

[← Back to Dashboard](#)

WireBox: The Enterprise Dependency Injection Framework

Contents

Covers up to version 1.3.1

Introduction

WireBox is an enterprise ColdFusion dependency injection and [AOP framework](#). This project has been part of ColdBox since its early version 2.0 releases but it is also a standalone library that can be used in ANY ColdFusion application or framework. WireBox's inspiration has been based on the idea of rapid workflows when building object oriented ColdFusion applications, programmatic configurations and simplicity. With that motivation we introduced dependency injection by annotations and conventions, which has been the core foundation of WireBox. We have definitely been influenced by great DI projects like [Google Guice](#), [Grails Framework Spring](#) and [ColdSpring](#) so we thank them for their contributions and inspiration.

Overview

Dependency injection is the art of making work come home to you.
- Dhanji R. Prasanna



WireBox alleviates the need for custom object factories or manual object creation in your ColdFusion applications. It provides a standardized approach to object construction and assembling that will make your code easier to adapt to changes, easier to test, [mock](#) and extend.

As software developers we are always challenged with maintenance and one ever occurring annoyance, *change*. Therefore, the more sustainable and maintainable our software, the more we can concentrate on real problems and make our lives more productive. WireBox leverages an array of metadata annotations to make your object assembling, storage and creation easy as pie! We have leveraged the power of event driven architecture via object listeners or interceptors so you can extend not only WireBox but the way objects are analyzed, created, wired and much more. To the extent that our [AOP](#) capabilities are all driven by our AOP listener which decouples itself from WireBox code and makes it extremely flexible.

We have also seen the value of a central location for object configuration and behavior so we created our very own WireBox Programmatic Mapping DSL ([Domain Specific Language](#)) that you can use to define object construction, relationships, AOP, etc in pure ColdFusion (No XML!). We welcome you to stick around and read our documentation so you can see the true value of WireBox in your web applications.

Fact: WireBox has been running and powering mission critical ColdFusion applications since 2009 and now you can too.

Useful Resources

- <http://code.google.com/coldfusion/>
- <http://www.mapping.com/prasanna>
- http://en.wikipedia.org/wiki/Aspect-oriented_programming
- http://en.wikipedia.org/wiki/Dependency_injection
- http://en.wikipedia.org/wiki/Inversion_of_control
- <http://martinfowler.com/articles/injection.html>
- <http://www.thesevencities.com/news/131158-A-hoosierseminar-discusses-Dependency-Injection>
- <http://www.developer.com/net/netarticle.php/363651/>
- <http://code.google.com/coldfusion/>

Training, Presentations, More

- [ColdBox Connection: Introducing WireBox](#)

Features At A Glance

Here are a simple listing of features WireBox brings to the table:

- Annotation driven dependency injection
- 0 configuration mode or a programmatic binder configuration approach via ColdFusion (No XML!)
- Creation and Wiring of or by:
 - ColdFusion Components
 - Java Classes
 - RSS Feeds
 - Webservice objects
 - Constant values
 - DSL string building
 - Factory Methods
- Multiple Injection Styles: Property, Setter, Method, Constructor
- Automatic Package/Directory object scanning and registration
- Multiple object life cycle persistence scopes:
 - No Scope (Transients)
 - Singletons
 - Request Scoped
 - Session Scoped
 - Application Scoped
 - Server Scoped
 - CacheBox Scoped
- Integrated logging via [LogBox](#), never try to figure out what in the world the DI engine is doing
- Parent Factories
- Factory Method Object Creations
- Object life cycle events via WireBox Listeners/Interceptors
- Customizable injection DSL
- WireBox object providers to avoid scope-widening issues on time/volatile persisted objects
- [Aspect Oriented Programming](#)
- [Standalone ORM Entity Injection](#)

Installing WireBox

WireBox can be downloaded as a separate framework or it is included with the latest ColdBox Platform release. The main difference between both versions is the instantiation and usage namespace, the rest is the same.

Standalone

wirebox.system.ioc

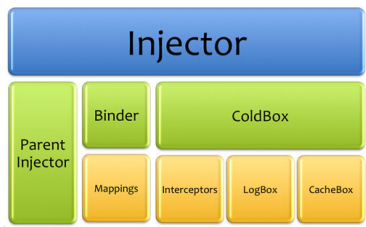


ColdBox

coldbox.system.ioc

WireBox: The Enterprise Dependency Injection Framework

- [Introduction](#)
- [Overview](#)
 - [Useful Resources](#)
 - [Training, Presentations, More](#)
 - [Features At A Glance](#)
- [Installing WireBox](#)
 - [Standalone](#)
 - [Download](#)
 - [System Requirements](#)
 - [Installation](#)
- [Dependency Injection Explained](#)
 - [Advantages of a DI Framework](#)
 - [Getting Insej Wire-It!](#)
 - [Instance Creations](#)
 - [Binder Introduction](#)
 - [Scoping](#)
 - [Scope Annotations](#)
 - [Scope Configuration Binder](#)
 - [Internal Scopes](#)
 - [Error Impl](#)
 - [How WireBox Resolves Dependencies](#)
 - [Instance Creation](#)
 - [Dependency Resolution](#)
- [The WireBox Injector](#)
 - [Injector Constructor Arguments](#)
 - [Injection Idioms](#)
 - [Common Methods](#)
- [Configuring WireBox](#)
 - [Binder Configuration Properties](#)
 - [CacheBox Embedding Binder](#)
 - [Types & Scopes](#)
 - [JMS TVPDS](#)
 - [JMS SCOPES](#)
 - [Implicit Configuration Settings](#)
 - [LogBox Config](#)
 - [CacheBox](#)
 - [Scope Registration](#)
 - [Custom DSL](#)
 - [Custom Scopes](#)
 - [Scan Locations](#)
 - [AOP Restrictions](#)
 - [Parent Injectors](#)
 - [Listeners](#)
- [Mapping DSL](#)
 - [WireBox Configuration](#)
 - [Mapping Injectors](#)
 - [Mapping Destinations](#)
 - [Persistence DSL](#)
 - [Dependencies DSL](#)
 - [Mapping Extra Attributes](#)
- [Component Annotations](#)
 - [Scope Persistence Annotations](#)
 - [CacheBox Integration](#)
- [Injection DSL](#)
 - [Property Annotation](#)
 - [Constructor Argument Annotation](#)
 - [Setter Method Annotation](#)
 - [ID/Model/Entity Namespace](#)
 - [Provider Namespace](#)
 - [WireBox Namespace](#)
 - [CacheBox Namespace](#)
 - [EntityService Namespace](#)
 - [LogBox Namespace](#)
 - [ColdBox Namespace](#)
- [ORM Entity Injection](#)
- [Virtual Inheritance](#)
- [Runtime Missing](#)
- [WireBox Event Model](#)
 - [WireBox Events](#)
- [WireBox Listeners](#)
 - [ColdBox Mode Listener](#)
 - [Standalone Mode Listener](#)
- [Providers](#)
 - [Custom Providers](#)
 - [Virtual Provider Injection DSL](#)
 - [Virtual Provider Mapping DSL](#)
 - [Virtual Provider Lookup Methods](#)
 - [Provider onMissingMethod Proxy](#)
 - [Scope Widening Injection](#)
- [Object Persistence & Thread Safety](#)
- [Custom DSL Builders](#)
 - [The DSL Builder Interface](#)
 - [Registering a custom DSL builder](#)
- [Custom Scopes](#)
 - [The Scope Interface](#)
 - [Scoping Process](#)
 - [Registering A Custom Scope](#)
- [WireBox Injector Interface](#)
- [WireBox Object Population](#)
 - [populateFromXML](#)
 - [Returns](#)
 - [Arguments](#)
 - [populateFromDir](#)
 - [Returns](#)
 - [Arguments](#)
 - [populateFromStruct](#)
 - [Returns](#)
 - [Arguments](#)
 - [populateFromDirsWithPrefix](#)
 - [Returns](#)
 - [Arguments](#)
 - [populateFromJSON](#)
 - [Returns](#)
 - [Arguments](#)
- [Mapping DSL Examples](#)



Download

- [Download WireBox](#)
- [Download API Docs](#)

System Requirements

- ColdFusion 8 and above
- Railo 3.1 and above

Installation

If you are using WireBox within a ColdBox application context, then WireBox is part of the platform. Just install [ColdBox normally](#). If you are using WireBox standalone, just drop WireBox in your application root or create a mapping called `wirebox` that points to the installation folder. If you can run the following snippet, then WireBox is installed correctly:

```
wirebox <createObject component="wirebox.system.ioc.Injector" init();
```

Note: Please remember that if you use the standalone version the namespace is `wirebox.system.ioc` and if you use the ColdBox application version it is `coldbox.system.ioc`. From this point on, we will use the standalone namespace for simplicity.

