

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

# LightWire Integration Guide

## Contents

- [LightWire Integration Guide](#)
- [Introduction](#)
- [What is LightWire](#)
- [Where to start?](#)
- [Configuration Overrides](#)
- [Where is the LightWire component?](#)
- [Creating the LightWire Configuration Bean](#)
- [How Do I get the Beans in my Handlers?](#)
- [Autowiring Your Handlers](#)
- [ColdBox Applications are IoC Framework Agnostic](#)
- [Advanced Topic: IOObjectCaching](#)
  - [Setting Caching Parameters in your Beans](#)
  - [The Caching Parameters](#)
  - [Caching Declaration Examples](#)
- [Guide Conclusion](#)

## Introduction

ColdBox provides you with a great set of tools and configurations to integrate LightWire into your applications. If you are not familiar with LightWire then you better get up to date. It is a lightweight Inversion of Control framework for ColdFusion written by [Peter Bell](#). It is also included in the ColdBox core to provide you with IoC capabilities out of the box.

## What is LightWire

LightWire's core focus is to make the configuration and dependencies of your CFCs easier to manage. LightWire uses the **inversion-of-control** pattern to **wire** your CFCs together. LightWire provides three methods for injecting dependencies and properties into your CFCs: Constructor Injection, Setter Injections and Mixin Injection.

To read more about LightWire go here:

- [LightWire on RIAForge](#)
- [LightWire blog](#)
- [LightWire discussion list](#)
- [Peter Bell's blog](#)

## 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
<b>IOCFramework</b>	The named framework to use: coldspring or lightwire.
<b>IOCFrameworkReload</b>	Boolean variable that tells the framework to reload the factory on each request. Great for development. Defaults to false.
<b>IOCDefinitionFile</b>	This is the instantiation path of your lightwire configuration bean or a valid coldspring xml file
<b>IOObjectCaching</b>	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 [wiki:cbCachingGuide ColdBox Cache Manager]. Please read the advanced topic on caching below.

The IOCDefinitionFile is a different value than if you would be using ColdSpring, you enter the instantiation class path to your LightWire config bean. For example, if your config bean is located in `/config/LightWire.cfc`, make the following adjustment:

```
<Setting name= 'IOCDefinitionFile' value= 'config.LightWire' />
```

Since ColdBox versions 2.6 and higher you can let go of that configuration object if you already have a valid coldspring xml file with all your bean declarations. The ioc plugin will detect that you want to use a coldspring xml file and it will create a base config object for you and make it parse this configuration file for you. So interchanging between ColdSpring and lightwire in coldbox becomes even simpler.

```
<Setting name= "IOCFramework" value= "lightwire" />
<Setting name= "IOCDefinitionFile" value= "config/services.xml.cfm" />
```

This feature really makes life easier for the developer who is used to working with bean configuration files and doesn't want to create a configuration object. ColdBox will create one for you and configure LightWire for you.

## 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= "LightWireBeanFactory" value= "frameworks.lightWire" />
</YourSettings>
```

## Where is the LightWire component?

The LightWire component used by ColdBox is located at `config.system.extras.lightwire.LightWire`. This is a setting that exists in the framework's `settings.xml` file. The ioc plugin uses this setting in order to know which IoC framework to use and how to invoke it. If you would like to use another location for the LightWire factory, you will have to change the **LightWireBeanFactory** setting in the `settings.xml` or use the overrides discussed above.

## Creating the LightWire Configuration Bean

In your application's `config` folder create a component and name it `Lightwire.cfc` [or `beanConfig.cfc`]. This is your LightWire configuration bean and it needs to extend `coldbox.system.extras.lightwire.BaseConfigObject`. The only difference between this base config object and the one shipped by LightWire is that it has an extra method so you can interact with ColdBox. This is transparent to the developer. A basic LightWire config bean should look like the following:

```
<cfcomponent name= "Lightwire" extends= "coldbox.system.extras.lightwire.BaseConfigObject" hint= "A LightWire configuration bean." >
    <cffunction name= "init" output= "false" returntype= "any" hint= "I initialize the config bean." >
        <cfscript>
            super.init();
            setLazyLoad( 'false' );

            // Create objects and inject till your heart's content

            // Use getController() to access ColdBox specific methods

            // Product Service
            addSingleton( "coldbox.samples.applications.lightwiresample.com.model.Product.ProductService" );
            addConstructorDependency( "ProductService" , "ProdDAO" );
            addConstructorProperty( "ProductService" , "MyTitle" , "My Title Goes Here" );
            addConstructorProperty( "ProductService" , "MyTitle2" , "My Other Title Goes Here" );
            addSetterProperty( "ProductService" , "MySetterTitle" , "My Setter Title Goes Here" );
            addMixinProperty( "ProductService" , "MyMixinTitle" , "My Mixin Title Goes Here" );
            addMixinProperty( "ProductService" , "AnotherMixinProperty" , getController().getSetting( 'Property_title' ) );
            addMixinDependency( "ProductService" , "CategoryService" );

            return THIS;
        </cfscript>
    </cffunction>
```

```
</cfcomponent>
```

**Note:** The LightWire config bean must extend the `baseConfigObject` located in `/coldbox/system/extras/lightwire/BaseConfigObject.cfc`. If your config bean does not correctly extend the `BaseConfigBean`, LightWire will throw an error.

Inside of the configuration bean, you can call the ColdBox controller by using the `getController()` method. With this capability you can ask ColdBox for properties, settings, datasource beans, plugins, and much more so you can inject them to your definitions. EXTRA COOL!!

## 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. See [Live API](#) for all the plugin's methods. Below are some of them:

Method	Description
<code>getBean(string beanName)</code>	To get a bean via the declared framework
<code>getExpandedIOCDefinitionFile()</code>	Get the expanded location of the configuration file
<code>getIOCDefinitionFile()</code>	Get the configuration file as defined in the config.
<code>getIoCFactory()</code>	Returns the IoC Factory instance in use.
<code>getIOCFramework()</code>	Returns the name of the IoC Factory in use.
<code>reloadDefinitionFile()</code>	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 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( "lightwire" );
oMyCS.setIOCDefinitionFile( "config.unitTestingLightWire" );

//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 lightwire = getPlugin( "ioc" ).getIoCFactory() >
  <cfset lightwire.getTransient( "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.

## 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 LightWire 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.

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 LightWire 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 LightWire combined. They are a powerful combination!!

🔗 Categories:

- [level-advanced](#)