='userID=23001&click=4')#>
```

```

<a href=#event.buildLink(linkTo='users.list', queryString='userID=234&page=3Click/nope)#*
// Get HTTP method
if( event.getHTTPMethod() != 'GET' ){ //do something }
// Following in the coldbox.cfc config
layouts = [
{name='awesome' file='Layout_Awesome.cfm'
{name='login' file='Layout_login.cfm' folder='security'
];
// Then in the handler
event.setLayout('awesome')
// Same as
event.setLayout('Layout_Awesome')

```

event.noExecution()

This method is a very cool method to disable the main execution of the incoming event (if any). Why would you want to do this? Well, you can build interceptors that handle the requests instead of main events. This simulates building servlets that react to interceptors instead of events. The most common usage of this method is on handlers or interceptors that occur BEFORE the execution of the main event.

Example: I will build an interceptor that listens on *preProcess* method and stops execution to render back some data.

```

function preProcess(event, interceptData){
// Do some work, we intercepted the request
// Render data
event.renderData(data=data, type='');
// Stop main event execution, there will be none
event.stopExecution();
}

```

event.renderData() updated

Our coolest method of all gets even cooler!

- You can now manipulate some of the json and xml produced
- We built our very own XML marshaller which is lightweight and can even marshal objects
- You can now also send status codes and text to the header in one single shot
- We added new core type conversions: HTML, json and text.

Below is the new signature:

```
public void renderData(string type, any data, string contentType, [string encoding], [numeric statusCode], [string statusText], [string jsonCase], [string jsonQueryFormat], [boolean jsonAsText], [string xmlColumnList], [bool
```

Parameters

- **type** - The type of data to render. Valid types are JSON, JSOIN, XML, WDDX, PLAIN/HTML, TEXT. The default is HTML or PLAIN. If an invalid type is sent in, this method will throw an error
- **data** - The data you would like to marshal and return by the framework
- **contentType** - The content type of the data. This will be used in the c:content tag: text/html, text/plain, text/xml, text/json, etc. The default value is text/html. However, if you choose JSON this method will choose application/json, if you choose WDDX or XML this method will choose text/xml for you. The default encoding is utf-8
- **encoding** - The default character encoding to use
- **statusCode** - The HTTP status code to send to the browser. Defaults to 200
- **statusText** - Explains the HTTP status code sent to the browser.
- **jsonCase** - JSON Only: Whether to use lower or upper case translations in the JSON transformation. Lower is default
- **jsonQueryFormat** - JSON Only: query or array
- **jsonAsText** - If set to false, defaults content mime-type to application/json, else will change encoding to plain/text
- **xmlColumnList** - XML Only: Choose which columns to inspect, by default it uses all the columns in the query, if using a query
- **xmlUseCDATA** - XML Only: Use CDATA content for ALL values. The default is false
- **xmlListDelimiter** - XML Only: The delimiter in the list. Comma by default
- **xmlRootName** - XML Only: The name of the initial root element of the XML packet