Dependency Injection Explained

We have released one of our chapters from our [CBOX202: Dependency Injection](#) course that deals with getting started with Dependency Injection, the problem, the benefits and the solutions. We encourage you to [download it](#), print it, share it, digest it and learn it: <http://otus-public-s3.amazonaws.com/chx202-unit1-3.pdf>

Advantages of a DI Framework

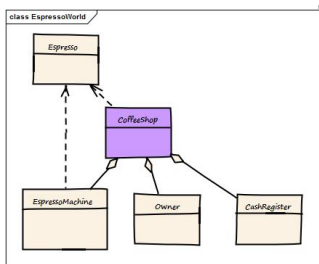
Compared to manual DI, using WireBox can lead to the following advantages:

- You will write less boilerplate code.
- By giving WireBox DI responsibilities, you will stop creating objects manually or using custom object factories.
- You can leverage object persistence scopes for performance and scalability. Even create time persisted objects.
- You will not have any object creation or wiring code in your application, but have it abstracted via WireBox. Which will lead to more cohesive code that is not plagued with boilerplate code or factory code.
- Objects will become more testable and easier to mock, which in turn can accelerate your development by using a TDD ([Test Driven Development](#)) approach.
- Once WireBox leverages your objects you can take advantage of AOP or other event life cycle processes to really get funky with OO.

Getting Jiggy Wit It!

A primer to WireBox usage

Dependency injection and instance construction with WireBox is easy. In its most simplest form we can just leverage annotations and be off to [Jancing Big Willy style!](#) You can use our global injection annotation `inject` on cfproperties, setter methods or constructor arguments. This annotation tells WireBox to inject something in place of the property, argument or method; basically it is your code shouting "Hey buddy, I need your help here". What it injects depends on the contents of this annotation that leverages our injection DSL (Domain Specific Language). The simplest form of the DSL is to just tell WireBox what mapping to bring in for injection. Please note that I say mapping and not object directly, because WireBox works on the concept of an object mapping. This mapping in all reality can be a CFC, a java object, an RSS feed, a webservice, a constant value or pretty much anything you like. If you don't like annotations because you feel they are too intrusive to your taste, don't worry, we also have a programmatic configuration binder you can use to define all your objects and their dependencies. We will discuss object mappings and our configuration binders later on, so let's look at how cool this is by checking out our Coffee Shop sample class. The `CoffeeShop` class below will use our three types of injections to showcase how WireBox works, please note that most likely we would build this class by picking one or the other, which in itself brings in pros and cons for each approach.



```

component name=CoffeeShop singleton {
  // define a property and tell WireBox to inject it
  property name=espressoMachine injectId=espressoMachine*

  function init(any owner inject) {
    variables.owner = arguments.owner;
    return this;
  }

  function openShop() onDIComplete {
    espressoMachine.turnOn();
    owner.nap();
  }

  function setCashRegister(cashRegister) inject {
    variables.cashRegister = arguments.cashRegister;
  }

  function makeEspresso() {
    return espressoMachine.makeEspresso();
  }
}

```

So let's break this class down. First, you can see a `singleton` annotation on the component declaration. This tells WireBox that this class should only be created once and then cached in its internal singleton scope of the injector. In other words, this is called object life scopes. You can refer to the persistence scopes annotations later on in the guide to learn all about how to scope your classes.

Second, we built our coffee shop class with three external dependencies: 1 by cfproperty, 1 by constructor argument and 1 by setter injection. Again, you can see later on in this guide the difference between all these injection styles and choose what you prefer. In this example, we just showcase the different injection styles. Also, as you can see from the source code the three types of injection uses the `inject` annotation but with different content:

1. `property name=espressoMachine injectId=espressoMachine*`
2. `function init(any owner inject)`
3. `function setCashRegister(cashRegister) inject`

If you just mark a property, argument or method with the `inject` annotation, WireBox will assume it is a mapping and the ID should be either the property name, the argument name or the method name. However, if you want to specify the `id` in the DSL string, just use the simple `id=[mapping]` notation. That's it! Isn't that cool, you just mark out your dependencies and WireBox will build and inject them for you!

Thirdly, this class has the following method:

```

function openShop() onDIComplete {
  espressoMachine.turnOn();
  owner.nap();
}
// or
<cffunction name=openShop returnType=void output=false onDIComplete
</cffunction>

```

The method has a cool little annotation called `onDIComplete` that tells WireBox that after all DI dependencies have been injected, then execute the method. That is so cool, WireBox can even open the coffee shop for me so I can get my espresso fix. Not only that

but you can have multiple *DIComplete* methods declared and *WireBox* will call them for you (in discovered order). These are called object post processors that are discovered by annotations or can be configured via our configuration binder and we will learn about them later on. *WireBox* also fires a series of object life cycle events throughout an object's life span in which you can build listeners to and actually perform some cool stuff on them. So now that we got all excited about opening the coffee shop let's get into something even more interesting, unit testing and mocking.

Another important aspect leveraging DI concepts when building our components is that we can immediately write tests for them and leverage mocking to test for actual behaviors. This is a great advantage as it allows you to rapidly test to confirm your component is working without worrying about building or assembling objects in your tests. You have eliminated all kinds of crazy creation and assembler code and just concentrated yourself on the problem at hand. You are now focused to code the greatest piece of software you have ever imagined, thanks to *WireBox*!

So let's build our unit test (Please note we use our base *ColdBox* testing classes for ease of use and *MockBox* integration):

```
component extends@dbox.system.testing.BaseModelTest*

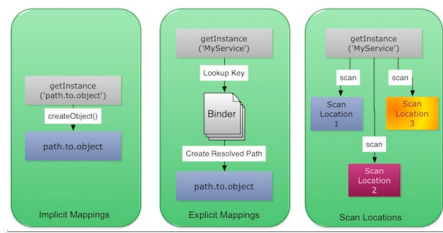
function setUp() {
    // mock some owner
    mockOwner = getMockBox().createEmptyMock();
    // create our coffee shop class with mocking capabilities
    shop = getMockBox().createEmptyMock(CoffeeShop);
    // mock the espresso machine
    mockMachine = getMockBox().createEmptyMock(EspressoMachine);
    // inject to the shop's variables scope to simulate DI
    shop.$setProperty(EspressoMachine, mockMachine);
}

function testMakeEspresso() {
    // mock methods
    mockMachine.$fakeEspresso('createStub');
    // test
    shop.makeEspresso();
    assertTrue('mockMachine.$fakeEspresso');
}

function testOpenShop() {
    // mocks
    mockMachine.$fakeOpen();
    mockOwner.$fake();
    // test
    shop.openShop();
    assertTrue('mockMachine.$fakeOpen');
    assertTrue('mockOwner.$fake');
}
}
```

Now we can run our tests and verify that our coffee shop is operational and producing sweet sweet espresso!

Instance Creations



We have now coded our classes and unit tests with some cool annotations in record time, so what do we do next? Well, *WireBox* works on the idea of three ways to discover and create your classes:

Approach	Motivation	Pros	Cons
Implicit Mappings	To replace createObject() calls	Very natural as you just request an object by its instantiation path. Very fast prototyping.	Refactoring is very hard as code is plagued with instantiation paths everywhere. Not DRY.
Explicit Mappings	To replace createObject() calls with named keys	DRY, you can create multiple named mappings that point to the same blueprint of a class. Create multiple iterations of the same class. Very nice decoupling.	Not as fast to prototype as we need to define our mappings before hand in our configuration binder.
Scan Locations	CFC discovery by conventions	A partial instantiation path(s) or folder(s) are mapped so you can retrieve by shorthand names. Very quick to prototype also without using full instantiation paths. Override of implementations can be easily done by discovery.	Harder concept to digest, not as straightforward as implicit and explicit locations.

