

[<< Back to Dashboard](#) | [<< Plugins Viewer](#)

ColdSpring Integrations Guide

Contents

Introduction

ColdBox provides you with a great set of tools and configurations to integrate ColdSpring into your applications. If you are not familiar with ColdSpring then you better get up to date. It is the premier Inversion of Control framework for Coldfusion written by Dave Ross and company.

What is Coldspring?

ColdSpring is a framework for CFCs (ColdFusion Components). ColdSpring's core focus is to make the configuration and dependencies of your CFCs easier to manage. ColdSpring uses the "inversion-of-control" pattern to "wire" your CFCs together. Inversion-of-control provides many advantages over traditional approaches to assembling your application's model. Also part of ColdSpring is the first Aspect-Oriented-Programming (AOP) framework for CFCs.

To read more about Coldspring go here:

- [ColdSpring's Main Website](#)
- [ColdSpring QuickStart Guide](#)
- [Documentation](#)

Where to start?

Now that we are down with the formalities. In order to activate ColdBox's IoC features you will have to declare three settings in your configuration file. You can [look at the config guide](#) for more in depth tutorial. Below are the three basic settings:

Setting	Description
IOCFramework	The named framework to use: coldspring or lightwire.
IOCFrameworkReload	Boolean variable that tells the framework to reload the factory on each request. Great for development. Defaults to false.
IOCDefinitionFile	This is the location of the coldspring xml file to load with the factory.
IOXObjectCaching	This settings is a boolean setting that defaults to FALSE. This tells the ioc plugin to cache the objects created by the factory in the ColdBox Cache if the correct metadata has been set in the component. Please read the advanced topic on caching below.

What get's passed to the ColdSpring Factory

ColdBox creates the Coldspring factory for you according to your settings, it will pass the configuration file you specified and the entire configuration structure. This will facilitate the developer, in which he/she can use any of the config's settings in their configuration files.

How can I access the configurations?

The entire configuration structure get's passed to the coldspring factory. So read the [Config Guide](#) to see all of the keys that get sent in.

Config Variables

Here is an example of using the config's variables. Let's say you want to use some custom settings you set in your coldbox.xml.cfm

```
<bean id="UserService" class="model.userService" >
  <property name="roles" ><value> ${UserRoles} </value> </property>
  <property name="cache" ><value> ${UserCaching} </value> </property>
</bean>
```

Important Note: The approach of complex variable calls via coldspring: \${Datsources.coldboxreader} works on ColdSpring 1.0

- [ColdSpring Integrations Guide](#)
 - [Introduction](#)
 - [What is Coldspring?](#)
 - [Where to start?](#)
 - [What get's passed to the ColdSpring Factory](#)
 - [How can I access the configurations?](#)
 - [Config Variables](#)
 - [Configuration Overrides](#)
 - [How Do I get the Beans in my Handlers?](#)
 - [Autowiring Your Handlers](#)
 - [The ColdBoxFactory.cfc : Leveraging Coldspring Custom Factories](#)
 - [Datasource Bean](#)
 - [Interceptor Service](#)
 - [Request Service](#)
 - [Debugger Service](#)
 - [ColdBox Applications are IoC Framework Agnostic](#)
 - [Advanced Topic: IOXObjectCaching](#)
 - [Setting Caching Parameters in your Beans](#)
 - [The Caching Parameters](#)
 - [Caching Declaration Examples](#)
 - [Guide Conclusion](#)

only, as future versions have disabled this feature.

You can use settings like:

- EnableBugReports
- OwnerEmail
- IOCFramework
- Layouts (Structure)
- WebServices (Structure)
- i18n (Structure)
- ColdBoxLogsLocation
- UDFLibraryFile
- DebugMode
- Environment (Very Useful)
- (All of YourSettings via their name attribute)
- BugTracerReports (Structure)
- MailServerSettings (Structure or use values to define a coldbox mail settings bean)

and many more...

Configuration Overrides

Since version 2.6.0 you can declare some extra settings in your configuration file that will be picked up by the IoC plugin. You can now override the ColdSpring or LightWire factory bean instantiation path to use by setting the following in your Custom Settings in your configuration file.

- **ColdSpringBeanFactory** : Instantiation path to the ColdSpring factory.
- **LightWireBeanFactory** : Instantiation path to the LightWire factory.

```
<YourSettings>
  <Setting name="ColdspringBeanFactory" value="frameworks.coldspring1_2.beans.DefaultXMLBeanFactory" />
</YourSettings>
```

How Do I get the Beans in my Handlers?