```

event.renderData(data=>Page not found</, statusCode=404, statusMessage=Page Not Found*)
event.renderData(type=; data=query, xmlRootName=; xmlUseCDATA=;
event.renderData(type=json; data=query, jsonCase=);

```

ColdBox BootStrapper Enhancements

- Performance and loading updates
- Ability to change the locking timeouts: *setLockTimeout()*
- Ability to set the key to use in Application scope: *setColdbox_app_key()* or use the *coldbox_app_key* variables in Application.cfc
- Ability to set the *appMapping*: *setColdbox_App_Mapping()* or use the *coldbox_app_mapping* variables in Application.cfc

ColdBox Main Controller Enhancements

The main coldbox controller which is used by the *Application.cfc/cfm* now has exposed methods for the locking timeouts.

- *get/setLockTimeout()*

You can now also choose the name of the key used to store your running ColdBox application. By default it still uses the key: *cbController*, but now you can choose this by using the following methods:

- *setCOLDBOX_APP_KEY()* or
- Adding a variable called *COLDBOX_APP_KEY* in your *Application.cfc* declaration.
- Adding a variable called *COLDBOX_APP_KEY* in your template that includes the *coldbox.cfm bootstrapper*

The *setNextEvent()* has been updated to support multiple arguments:

Unit/Integration Testing Enhancements

- *cfcUnit* support has now been deprecated in favor of *MXUnit*.
- The *execute()* method calls now follow suite of the full request lifecycle:
 - Execute *ApplicationStartHandler*
 - Execute *preProcess* interceptors
 - Execute *RequestStartHandler*
 - Execute *event* in testing
 - Execute *RequestEndHandler*
 - Execute *postProcess* interceptors
- New public property for telling the testing framework to load a mock coldbox application or not when testing: *loadColdbox (boolean)*. This defaults to *true*. So if you want to use the *BaseTestCase* for unit testing model objects, you can do so very easily, by just saying *this.loadColdbox = false*.

```

function setup(){
this.loadColdbox = false;
super.setup();
}

```

- New annotations on the test case *cfcComponent* tag in order to tell it what application to test:

Annotation	Type	Required	Default	Description
appMapping	string	false	/	The application mapping of the ColdBox application to test. By default it maps to the root
configMapping	string	false	{appMapping}/config/Coldbox.cfc or xml	The configuration file to load for this test. If not defined, then by default it looks for the CFC configuration file first and if not found, it tests for the <i>coldbox.xml.cfm</i> second. It searches in the convention of the <i>config</i> folder.
loadColdBox	Boolean	false	true	If you call <i>super.init()</i> on the test case, this flag tells the base test case to load up the virtual testing application or not.

So now you can declare an integration test so much cleaner and nicer:

```

/**
 * Integration test for my Users.cfc handler
 * setup() is called implicitly from the inherited BaseTestCase, you can also override it.
 * @appMapping /my-nested-app
 */
component extend='coldbox.system.testing.BaseTestCase'
function testIndex(){
event = execute{@ers.index};
// Assert here
}
function testSave(){
// setup some data for the handler, FORM or URL
form.firstname='bula?
form.lastname='majanor'
// execute integration test
event = execute{@ers.nav};
//assert here
}
}

```

New Methods

There are some new methods to help you in your unit testing:

Method	Description
querySim()	enables you to simulate ANY query (Part of MockBox)
getMockBox()	Get a reference to MockBox
getMockRequestContext(clearMethods)	Builds a new request context object that you can use for mocking purposes
getMockModel(name, clearMethods)	Gives you back a mock model object
reset()	Clears the application scope of any leftover coldbox bits, when all else fails, call reset.
getFlashScope()	Gives you the current loaded flash scope object
getMockPlugin()	Gives you a plugin instance already mocked up and ready for usage or testing
getMockDataSource(name, alias, btype, username, password)	Gives you a mocked capable datasource object that maps to <i>coldbox.system.core.db.DataSourceBean</i>
getMockConfigBean(configStruct)	Gives you a mocked configuration bean with whatever data structure you want that maps to <i>coldbox.system.core.collections.ConfigBean</i>

New Targeted Plugin Testing

You can now test plugins directly with no need of doing integration testing. This way you can unit test plugins directly very very easily. All you need to do is the following:

1. Create a test class that inherits from `coldbox.system.testing.BasePluginTest`
2. Create a component annotation called `plugin` that equals the full path of the plugin to target for testing

This testing support class will create your plugin, and decorate with mocking capabilities via [MockBox](#), and mock all the necessary companion objects around plugins. The following are the objects that are placed in the `variables` scope for you to use:

- **plugin**: The target plugin to test
- **mockController**: A mock ColdBox controller in use by the target plugin
- **mockRequestService**: A mock request service object
- **mockLogger**: A mock logger class in use by the target plugin
- **mockLogBox**: A mock LogBox class in use by the target plugin
- **mockFlash**: A mock flash scope in use by the target plugin

All of the mock objects are essentially the dependencies of plugin objects. You have complete control over them as they are already mocked for you. We actually use this approach to test all shipped ColdBox plugins.

Basic Setup