So let's do examples for each where our classes we just built are placed in a directory called *model* of the root directory.

Implicit Creation

```
injector createObject(component,*wirebox.system.ioc.Inject@t@n@t());
espresso = injector.getInstance(model.CoffeeShop).makeEspresso();
```

Explicit Binder Configuration

```
map("CoolShop":to(model.CoffeeShop*);
```

Explicit Creation

```
injector createObject(component,*wirebox.system.ioc.Inject@t@n@t());
espresso = injector.getInstance(model.CoffeeShop).makeEspresso();
```

Scan Locations Binder Configuration

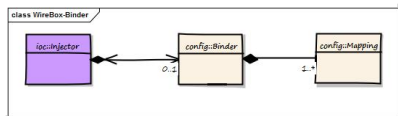
```
wirebox.scanLocations=model;
```

Scan Locations Creation

```
injector createObject(component,*wirebox.system.ioc.Inject@t@n@t());
espresso = injector.getInstance(model.CoffeeShop).makeEspresso();
```

So our recommendation is to always try to create configuration binders as best practice, but your requirements might dictate something else.

Binder Introduction



In all reality we could be building our objects and its dependencies, *object graph*, without any configuration just plain location and implicit conventions. This is great but not very flexible for refactoring, so let's do the best practice of defining a mapping or an alias to a real object. We do this by creating a *WireBox* configuration binder *wirebox.system.ioc.config.Binder*, which is a simple CFC that defines the way *WireBox* behaves and defines object mappings. This binder is then used to initialize *WireBox* so it has knowledge of these mappings and our settings.

Scoping

We touched briefly on singleton and no scope objects in this section, so let's delve a little into what scoping is. *WireBox's* default behavior is to create a new instance of an object each time you request it via creation or injection (Transient/Prototype objects), this is the **NO SCOPE** scope. Scopes allow you to customize the object's life span and duration. The singleton scope allows for the creation of only one instance of an object that will live for the entire life span of the injector. *WireBox* ships with several different life span scopes but you can also create your own custom scopes (please see the custom scopes section). You can also tell *WireBox* in what scope to place the instance into by annotations or via the configuration binder. We have an entire section dedicated to discovering all the *WireBox* annotations, but let's get a sneak peek at them and also how to do it via our mapping DSL.

Scope Annotations

- You can tag a *cfc:component* tag or component declaration with a *scope={named scope}* annotation that tells *WireBox* what scope to use
- You can have nothing on the *cfc:component* tag or component declaration which denotes the **NO SCOPE**
- You can tag a *cfc:component* tag or component declaration with a *singleton* annotation

Scope Configuration Binder

```
component extends@wirebox.system.ioc.config.B@n@der*

function configure() {
    // map with shorthand or full scope notation
    mapPath(model.CoffeeShop,Singleton());
}
```

```
mapPath(model.CoffeeShop into this.SCOPE.SINGLETON);
// map some long espresso into request scope
map{longEspresso*
    .toModel.Espresso*
    .into this.SCOPE.REQUEST};
// cache some tea
map{GreenTea*
    .toModel.Tea*
    .inCacheBox(timeout=20,provides@refresh);
// cache some google news that refresh themselves every 40 minutes or after 20 minutes of inactivity
map{latestNews*
    .inCacheBox(timeout=40,lastAccessTimeout=20,provides@
    .toRSShttp://news.google.com/news?output=rss*
}
}
```

Note: In the configuration binder section you will see where the *this.SCOPE* enumeration class comes from.

Internal Scopes

Here are the internal scopes that ship with WireBox:

Scope	Description
NOSCOPE	A prototype object that gets created every time it is requested.
PROTOTYPE	A prototype object that gets created every time it is requested.
SINGLETON	Only one instance of the object exists
SESSION	The object will exist in the session scope
APPLICATION	The object will exist in the application scope
REQUEST	The object will exist in the request scope
SERVER	The object will exist in the server scope
CACHEBOX	A object will be time persisted in any CacheBox cache provider

This is cool! We can now have full control of how objects are persisted via the WireBox injector, we are not constricted to one type of persistence anymore.

Important: If you use a persistence scope that expires after time like session, request, cachebox, etc, you will experience a side effect called scope widening injection. WireBox offers a solution to this side effect via WireBox Providers, which we will cover in detail.

Eager Init

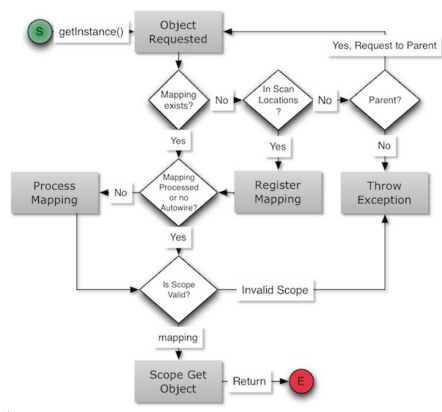
Another aspect of our objects is when are they created? Good question! By default all objects are created **ONLY** when they are requested, in other words they are lazy created. But what if you are spoiled and you want your stuff NOW NOW NOW! Well, you can, you ride little bra! Just tell WireBox that you want your objects to be eagerly created. How? Via the mapping DSL and our cool *asEagerInit()* function.

```
component.extend@wirebox.system.loc.config.Binder*
function configure(){
    // map with shorthand or full scope notation
    mapPath(model.CoffeeShop*
        .asSingleton()
        .asEagerInit();
}
}
```

How WireBox Resolves Dependencies

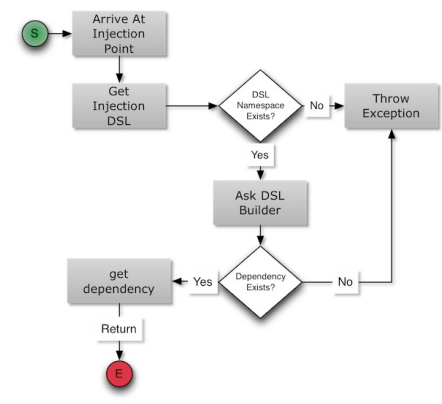
Most of the time we believe our DI engines should be black boxes, but we try to think otherwise. We encourage developers to know what is going on so they can debug easily and not hit their foreheads against their keyboards. Believe me, I have done so before. That is why WireBox is tightly integrated with [LogBox](#) to provide incredible debugging information to ANY appender you desire so you can know what is going on. Another aspect of knowing what the DI engine does is how dependencies are resolved. Here is a typical flow of injection:

Instance Creation



- Object is requested by name and the Injector tries to check if the mapping exists for that name. If no mapping is found then it tries to locate the object by using the internal scan locations to try to find it. If it cannot find it and there is a parent injector defined, then the request is funneled to the parent injector and we start our process again. If no parent injector is declared and no localization, then we throw a not located exception.
- If the object was found via the scan locations, then we register a new mapping according to its location and discover all the metadata out of the object in preparation for construction and DI
- We now have a guaranteed mapping so we retrieve it and we verify if the mapping's metadata has been processed or not. If the mapping is marked with no autowiring then we skip to the next step. If not, we process the mapping's metadata and prepare it for DI
- We verify that the scope define for the mapping exists, else we throw an invalid scope exception
- We ask the scope to produce the mapping object for us. The scope is in charge of persistence, locking, etc.
- The scope builds the instance by asking the injector to build a new instance with the correct constructor and constructor arguments and stores it in its scope once the injector builds it. The builder decides what type of construction is needed for the mapping as it can be a CFC, java object, webservice, RSS feed, factory method call, etc. Each constructor argument is processed for dependency resolution.
- The scope then sends the instance for DI wiring and process back to the injector
- The injector returns the instance

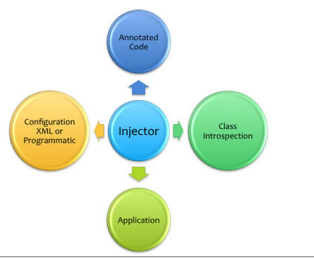
Dependency Resolution



- Arrive at the desired injection point and get the injection DSL. If the DSL is empty, then it defaults to the *id/model* namespace. For this injection DSL Namespace we try to find a valid DSL builder for it. If none is found an exception is thrown. If we have a match, then the DSL builder is called with the DSL string to retrieve.
- The DSL builder then tries to parse and process the DSL string for object retrieval. If the DSL is a WireBox mapping then we try to retrieve the instance by name (Refer back to Instance Creation).
- If the builder could not produce an instance, it is logged and DI is skipped on it.