The main entry point or facade to the IoC Framework is the **ioc plugin**. This plugin gets configured by the framework and is ready for usage by the coldbox event handlers, plugins or interceptors. See [Live API](#) for all the plugin's methods. Below are some of them:

Method	Description
getBean(string beanName)	To get a bean via the declared framework
getExpandedIOCDefinitionFile()	Get the expanded location of the configuration file
getIOCDefinitionFile()	Get the configuration file as defined in the config.
getIoCFactory()	Returns the IoC Factory instance in use.
getIOCFramework()	Returns the name of the IoC Factory in use.
reloadDefinitionFile()	Reloads the configuration file in the IoC Framework.

You can also programmatically set the properties of this plugin. Why would you want to do this? Well, you might need to create other ioc plugin factories with different configuration files and manage even more model options. You can get a new ioc plugin, configure it and store it in the coldbox cache manager. You can then use it throughout your application. This is extremely powerful. You can programmatically load an Nth amount of configuration files or load the necessary files according to your needs. The possibilities here are endless. So how do I do this? Well, the ColdBox plugin factory provides with the **getPlugin()** method in which you can state that you want a new instance, not the one that is already cached. So to get a new IoC plugin, configure it and cache it, you would do the following:

```
<cfscript>
//Get a new ioc instance plugin
var oMyCS= getPlugin(plugin= "ioc",newInstance= "true");

//Configure the plugin properties
oMyCS.setIOCFramework( "coldspring" );
oMyCS.setIOCDefinitionFile( "config/coldspring.xml.cfm" );

//Startup the engines, configure itself.
oMyCS.configure();

//Cache it for usage in my app, indefinitely
getColdBoxOCM().set( "MyCSFactory" ,oMyCS, 0);

//You are ready to roll baby!
getColdBoxOCM().get( "MyCSFactory" ).getBean( "myService" );

</cfscript>
```

That was fun! So how do I do the basic stuff then, I just want to use it. Well, for that, we have the following example. Of course, there are several ways to go about it, one is just using the ioc plugin calls everywhere, but if you plan ahead, you can just make the plugin or the service part of the event handler as a composition. You can do this in the **init** method of the event handler, where you can just setup the plugin or service as a property of the handler. You can do this like so:

```
<cffunction name="init" access="public" returntype="any" output="false" >
  <cfargument name="controller" type="any" required="true" >
  <cfset super.init(arguments.controller) >
  <!-- Any constructor code here -->
  <cfscript>
    super.init(arguments.controller);

    //Setup the plugin as a property
    variables.ioc = getPlugin( "ioc");
    //or if you want the actual service
    variables.oUserService = getPlugin( "ioc").getBean( "userService" );

    //return the handler instance
    return this;
  </cfscript>
</cffunction>
```

Ok, so I deviated once more from the basic sample and probably just confused you even more. However, here is the basic example from within an event handler method call I show the several ways to accomplish a task via the ioc plugin.:

```
<cffunction name="doValidation" output="false" hint="do user validation" returntype="void" access="public" >
  <cfargument name="Event" type="coldbox.system.beans.requestContext" >

  <cfset var rc = event.getCollection() >

  <!-- Get a security service from the plugin -->
  <cfset var securityService = getPlugin( "ioc").getBean( "securityService" )>

  <!-- Get and execute at once -->
  <cfset getPlugin( "ioc").getBean( "securityService" ).validateUser( rc.username, rc.password) >

  <!-- Get the IoC factory and call with it -->
  <cfset var coldspring = getPlugin( "ioc").getIoCFactory() >
  <cfset coldspring.getBean( "securityService" ).clearSession() >

  <!-- Call the ioc plugin property, if setup in the init -->
  <cfset variables.ioc.getBean( "securityService" ).validateUser(rc.username, rc.password) >
  <!-- OR with a getter -->
  <cfset getioc().getBean( "securityService" ).validateUser(rc.username, rc.password) >

  <!-- Call the security Service property, if setup in the init -->
  <cfset variables.securityService.validateUser(rc.username, rc.password) >
  <!-- OR with a getter -->
  <cfset getSecurityService().validateUser(rc.username, rc.password) >

</cffunction>
```

Autowiring Your Handlers

ColdBox also provides you a more advanced way to autowire your handlers with dependencies from the IoC plugin. You can see all of the techniques by reading the [Autowire guide](#). This guide will show you how to easily bring in dependencies in to your handlers by the use of metadata and the autowire interceptor.