```
/**
 * @plugin myApp.plugins.CoolPlugin
 */
component extends=coldbox.system.testing.BasePluginTest {
    // Just create test methods, no need to use the setup() method unless you want to:
    function setup() {
        super::setup();
        // test custom constructor
        plugin::init();
    }
}
```

Real Sample

```
<cfcomponent extends=coldbox.system.testing.BasePluginTest plugin=coldbox.system.plugins.HTMLHelper>
<cfscript>
function testAddAssetJS() {
    var mockEvent = getMockRequestContext();
    mockRequestService.@Context=mockEvent;

    // mock the plugin's htmlhead method
    plugin.$htmlhead;

    // Call method to test
    plugin.addAssetTest, 'ja.luis.js';

    debug( plugin.$callLog().$htmlhead);

    // test duplicate call
    assertEqual( script, 'test.js' type='text/javascript'</script><script src='luis.js' type='text/javascript'</script>($callLog().$htmlhead[1][1] );
    plugin.addAssetTest, 'ja';
    assertEquals( 1, arrayLen( plugin.$callLog().$htmlhead ) );
}

function testTableORM() {
    data = entityLoad( test );
    str = plugin.table( data=data, includeName=1 );
    debug( str );
    assertEquals( table

```

New Targeted Handler Testing

You can now test handlers directly with no need of doing integration testing. This way you can unit test handlers directly very very easily. All you need to do is the following:

1. Create a test class that inherits from `coldbox.system.testing.BaseHandlerTest`
2. Create a component annotation called `handler` that equals the full path of the handler CFC to target for unit testing
3. Create an optional annotation called `UDFLibraryFile` that will be the path of the UDF library file that is declared in your ColdBox configuration or any other UDF file you load dynamically in your handlers.

This testing support class will create your handler, and decorate with mocking capabilities via [MockBox](#), and mock all the necessary companion objects around handlers. The following are the objects that are placed in the `variables` scope for you to use:

- **handler**: The target handler to test
- **mockController**: A mock ColdBox controller in use by the target handler
- **mockRequestService**: A mock request service object
- **mockLogger**: A mock logger class in use by the target handler
- **mockLogBox**: A mock LogBox class in use by the target handler
- **mockFlash**: A mock flash scope in use by the target handler

All of the mock objects are essentially the dependencies of handler objects. You have complete control over them as they are already mocked for you.

Basic Setup

```
/**
 * @handler myApp.handler.User
 * @UDFLibraryFile /myApp/includes/helpers/AppHelper.cfm
 */
component extends=coldbox.system.testing.BaseHandlerTest {
    // Just create test methods, no need to use the setup() method unless you want to:
    function setup() {
        super::setup();
        mockUserService = getMockBox().createEmptyMockModel, user.UserService;
        // wire in my mock dependencies as this handler already has mocking capabilities
        handler.$property( 'Service' variables=mockUserService );
    }
}
```

New Targeted Interceptor Testing

You can now test interceptors directly with no need of doing integration testing. This way you can unit test interceptors directly very very easily. All you need to do is the following:

1. Create a test class that inherits from `coldbox.system.testing.BaseInterceptorTest`
2. Create a component annotation called `interceptor` that equals the full path of the handler CFC to target for unit testing
3. Create an optional structure variable in the `variables` scope called `configProperties` in the `setup()` method before your `super.setup()` method, if you would like to test the interceptor with your configuration properties structure.

This testing support class will create your interceptor, and decorate with mocking capabilities via [MockBox](#), and mock all the necessary companion objects around interceptors. The following are the objects that are placed in the `variables` scope for you to use:

- **interceptor**: The target interceptor to test
- **mockController**: A mock ColdBox controller in use by the target interceptor
- **mockRequestService**: A mock request service object
- **mockLogger**: A mock logger class in use by the target interceptor
- **mockLogBox**: A mock LogBox class in use by the target interceptor
- **mockFlash**: A mock flash scope in use by the target interceptor

All of the mock objects are essentially the dependencies of interceptor objects. You have complete control over them as they are already mocked for you.

Basic Setup