Important: Circular dependencies are supported in all injection styles within WireBox. With one caveat, if you choose constructor arguments with circular dependencies, you must use object providers.

The WireBox Injector



WireBox bases itself on the idea of creating object injectors (*wirebox.system.ioc.Injector*) that in turn will produce and wire all your objects. You can create as many injector instances as you like in your applications, each with configurable differences or be linked hierarchically by setting each other as parent injectors. Each injector can be configured with a configuration binder or none at all. If you are a purely annotations based kind of developer and don't mind requesting pathed components by convention, then you can use the no-configuration approach and not even have a single configuration binder file, all using autowiring and discovery of conventions. However, if you would like to alter the behavior of the injector and also create object mappings, you will need a configuration binder. The next section explains the way to create this configuration binder, below is how to startup or bootstrap the injector in different manners:

No Configuration Binder:

```
myObject createObjectComponent*coldbox.system.ioc.Injector::init().getInstances(myObject?)
```

With a Configuration Binder:

```
myObject createObjectComponent*coldbox.system.ioc.Injector::init(myBinderPath::getInstances[fooObject?]
```

The WireBox injector class is the pivotal class that orchestrates DI, instance events and so much more. We really encourage you to study its [API Docs](#) to learn more about its construction and usage methods.

Injector Constructor Arguments

The injector can be constructed with three optional arguments:

Argument	Type	Required	Default	Description
binder	instance or instantiation path	false	<i>wirebox.system.ioc.config.DefaultBinder</i>	The binder instance or instantiation path to be used to configure this WireBox injector with
properties	struct	false	structnew()	A structure of name value pairs usually used for configuration data that will be passed to the binder for usage in configuration.
coldbox	<i>coldbox.system.web.Controller</i>	false	null	A reference to the ColdBox application context you will be linking the Injector to.

If you are using WireBox within a ColdBox application, you don't even need to do any of this, we do it for you by using some configuration data in your ColdBox configuration file or conventions. Please see the ColdBox application section for more information.

Injection Idioms

Now that we have constructed our injector let's discuss a little about injection idioms or styles WireBox offers before we go all cowboy and start configuring and using this puppy. Below is a nice chart that showcases the WireBox injection styles, but we really encourage you to review our dependency injection section to learn the different approaches to DI, their values, and when to use them.

Injection Location	Injection Order	Motivation	Comments
Constructor	First	Mandatory dependencies for object creation	Each constructor argument receives a <i>inject</i> annotation with its required injection DSL. Be careful when dealing with object circular dependencies as they will fail via constructor injection due to its chicken and the egg nature.
CFProperty	Second	Great documentable approach to variable mixins to reduce getter/setter verbosity	Leverages the greatest aspect of ColdFusion, dynamic language, to mixin variables at runtime by using the <i>cfproperty</i> annotations. Great for documentation and visualizing object dependencies and safe for circular dependencies. Cons is that you can not use the dependencies in an object's constructor method.
Setter Methods	Third	Legacy classes	The <i>inject</i> annotation MUST exist on the setter method if the object is not mapped. Mapping must be done if you do not have access to the source or you do not want to touch the source.

These are the three injection styles that WireBox supports and which style you choose depends on your requirements and also your personal taste. The setter method approach is linked to the way Spring and ColdSpring approach it which is the traditional JavaBean style of *setXXX* where XXX is the name of the mapping or object to pass into the setter method for injection.

Note: Whichever injection style you use with WireBox, the target's visibility does not matter. This means that you can create private or package methods and WireBox will still inject them for you. This is absolutely great when you are an encapsulation freak and you do not want to expose public setter methods.

Common Methods

The following chart shows you the most common methods when dealing with the WireBox Injector. This doesn't mean there are no other methods on the Injector that are of value, so please check out the [CFC Docs](#) for more in-depth knowledge.

Method Signature	Comments
autowire(target,[mapping],[targetID],[annotationCheck])	A method you can use to send objects to get autowired by convention or mapping lookups
clearSingletons()	A utility method that clears all the singletons from the singleton persistence scope. Great to do in development.
containsInstance(name)	Checks if an instance can be created by this Injector or not
getBinder()	Get the configuration binder for this injector
getInstance([name],[dsl],[initArguments])	The main method that asks the injector for an object instance by name or by autowire DSL string.
getObjectPopulator()	Retrieve the ColdBox object populator that can populate objects from JSON, XML, structures and much more.
getParent()	Get a reference to the parent injector (if any)
getScope(name)	Get a reference to a registered persistence scope
setParent(injector)	Set a parent injector into the target injector to create hierarchies

Configuring WireBox

You can configure WireBox in two approaches:

1. Create a simple configuration CFC that has a *configure(binder)* method that accepts a WireBox configuration binder object

```
component {
    function configure(required binder) {
    }
}
```

2. Create a configuration CFC that extends the WireBox configuration object: *coldbox.system.ioc.config.Binder* and has a *configure()* method.

```
component extends=coldbox.system.ioc.config.Binder*
function configure() {
}
}
```

The latter approach will be less verbose when talking to the mapping DSL the Binder object exposes. However, both are fully functional and matter of preference.

From the *configure()* method you will be able to interact with the Binder methods or creating implicit DSL structures in order to configure WireBox for operation and also to create object mappings.

Please also note that the Binder itself has a reference to the current Injector it belongs to (*getInjector()*).

Binder Configuration Properties

Whether you use WireBox standalone or within a ColdBox context a Binder gets a structure of configuration properties so it can use them whenever you are configuring it or declaring mappings. If you are in standalone mode, the Injector can be constructed with a *properties* structure that will be passed to the binder for usage. If you are in a ColdBox application the ColdBox application configuration structure is passed for you. You can then use these properties with the following methods:

- **getProperty(name,[default])**: Get a specific property
- **getProperties()**: Get all the properties structure
- **propertyExists(name)**: Check if a property exists
- **setProperty(name,value)**: Dynamically add properties to the structure

ColdBox Enhanced Binder

If you are using your configuration binder within a ColdBox application you will have some extra goodies in the Binder that come in very handy:

- **getColdBox()**: Retrieve the instance of the running ColdBox application
- **getAppMapping()**: Get the current *AppMapping* setting for the running ColdBox application

```
// map the model folder
mapDirectory( getAppMapping('model');
```

Types & Scopes

Each configuration binder has two public properties accessible in the *this* scope:

1. **this.TYPES**: A reference to *coldbox.system.ioc.Types* used to declare what type of object you are registering for construction or wiring
2. **this.SCOPES**: A reference to *coldbox.system.ioc.Scopes* used to declare in what life cycle scope the object will be stored under

These two classes contain static public members in the *this* scope that facilitate the declaration of persistence scopes and construction types for object mappings. Below are the valid enumerations for these two classes:

this.TYPES.

- **CFC**: Construction of a CFC
- **JAVA**: Construction of a Java class
- **WEBSERVICE**: Construction of a webservice object
- **RSS**: Construction of an RSS feed
- **DSL**: Construction by DSL string
- **CONSTANT**: A constant value
- **FACTORY**: Construction by factory method

this.SCOPEs.