The ColdBoxFactory.cfc : Leveraging Coldspring Custom Factories

This feature in Coldspring lets you define your own factories to create beans for you via the definitions file. Below is what the Coldspring documentation states. Please see [Coldspring Docs](#)

factory-method and **factory-bean** are two attributes of the `<bean/>` tag in Coldspring bean definitions that allow you to use legacy factories that you may have written, or otherwise any component who exposes a method that creates other components (or theoretically any value).

So now that we understand what this does, lets review: We define a ColdBox factory in our Coldspring declaration to create a reference to the application's ColdBox controller, a configuration bean with all of the application's configuration data (coldbox.xml), and to create configured core and custom plugins. WOW! Yep, you can create already configured ColdBox plugins like the logger, ioc, i18N, and many more, for you to use in your model or AOP designs.

So to use this functionality, you must first define the factory you want to use, in this case the ColdBox Factory

```
<bean id="coldboxFactory" class="coldbox.system.extras.ColdboxFactory" />
```

Then, you can use your factory to define other Coldspring beans, by using the factory-bean attribute to point at the ColdBox factory and the factory-method attribute to indicate which method Coldspring should call on the ColdBox factory to obtain an instance from it. The following are some methods available in the ColdBox Factory:

ColdBoxFactory.cfc :

Methods	Returns	Description
<code>getConfigBean()</code>	ConfigBean	Returns an application's config bean
<code>getColdbox()</code>	controller	Get the coldbox controller reference

<code>getPlugin(string plugin, [boolean customPlugin = false], [boolean newInstance = false])</code>	Plugin	Returns a custom or core plugin
<code>getColdBoxOCM()</code>	ColdBoxCache	Returns the instance of the Application's ColdBox cache Manager
<code>getDatasource(string alias)</code>	datasourceBean	A datasource bean object as defined in your config
<code>getMailSettings()</code>	mailsettingsBean	Get a mail settings bean as defined in your config

The first example is to define a configuration bean and then inject it in a user service object.

```
<bean id="ConfigBean" factory-bean="ColdboxFactory" factory-method="getConfigBean" />
<bean id="userService" class="myapp.model.userService" >
  <property name="ConfigBean" >
    <ref bean="ConfigBean" />
  </property>
</bean>
```

The second example is to inject the same user Service with the config bean and a reference to the coldbox controller:

```
<bean id="ConfigBean" factory-bean="ColdboxFactory" factory-method="getConfigBean" />
<bean id="userService" class="myapp.model.userService" >
  <property name="ConfigBean" >
    <ref bean="ConfigBean" />
  </property>
  <property name="coldbox" >
    <bean id="coldbox" factory-bean="ColdBoxFactory" factory-method="getColdbox" />
  </property>
</bean>
```

I will use the same factory to create a logger plugin for me and inject it into my same service:

```
<bean id="ConfigBean" factory-bean="ColdboxFactory" factory-method="getConfigBean" />
<bean id="loggerPlugin" factory-bean="ColdboxFactory" factory-method="getPlugin" >
  <constructor-arg name="plugin" >
    <value>logger</value>
  </constructor-arg>
</bean>
<bean id="userService" class="myapp.model.userService" singleton="false" >
  <property name="ConfigBean" >
    <ref bean="ConfigBean" />
  </property>
  <property name="coldbox" >
    <bean id="coldbox" factory-bean="ColdBoxFactory" factory-method="getColdbox" />
  </property>
  <property name="logger" >
    <ref bean="loggerPlugin" />
  </property>
</bean>
```

The user service will need the appropriate setter methods for this injections, and for that look at the [Coldspring documentation](#). The user Service will then be configured with a configBean, a ColdBox controller reference and a reference to a logger plugin. You can now use ColdBox to enhance your model with logging features, configuration data, etc.

The last example is to create a User Gateway object configured with a datasource, a User service with the gateway, and a logger plugin:

```
<bean id="ConfigBean" factory-bean="ColdboxFactory" factory-method="getConfigBean" />
<bean id="dsnBean" factory-bean="ColdboxFactory" factory-method="getDatasource" >
  <constructor-arg name="alias" >
    <value>mysnalias</value>
  </constructor-arg>
</bean>
<bean id="loggerPlugin" factory-bean="ColdboxFactory" factory-method="getPlugin" >
  <constructor-arg name="plugin" >
    <value>logger</value>
  </constructor-arg>
</bean>
<bean id="userGateway" class="myapp.model.userGateway" >
  <property name="dsnBean" >
    <ref bean="dsnBean" />
  </property>
</bean>
```

```
<bean id="userService" class="myapp.model.userService" >
  <property name="ConfigBean" >
    <ref bean="ConfigBean" />
  </property>
  <property name="logger" >
    <ref bean="loggerPlugin" />
  </property>
  <property name="userGateway" >
    <ref bean="userGateway" />
  </property>
</bean>
```