```
/**
 * @interceptor myApp.interceptors.Security
 */
component extends=coldbox.system.testing.BaseInterceptorTest {
    // Just create test methods, no need to use the setup() method unless you want to:
    function setup() {
        configProperties = {
            roles 'admin,user,moderator'
            security 'active'
        };
        super::setup();
        mockSecurityService = getMockBox().createEmptyMockModel, SecurityService;
        // wire in my mock dependencies as this interceptor already has mocking capabilities
        interceptor.$property( 'SecurityService' variables=mockSecurityService );
        // we are now ready to test this interceptor
    }
}
```

Real Example

```
<cfcomponent extends=coldbox.system.testing.BaseInterceptorTest interceptor=coldbox.system.interceptors.Deploy>
<cfscript>
function setup() {
    super::setup();
    // mocks
    mockController.$(AppRootPath).expandPath( coldbox.testharness;
    interceptor.$setProperty( 'File', 'config/.deploy.thg';
    interceptor.$locateFilePath( 'config/.deploy.thg' );
}

function testConfigure() {
    interceptor.$configure();
}

function testAfterAspectsLoad() {
    mockLogger.$info();
    interceptor.$afterAspectsLoad( getMockRequestContext() );
}

function testPostProcess() {
    // mocks
    mockController.$(ColdBoxInitialDate).$set( 'setColdBoxInitialDate', 'setAspectsInitialDate';
    testDate = now();
    mockLogger.$info();
    interceptor.$setProperty( 'File', 'instance.config/.deploy.thg';
    interceptor.$property( 'DeployCommandObject', 'instance' );
}
```

```

// Test no setting
interceptor.$settingExists>false.$('configure');
interceptor.postProcess(getMockRequestContext());
assertEquals( 1, arrayLen(interceptor.$callLog().configure) );

// Test setting exists but same date
interceptor.$settingTestDate.$fileLastModifiedSetDate.$settingExists>true;
interceptor.postProcess(getMockRequestContext());
assertEquals( 0, arrayLen(mockController.$callLog().setColdboxInitiated) );

// Test it works
interceptor.$settingTestDate.$fileLastModifiedSetDate+10.$settingExists>true;
interceptor.postProcess(getMockRequestContext());
assertEquals( 1, arrayLen(mockController.$callLog().setColdboxInitiated) );

}
</cfscript>
</cfcomponent>

```

New Targeted Model Testing

You can now test model objects directly with no need of doing integration testing. This way you can unit test model objects very very easily using great mocking capabilities. All you need to do is the following:

1. Create a test class that inherits from *coldbox.system.testing.BaseModelTest*
2. Create a component annotation called **model** that equals the full path of the model object CFC to target for unit testing

This testing support class will create your model object, and decorate with mocking capabilities via [MockBox](#), and create some mocking classes you might find useful in your model object unit testing. The following are the objects that are placed in the *variables* scope for you to use:

- **model**: The target model object to test
- **mockLogger**: A mock logger class
- **mockLogBox**: A mock LogBox class

Basic Setup

```

/**
 * @model myApp.model.User
 */
component extends=coldbox.system.testing.BaseModelTest {
    function setup() {
        super.setup();
        user = model;
    }
    function testIsActive() {
        assertEquals(false user.isActive());
    }
}

```

Performance Updates

- Lock optimizations on ColdBox dispatcher have improved performance and avoids lock downs.
- Railo now runs ColdBox in multi-threaded mode
- preEvent security optimized
- SES expressions are now evaluated at startup instead of in EACH request, extreme improvement on performance for SES translations.

Categories:

- [what's new](#)