- **NOSCOPE**: Transient objects
- **PROTOTYPE**: Transient objects
- **SINGLETON**: Objects constructed only once and stored in the injector
- **SESSION**: ColdFusion session scoped based objects
- **APPLICATION**: ColdFusion application scope based objects
- **REQUEST**: ColdFusion request scope based objects
- **SERVER**: ColdFusion server scope based objects
- **CACHEBOX**: CacheBox scoped objects

Implicit Configuration Settings

In this `configure()` method you can create a structure called `wirebox` in the `variables` scope that will hold the configuration data for WireBox. The following are the keys you can create in this structure:

Key	Type	Required	Default	Description
<code>logBoxConfig</code>	instantiation or xml path	false	<code>coldbox.system.ioc.config.LogBox</code>	The LogBox configuration to use when logging. If you are within a ColdBox application context, this value is ignored.
<code>cacheBox</code>	struct	false	<code>{enabled=false}</code>	A structure that defines the tight integration between WireBox and CacheBox (explained below). If you are within a ColdBox application context, this value is ignored.
<code>scopeRegistration</code>	struct	false	<code>{enabled=true,scope="application",key="wirebox"}</code>	A structure that tells WireBox to register itself into ANY ColdFusion scope once instantiated. By default, each injector gets loaded into application scope with a key of <code>wirebox</code> , which we encourage you to modify.
<code>customDSL</code>	struct	false	<code>{}</code>	A structure where you will register your own DSL Namespace implementations. Please refer to our Custom DSL section.
<code>customScopes</code>	struct	false	<code>{}</code>	A structure where you will register your own object scope implementations. Please refer to our Custom Scopes section.
<code>scanLocations</code>	array	false	<code>[]</code>	An array of instantiation locations WireBox will use (in order) when searching for objects when they are requested by convention.
<code>stopRecurSIONs</code>	array	false	<code>[]</code>	An array of class paths that WireBox will detect when inspecting objects for dependencies and STOP the inspection when doing inheritance recursion.
<code>parentInjector</code>	object	false	<code>---</code>	The actual instance to a parent injector you would like to configure this injector with.
<code>listeners</code>	array of structs	false	<code>---</code>	A array of listener definitions that will be registered with this injector and listen to object life cycle events.

Please note that it is completely optional to use the implicit structure configuration. You can use the programmatic methods instead. Each configuration key has the same method in the binder for programmatic configuration.

logBoxConfig

A string containing the path of the [LogBox](#) configuration file. If you are using WireBox within a ColdBox application this setting is ignored.

```
wirebox.logBoxConfig=coldbox.system.ioc.config.LogBox*
```

cachebox

If you are using WireBox within a ColdBox application this setting is ignored. The following are the keys for this configuration structure:

Key	Type	Required	Default	Description
<code>enabled</code>	boolean	false	false	Enables CacheBox integration
<code>configFile</code>	config path	false	<code>coldbox.system.ioc.config.CacheBox</code>	The CacheBox configuration file to use when creating CacheBox
<code>cacheFactory</code>	object	false	<code>---</code>	A reference to an already instantiated CacheBox factory to use with this Injector
<code>classNamespace</code>	class path	false	<code>coldbox.system.cache</code>	The default namespace location of CacheBox. If using the standalone version of CacheBox you most likely will change this to <code>cachebox.system.cache</code> , else ignore this setting.

```
wirebox.cacheBox = {
  enabled false
  configFile "coldbox.system.ioc.config.CacheBox" optional configuration file to use for loading CacheBox
  cacheFactory "#", //A reference to an already instantiated CacheBox CacheFactory
  classNamespace "*" //A class path namespace to use to create CacheBox: Default=coldbox.system.cache or wirebox.system.cache
};
```

scopeRegistration

This structure tells WireBox how to teach itself into any ColdFusion scope when initialized.

Key	Type	Required	Default	Description
<code>enabled</code>	boolean	false	true	Enables scope registration
<code>scope</code>	CF Scope	true	<code>application</code>	The ColdFusion scope
<code>key</code>	string	true	<code>wirebox</code>	The key to use in the ColdFusion scope when registering

```
wirebox.scopeRegistration = {
  enabled true
  scope "application" / server, cluster, session, application
  key "wirebox"
};
```

customDSL

Please refer to the Custom DSL section to find out more about custom DSLs, the following are just the way you declare them:

Key	Type	Required	Default	Description
<code>{DSL.Namespace}</code>	string	true	<code>---</code>	The value of the DSL Namespace is the instantiation path of the DSL Namespace builder that implements <code>coldbox.system.ioc.DSL.IDSLBuilder</code>

```
wirebox.customDSL = {
  cool "my.path.CoolDSLBuilder"
  funkyBox "my.funky.DSLBuilder"
};
```

CustomScopes

Please refer to the Custom scopes section to find out more about custom scopes, the following are just the way you declare them:

Key	Type	Required	Default	Description
<code>{ScopeName}</code>	string	true	<code>---</code>	The value of the instantiation path of the custom scope that implements <code>coldbox.system.ioc.scopes.IScope</code> . The name of the scope will be used when registered the <code>scope</code> annotation.

```
wirebox.customScopes = {
  CoolSingletons "my.path.SingletonScope"
  FunkyTransaction "my.funky.Transaction"
};
```

scanLocations

The instantiation paths that this Injector will have registered to do object locations in order. So if you request an object called `Service` and no mapping has been configured for it, then WireBox will search all these scan locations for a `Service.cfc` in the specified order. The last lookup is the no namespace lookup which basically represents a `createObject("component", "Service")` call. If you are using WireBox within a ColdBox application, ColdBox will register the `models` convention folder for you and also whenever a ColdBox module is activated, that module's `model` convention folder will be added here too.

Important: Please note that order of declaration is the same as order of lookup, so it really matters. Also note that this setting only makes sense if you do not like to create mappings for objects and you just want WireBox to discover them for you.

```
wirebox.scanLocations="model"transfer.com#brg.majand?
```

stopRecurSIONs

This is an array of class paths that WireBox will use to stop recursion on any object graph that has inheritance when looking for dependencies. For example, let's say your object inherits from `transfer.com.TransferDecorator`, but you don't want WireBox to go past that inheritance class when looking for DI data, then you would add `transfer.com.TransferDecorator` to this setting.

```
wirebox.stopRecurSIONs="transfer.com.TransferDecorator"coldbox.system.EventHandler*
```

parentInjector

This setting is actually a reference to another parent injector you would like this injector to set as its parent injector. Now say this sentence 10 times without hiccuping.

```
wirebox.parentInjector = application.coolInjector;
// or
wirebox.parentInjector=createObject("component","coldbox.system.ioc.Injector").bind#*
```

listeners

This section only shows you how to register WireBox listeners, so please refer to the object life cycle events section for more information. This setting is an array of listener structure definitions that WireBox's event manager will use when broadcasting object life cycle events. Each interceptor structure definition has the following keys:

Key	Type	Required	Default	Description
<code>class</code>	class path	true	<code>---</code>	The instantiation class path of the listener

name	string	false	Name of the CFC	The unique name of this listener when registered in our event manager. We recommend setting one up as best practice, else the name of the CFC file will be used instead. This setting is great for registering the same class with different configurations.
properties	struct	false	{ }	A structure of configuration data for this listener

Important: Please note that order of declaration is the same as order of execution, so it really matters, just like ColdBox [Interceptors](#). Please note that if you are using WireBox within a ColdBox application, you can also register listeners as interceptors in your ColdBox configuration file.

```
wirebox.listeners = [
  {class:my.AOPTracker},
  {class:annotationTransactionProperties={target}},
  {class:Timer; name:CoolTime}
];
```

Mapping DSL

WireBox can also be configured by using the programmatic Mapping DSL exposed in the configuration binder instead of the implicit data structures DSL we just saw. Our recommendation is to use this mapping DSL as it makes your configuration become alive and more human readable than creating a bunch of arrays and structures. All mappings DSL methods return back an instance of the binder so you can concatenate methods to create readable execution chains:

```
map("Lul")
  .to("model.Awesomenebs")
  .asEagerInit()
  .asSingleton();
```

WireBox Configuration

The configuration binder has the same methods as the implicit structures that can be used to configure WireBox for operation:

Method Signature	Description
cacheBox (configFile,[cacheFactory],[enabled],[classNamespace])	The method used to configure the injector's CacheBox integration. Ignored in an application context
listener (class,[properties],[name])	The method used to register a new listener within the injector's event manager.
logBoxConfig (config)	The method used to tell the injector which LogBox configuration file to use for logging operations. Ignored in an application context
mapDSL (namespace,path)	The method used to register a new DSL annotation namespace with a DSL Builder object.
mapScope (annotation,path)	The method used to register a new custom scope in this injector.
parentInjector (injector)	Register a CFC reference to be the parent injector for the configuring injector
removeScanLocations (locations)	A method used to remove one or a list (array) of scan locations from the configuration binder
reset ()	Reset the entire configuration binder to factory defaults
scanLocations (locations)	A method used to add one or a list (array) of scan locations to the configuration binder. If a path already exists it will not be appended again.
scopeRegistration (enabled,scope,key)	This method is used to tell the Injector if it should auto-register itself in any ColdFusion scope automatically.
stopRecursion (classes)	A method used to register one or a list (array) of class paths the injector will look out for when discovering DI metadata. If these classes are found in the inheritance chain of an object, the injector will not process that inherited chain.

```
logBoxConfig(config,LogBox")
  .scanLocations( getAppMapping(1)includes.model)
  .stopRecursion(model.BaseService,model.BaseModel)
  .mapScope(ortus,"model.scopes.ortus")
```