Here are some more examples:

Datasource Bean

```
<bean id="ColdboxFactory" class="coldbox.system.extras.ColdboxFactory" autowire="no" />
<bean id="datasourceBean" factory-bean="ColdBoxFactory" factory-method="getDatasource" >
  <constructor-arg name="alias" >
    <value>coldboxreader </value>
  </constructor-arg>
</bean>
```

Interceptor Service

```
<bean id="ColdboxFactory" class="coldbox.system.extras.ColdboxFactory" autowire="no" />
<bean id="ColdBoxController" factory-bean="ColdBoxFactory" factory-method="getColdBox" />
<bean id="InterceptorService" factory-bean="ColdBoxController" factory-method="getInterceptorService" />
```

Request Service

```
<bean id="ColdboxFactory" class="coldbox.system.extras.ColdboxFactory" autowire="no" />
<bean id="ColdBoxController" factory-bean="ColdBoxFactory" factory-method="getColdBox" />
<bean id="RequestService" factory-bean="ColdBoxController" factory-method="getRequestService" />
```

Debugger Service

```
<bean id="ColdboxFactory" class="coldbox.system.extras.ColdboxFactory" autowire="no" />
<bean id="ColdBoxController" factory-bean="ColdBoxFactory" factory-method="getColdBox" />
<bean id="DebuggerService" factory-bean="ColdBoxController" factory-method="getDebuggerService" />
```

ColdBox Applications are IoC Framework Agnostic

What does this mean? Well, that your ColdBox applications are totally decoupled from a dependency injection and IoC framework. Your code will work with any supported framework, because you only talk to the `ioC` plugin. This makes your applications extremely portable and flexible. If for some reason, you need to change framework, you can by just switching your configuration parameters, and that IS IT!

Advanced Topic: IOObjectCaching

Object caching from the factory is an advanced technique and it should be used with extreme caution and understanding of dependencies, persistence and logic.

Important: If a bean is injected in multiple locations, for example a `loggingService` and it is being cached by ColdBox, then the IoC factory will create it again. This is a caveat when using the ColdBox cache for IOC object caching. If your singleton never expires, then use `singleton=true` and don't use ColdBox. But if you want to use the ColdBox IoC cache, then do set them as non-singletons (transients). In Summary, **The ColdBox IoC caching, only caches what ColdSpring produces.**

Setting Caching Parameters in your Beans

If you do not do the following parameter definitions in your `cfc`'s, the coldbox cache manager will not cache them. This is absolutely useful since you can make beans expire rather quickly and make other beans never expire. Again, I reiterate that this technique is for singletons not for transient objects and you must be careful when this bean is used as a dependency on other objects. If you don't understand why, THEN DON'T USE IT.

Note: The cache parameter is optional, beans are NOT cached by default.

The Caching Parameters

Since ColdBox is built with a solid object cache foundation, your beans can also be cached if needed. You will do this by adding two meta data attributes to the `cfcomponent` tag. Caching of beans simulates persistence, so remember this if you are planning services that can maintain their own persistence. This is a true flexible and awesome feature. Persistence controlled by the framework for you (Please see important notes on dependencies above).

Attribute	Type	Description
cache	boolean	A true or false will let the framework know whether to cache this bean object or not.
cachetimeout	numeric	The timeout of the object in minutes. This is an optional attribute and if it is not used, the framework defaults to the default object timeout in the cache settings. You can place a 0 in order to tell the framework to cache the object for the entire application timeout controlled by coldfusion.

Caching Declaration Examples

```
//Cache the userService indefinitely by setting a 0
<cfcomponent name="UserService" output="false" cache="true" cachetimeout="0">

//Cache my mail service for 20 minutes
<cfcomponent name="MailService" output="false" cache="true" cachetimeout="20">

//Cache my bugservice for 5 minutes
<cfcomponent name="bugService" output="false" cache="true" cachetimeout="5">

//Do not cache my cheapService
<cfcomponent name="cheapService" output="false" cache="false">
```

Guide Conclusion

Well, there you go! You have been presented with how to declare ColdSpring usage, how to use the variables that are embedded into the factory and how to use the ColdBox cache manager to do your singleton's and persistence. You are all set to start delving into more complex and object oriented designs that will be leveraged by the ease of use of ColdBox and ColdSpring combined. They are a powerful combination!

📁 Categories:

- [level-intermediate](#)