Mapping Initiators

Ok, now that we know how to configure WireBox, let's get into the fun stuff of object mapping. How do we do this? By using our DSL mapping initiators that tell WireBox how to start the object registration process. You will then concatenate the initiators with some DSL destination methods, DI data, etc to tell WireBox all the information it might need to construct, wire and persist the object. Here are the DSL initiators:

Method Signature	Description
map (alias)	The method that starts the mapping process. You pass in a mapping name or a list of names to start registering.
mapPath (path)	Map a CFC instantiation path. This method internally delivers a two-fold punch of doing <code>map('CFCFileName').to(path)</code> . This is a quick way to map a CFC instantiation path that uses the name of the CFC as the mapping name.
mapDirectory (packagePath,[include],[exclude])	A cool method that tells WireBox to automatically register ALL the CFCs found recursively in that instantiation package path. All CFCs will be registered using their CFC names as the mapping names and WireBox will inspect all the CFCs immediately for DI metadata. The include and exclude arguments can be used for inclusions/exclusions lists via regex.
with (alias)	This method is a utility method that retrieves the <i>alias</i> mapping so you can start concatenating methods for that specific mapping. Basically putting it into a workable context.

Important: From the methods we have seen above only the `map()` and `with()` methods require a DSL destination.

Mapping Destinations

The mapping destinations tell WireBox what type of object you are mapping to. You will usually use these methods by concatenating `map()` or `with()` initiator calls:

Method Signature	Description
to (path)	Maps a name to a CFC instantiation path
toDSL (dsl)	Maps a name to DSL builder string. Construction is done by using this DSL string (Look at Injection DSL)
toFactoryMethod (factory,method)	Maps a name to another mapping (factory) and its method call. If you would like to pass in parameters to this factory method call you will use the <code>methodArg()</code> DSL method concatenated to this method call.
toJava (path)	Maps a name to a Java class that can be instantiated via <code>createObject("java")</code>
toProvider (provider)	Maps a name to another mapping (provider) that must implement the WireBox Provider interface (<code>coldbox.system.ioc.IProvider</code>)
toRSS (path)	Maps a name to an atom or RSS URL. WireBox will then use the <code>cffeed</code> tag to construct this RSS feed. It builds out into a structure with two keys: <ul style="list-style-type: none"> metadata: The metadata of the feed items: The items in the feed
toValue (value)	Maps a name to a constant value, which can be ANYTHING.
toWebservice (path)	Maps a name to a webservice WSDL URL. WireBox will create the webservice via <code>createObject("webservice")</code> for you.

Here are some examples:

```
// CFC
map("FunkyObject":to("myapp.model.service.FunkyService")
mapPath("myapp.model.service.FunkyService")
mapDirectory("myapp.model")
// Java
map("buffer").toJava("java.lang.StringBuffer")
// RSS Feed
map("googleNews":toRSS("http://news.google.com/news?outpu=rss")
// Webservice
map("myWS").toWebservice("http://myapp.com/app.cfc?wml")
// Provider
map("Espresso").toProvider("funkyEspressoProvider")
// DSL
map("logger").toDSL("logbox:root")

// factory methods
map("ColdboxFactory":to("coldbox.system.extras.ColdboxFactory")
map("ColdboxController":toFactoryMethod("factoryColdboxFactory:methodGetColdBox")
map("BeanInjector")
  .toFactoryMethod("factoryColdboxFactory:methodGetPlugin")
  .methodArg("name=ugin,value=BeanFactory")

// Mixin a new method in my object that dispenses users
mapPath("UserService")
  .providerMethod("getUser":"User");
```

Important: Please note that WireBox can create different types of objects for DI. However, only CFCs will be inspected for autowiring automatically unless you specifically tell WireBox that a certain mapping should not be autowired. In this case you will use the dependencies DSL to define all DI relationships.

Persistence DSL

The next step in our mapping DSL excursion is to learn about how WireBox will persist these object mappings into WireBox scopes. By default (as we have seen), all object mappings are transient objects and they belong to a scope type called **NOSCOPE**. However, we need to specifically tell WireBox into what scope the declared mapped objects should be placed on in order for us to leverage caching, the singleton pattern, etc. This is accomplished by leveraging our persistence component annotations or the following methods if you prefer a non-annotation approach:

Note: Please note that all WireBox configuration binders have two public properties:

- `this.TYPES` - Enum class (`coldbox.system.ioc.Types`)
- `this.SCOPE` - Enum class (`coldbox.system.ioc.Scopes`)

These classes have on themselves several public properties that are a cool shorthand way to link to construction types or persistence scopes.

Method Signature	Description
asSingleton ()	Maps an object to the WireBox internal <i>Singleton</i> scope
into (scope)	Maps an object to a valid WireBox internal scope or any custom registered scopes by using the registered scope name. Valid internal WireBox scopes are: <ul style="list-style-type: none"> • NOSCOPE • PROTOTYPE • SINGLETON • SESSION • APPLICATION • REQUEST • SERVER • CACHEBOX

inCacheBox([key='mappingName'],[timeout],[lastAccessTimeout],[provider='default']) Maps an object to the integrated [CacheBox](#) instance.

asEagerInit() Maps an object to be created immediately once the Injector is created. By default all object mappings are lazy loaded in construction.

So just remember that these persistence DSL methods are not mandatory. If you are an annotations kinda developer, then you can easily add these persistence annotations to your classes.

```
// cfc
map("FunkyObject"
  .to("myapp.model.service.FunkyService"
    .asSingleton());
mapPath("myapp.model.service.FunkyService"
  .into("this.SCOPE.REQUEST");
// Java as NO SCOPE
map("buffer").toJava("java.lang.StringBuffer"
// RSS feed
map("googleNews"
  .toRSS("http://news.google.com/news?output=rss"
  .inCacheBox(timeout=60,lastAccessTimeout=15);
// Webservice
map("myWS"
  .toWebService("http://myapp.com/app.cfc?wsl"
  .into("this.SCOPE.APPLICATION");
```

Important : Please note that by leveraging scopes that can expire such as *cachebox.request.session.applications.etc* you must take into account the way they are injected into other objects. They can experience a DI side effect called scope widening injection that can link an object reference that expires into another object reference that does not expire (like singleton). This causes nasty side effects and issues, so please refer to the [WireBox Providers](#) section to find out how you can avoid this nasty pitfall by using WireBox providers.

Dependencies DSL

The dependencies DSL methods are mostly used to define dependencies and also to activate advanced features on target objects, such as runtime mixins, virtual inheritance, etc.

Please note that you can concatenate more than one of these methods calls to dictate multiple constructor arguments, setter methods, cf properties, and more.

Method Signature	Description
constructor (constructor)	Tells WireBox which constructor to call on the mapped object. By default if an object has an <i>init</i> () method, that will be used as the constructor
noInit ()	Tells WireBox that this mapped object will skip the constructor call for it. By default WireBox always calls object constructors
threadSafe ()	Tells WireBox that the mapped object should be constructed and then wired with a strict concurrency lock for property injections, setter injections and onDICComplete(). Please be aware that if you use this mode of construction, circular dependencies are not allowed. The default is that property and setter injections and onDICComplete() are outside of the construction locks.
notThreadSafe ()	Tells WireBox to construct objects by locking only the constructor and constructor argument dependencies to allow for circular dependencies. This is the default construction mode of all persisted objects: singleton, session, server, application and cachebox scope.
noAutowire ()	Tells WireBox that this mapped object has its dependencies described programmatically instead of using metadata inspection to discover them.
initArg ([name],[ref],[dsl],[value],[javaCast])	Used to define a constructor argument for the mapped object. <ul style="list-style-type: none"> name : The name of the constructor argument. Not used for Java or Webservice construction ref : The mapping reference id this constructor is mapped to. E.G. ref="MyFunkyEspresso" dsl : The construction dsl that will be used to construct this constructor argument value : The constant value you can use instead of a dsl or ref for this constructor argument javaCast : If using a java object, you can cast the value of this constructor argument
initWith ()	You can pass as many arguments (named or positional) to this method to simulate the <i>init</i> () call of the mapped object. WireBox will then use that argument collection to initialize the mapped object.
methodArg ([name],[ref],[dsl],[value],[javaCast])	Used to define a factory method argument for the mapped object when using a factory method construction. <ul style="list-style-type: none"> name : The name of the method argument. Not used for Java or Webservice construction ref : The mapping reference id this method argument is mapped to. E.G. ref="MyFunkyEspresso" dsl : The construction dsl that will be used to construct this method argument value : The constant value you can use instead of a dsl or ref for this method argument javaCast : If using a java object, you can cast the value of this method argument
property ([name],[ref],[dsl],[value],[javaCast],[scope])	Used to define a property mixin that will occur at runtime. <ul style="list-style-type: none"> name : The name of the property value to inject. Not used for Java or Webservice construction ref : The mapping reference id this property is mapped to. E.G. ref="MyFunkyEspresso" dsl : The construction dsl that will be used to construct this property argument value : The constant value you can use instead of a dsl or ref for this property argument javaCast : If using a java object, you can cast the value of this property argument scope : The scope inside the CFC this property will be injected too. The default scope is the variables scope.
setter ([name],[ref],[dsl],[value],[javaCast])	Used to define all the setter dependencies for a mapped object that follows the JavaBean spec: <i>setXXX</i> where XXX is the name of the mapped object. <ul style="list-style-type: none"> name : The name of the setter. Not used for Java or Webservice construction ref : The mapping reference id this setter is mapped to. E.G. ref="MyFunkyEspresso" dsl : The construction dsl that will be used to construct this setter dependency value : The constant value you can use instead of a dsl or ref for this setter dependency javaCast : If using a java object, you can cast the value of this setter dependency
mixins (udfIncludeList)	A UDF template, a list of templates or an array of templates that WireBox should use to mix-in into the target object. It will take all the methods defined in those UDF templates and mixed them into the target object at runtime.
providerMethod (method,mapping)	Will inject a new method or override a method on the target object with a new method that provides objects of the mapping you specify.
virtualInheritance (Mapping)	Create a runtime virtual inheritance from a target object into a target mapping. This approach blends the CFCs together at runtime via mixins and WireBox Funkyness!
extraAttributes (struct)	Allows the ability to store extra metadata about a mapping into WireBox that can later be retrieved via AOP invocations or WireBox events.

Mapping Extra Attributes

You can store a-la-carte attributes in a specific mapping so it can be retrieved at a later time by either an AOP aspect or Events. This is a great way to store custom metadata about an object so it can be read later for some meaningful purpose. Let's say you want to tag a mapping with a custom type that is not so easily determined from the object instance itself. You don't want to do all kinds of introspection in order to know what object you received in an aspect or an event.

```
map("MyPlugin"
  .to("plugins.CustomPlugin"
  .extraAttributes({
    pluginPath = pluginLocation,
    custom = arguments.custom,
    module = arguments.module,
    isPlugin true
  }));
```

This mapping declares that an object has some extra attributes that will be stored in the mapping, such as the location, if it is a custom plugin, if it belongs to a module and a marker that determines if it is a plugin or not. This is then incredibly useful when you have an attached listener to WireBox:

```
function afterInstanceAutowire(event, interceptData){
  var attrs = interceptData.mapping.getExtraAttributes();
  var iData = {};
  // listen to plugins only
  if( structKeyExists(attrs,"plugin*"){
    //Fill-up Intercepted MetaData
    iData.pluginPath = attrs.pluginPath;
    iData.custom = attrs.custom;
    iData.module = attrs.module;
    iData.oPlugin = interceptData.target;
  }
  //Fire My Own Custom Interception
  instance.interceptorService.processStorePluginCreate(iData);
}
```

As you can see from this sample, the extra attributes are incredibly essential, as the listener just sends the target object. It would take lots of introspection and metadata inspections in order for me to determine if the incoming object is my system's plugin or not. However, with the extra attributes, it is just a snap!

Component Annotations

The following are all the annotations that are discovered by WireBox on any **component** declaration that WireBox constructs:

Annotation	Type	Description
autowire	boolean	All objects are marked as <i>autowire=true</i> , so if you want to disable autowiring, you can add this annotation as false . You do NOT need to add this annotation if you want to autowire it, it is redundant if you do.
alias	string	A list of aliased names you can attach to a CFC instance apart from its Component name. This is great when using the <i>mapDirectory()</i> binder function.
eagerInit	none	All objects are lazy loaded unless they are marked with this annotation or marked as eager init in the binder configuration.
threadSafe	none or boolean	Determines the locking construction of the object for its wiring of dependencies. Please see our <i>Object Persistence & Thread Safety</i> Section.
scope	string	A valid WireBox scope or a custom registered scope. Remember that ALL components by default are placed in the NO SCOPE scope. This means they are considered transient objects.
singleton	none	Marks a component as a singleton object.
cachebox	string	Marks a component to be stored in CacheBox. The value of this annotation should be a valid registered CacheBox cache provider. The default cache provider is called <i>default</i>
cache	boolean	Marks a component to be cached in CacheBox in the <i>default</i> provider.
cacheTimeout	numeric	The timeout in minutes when the object is stored in the CacheBox provider
cacheLastAccessTimeout	numeric	The timeout in minutes when the object is stored in the CacheBox provider
mixins	list	A list of UDF templates to mixin into the object

Scope Persistence Annotations

The following annotations can be placed in the component declaration to tell the WireBox injector where to persist the constructed object. If no scope annotations are found on the component or mappings then the object is treated as **NO SCOPE** or a prototype/transient object; one that gets constructed every time.

- **singleton** - singleton object
- **scope="registered_scope"** : Registered Scope: session, request, singleton, custom, etc.

```
component singleton{}
component scope=singleton{}
component scope=request{}
component singleton threadsafe{}
```

CacheBox Integration

If you would like to use CacheBox for persistence for you objects you will need to mark your CFC with the following annotation(s)

- **cachebox="provider"** - The default provider is called 'default', so this annotation can be empty or a named cache provider
- **cache** - Cache into the default provider, shorthand annotation, no value needed

This annotation has two sub annotations that you can also leverage for granular control of your CacheBox integration:

- **cacheTimeout** - The timeout in minutes (optional)
- **cacheLastAccessTimeout** - The last access or idle timeout in minutes (optional)

```
// cache into the default provider
component cache{}
// cache into the default provider
component cachebox{}
// cache into the ehcache provider
component cachebox=ehcache{}
// cache into the ehcache provider with settings
component cachebox=ehcache cacheTimeout=60{}
// cache with settings
component cache cacheTimeout=60 cacheLastAccessTimeout=60{}
```

Important : When storing objects in volatile scopes like cache, session, request, etc. You must be careful of not injecting them directly into singletons or other volatile objects as you could have memory leaks via a side effect called **Scope Widening Injection**. We recommend combining them via WireBox Providers to avoid this side effect.

Injection DSL

The injection DSL is a domain specific language that denotes what to inject in the current placeholder: property, argument, or method via the **inject** annotation. This injection DSL not only can it be used via annotations but also via our mapping dsl whenever a **dsl** argument can be used. This DSL is constructed by joining words separated by a colon. The first part of this string is what we will denote as the injection DSL Namespace.

```
inject="namespace":extra:extra:extra"
```

Property Annotation

Every cfproperty can be annotated with our injection annotations:

- **@inject** : The injection DSL
- **@scope** : The visibility scope to inject the dependency into. By default it injects into **variables** scope

```
property name=myService inject=id:MyService"
property name=@VPES inject=id:CustomType@scope:this"
property name=@roles inject=id:RoleService:getRoles@scope:instance"
```

Constructor Argument Annotation

You can also annotated constructor arguments with the **inject** annotation.

```
<!-- Via tag based annotations -->
<cffunction name=init returntype=any output=false>
<cfargument name=myService inject=UserService"
<cfargument name=cache inject=cachebox:default"
</cffunction>

// Via script but alternative method as inline annotations are broken in ACF
/*+
 * Init
 * @myService inject UserService
 * @cache inject cachebox:default
 */
function init(required myService, required cache){
}
```

Important : In full script components, annotating inline arguments is broken in Adobe ColdFusion 9. You will have to annotate them via the alternative annotation syntax in ColdFusion 9 via the javadocs style comments.

Setter Method Annotation

You can also annotate setter methods with the **inject** annotation to provide injections

```
<!-- Via tag based annotations -->
<cffunction name=setService returnType=any output=false inject=UserService"
<cfargument name=service"
</cffunction>

function setService(required service) inject=service"
variables.service = arguments.service;
}
```

WireBox offers a wide gamut of annotation namespaces you can use in your applications and ColdBox applications. However, we took it a step further and allowed you to create your own custom DSL namespaces making your annotations come alive! So let's investigate the shipped namespaces:

ID-Model-Empty Namespace

The default namespace is not specifying one. This namespace is used to retrieve either named mappings or full component paths.

DSL	Description
empty	Same as saying <i>id</i> . Get a mapped instance with the same name as defined in the property, argument or setter method.
id	Get a mapped instance with the same name as defined in the property, argument or setter method.
id:{name}	Get a mapped instance by using the second part of the DSL as the mapping name.
id:{name}:{method}	Get the {name} instance object, call the {method} and inject the results
model	Get a mapped instance with the same name as defined in the property, argument or setter method.
model:{name}	Get a mapped instance by using the second part of the DSL as the mapping name.
model:{name}:{method}	Get the {name} instance object, call the {method} and inject the results

```
// Let's assume we have mapped a few objects called: UserService, SecurityService and RoleService
// Empty inject, use the property name, argument name or setter name
property name=myUserService inject;
// Using the name of the mapping as the value of the inject
property name=mySecurity inject=SecurityService"
// Using the full namespace
property name=myUserService inject=id:UserService"
property name=myUserService inject=model:UserService"
// Simple factory method
property name=@roles inject=id:RoleService:getRoles"
```

Provider Namespace

Inject object providers, please refer to our provider section in this guide.

DSL	Description
provider	Build an object provider that will return the mapping according to the property, method or argument name.
provider:{name}	Build an object provider that will return the {name} mapping.

```
property name=myTimedService inject=provider:TimedService"
```

WireBox Namespace

Talk and get objects from the current wirebox injector

DSL	Description
wirebox	Get a reference to the current injector
wirebox:parent	Get a reference to the parent injector (if any)
wirebox:eventManager	Get a reference to injector's event manager
wirebox:binder	Get a reference to the injector's binder

wirebox:populator Get a reference to a WireBox's Object Populator utility
wirebox:scope:{scope} Get a direct reference to an internal or custom scope object
wirebox:properties Get the entire properties structure the injector is initialized with. If running within a ColdBox context then it is the structure of application settings
wirebox:property:{name} Retrieve one key of the properties structure

```
property name=beanFactoryInject@wirebox?
property name=settingsInject@wirebox:properties?
property name=singletonCacheInject@wirebox:scope:singleton?
property name=populatorInject@wirebox:populator?
property name=binderInject@wirebox:binder?
```

CacheBox Namespace

This DSL namespace is only active if using CacheBox or a ColdBox application context.

DSL	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 objectKey

```
property name=cacheFactoryInject@cachebox?
property name=cacheInject@cachebox:default?
property name=dataInject@cachebox:default:myKey?
```

EntityService Namespace

Gives you the ability to easily inject base orm services or binded virtual entity services for you:

DSL	Description
entityService	Inject a BaseORMService object for usage as a generic service layer
entityService:{entity}	Inject a VirtualEntityService object for usage as a service layer based off the name of the entity passed in.

```
// Generic ORM service layer
property name=genericServiceInject@entityService?
// Virtual service layer based on the User entity
property name=userServiceInject@entityService:User?
```

LogBox Namespace

Interact with LogBox

DSL	Description
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
logbox:logger:{this}	Get a reference to a named logger using the current target object's path as the category name

```
property name=logboxInject@logbox?
property name=rootInject@logbox:root?
property name=logInject@logbox:logger:myapp?
property name=logInject@logbox:logger:{this}?
```

ColdBox Namespace

This namespace is a combination of namespaces that are only active when used within a ColdBox application:

DSL	Description
coldbox	Get the coldbox controller reference
coldbox:flash	Get a reference to the application's flash scope object
coldbox:setting:{setting}	Get the coldbox application <i>{setting}</i> setting and inject it
coldbox:setting:{setting}@{module}	Get the coldbox application <i>{setting}</i> from the <i>{module}</i> and inject it
coldbox:plugin:{plugin}	Get the <i>{plugin}</i> plugin and inject it
coldbox:myPlugin:{MyPlugin}	Get the <i>{MyPlugin}</i> custom plugin and inject it
coldbox:myPlugin:{MyPlugin}@{module}	Get the <i>{MyPlugin}</i> custom plugin from the <i>{module}</i> module and inject it
coldbox:datasource:{alias}	Get a new datasource bean according to <i>{alias}</i>
coldbox:configBean	Get a new config bean object and inject it
coldbox:mailsettingsbean	Get a new mail settings bean and inject it
coldbox:loaderService	Get a reference to the loader service
coldbox:requestService	Get a reference to the request service
coldbox:debuggerService	Get a reference to the debugger service
coldbox:pluginService	Get a reference to the plugin service
coldbox:handlerService	Get a reference to the handler service
coldbox:interceptorService	Get a reference to the interceptor service
coldbox:moduleService	Get a reference to the ColdBox Module Service
coldbox:interceptor:{name}	Get a reference of a named interceptor <i>{name}</i>
coldbox:cacheManager	get the cache manager
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
coldbox:moduleSettings:{module}	Inject the entire <i>{module}</i> settings structure
coldbox:moduleConfig:{module}	Inject the entire <i>{module}</i> configurations structure
ioc	Get the named ioc bean and inject it. Name comes from the cfproperty, setter or argument name
ioc:{beanName}	Get the ioc bean according to <i>{beanName}</i>
javaLoader:{class}	Create an object from the JavaLoader plugin and its set of loaded java libraries
webservice:{alias}	Get a webservice object using an <i>{alias}</i> that matches in your coldbox configuration file.

```
// some examples
property name=logboxInject@logbox?
property name=rootLoggerInject@logbox:root?
property name=userInject@logbox:logger:model.com.UserService?
property name=moduleServiceInject@coldbox:moduleService?
property name=producerInject@coldbox:interceptor:MessageProducer?
property name=moduleConfigInject@coldbox:fwConfigBean?
property name=producerInject@interceptor:MessageProducer?
property name=appPathInject@coldbox:fwSetting:ApplicationPath?

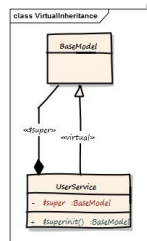
// JavaLoader goodness
property name=BinaryHeapInject@javaLoader:org.apache.commons.collections.BinaryHeap?
property name=mailInject@javaLoader:org.apache.commons.mail.SimpleEmail?
```

ORM Entity Injection

WireBox is fully capable to do ORM entity injection in your ColdFusion applications in two approaches:

1. [ColdBox Applications](#) - for use in ColdBox applications
2. [Standalone WireBox](#) - for use in ColdBox or any framework or any ColdFusion application.

Virtual Inheritance



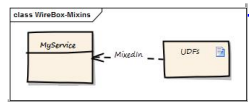
You can make two CFCs blend together simulating a virtual runtime inheritance with WireBox. WireBox will grab the target CFC and blend into it all of the virtual inheritance CFC's methods and properties. It will then also create a **\$Super** reference in the target and a **\$SuperIntr** reference. This is a great alternative to real inheritance and allow for runtime mixins to occur. You start off by mapping the base or source CFC and then mapping the target CFC and declaring a **virtualInheritance** to the base or source CFC.

```
// Declare base CFC
```

```
map('BaseModel':to('model.base.BaseModel'))
map('UserService':to('model.users.UserService').virtualInheritance('BaseModel'))
```

This will grab all methods and properties in the **AbstractModel** CFC and mix them into the **UserService**, then create a virtual **\$super** scope which will map to an instantiated instance of the **BaseModel** object.

Runtime Mixins()



You can use the **mixins()** binder method or **mixins** annotation to define that a mapping should be mixed in with one or more set of templates. It will then at runtime inject all the methods in those templates and mix them into the target object as **public** methods.

```
// map with mixins
map('MyService'
.to('model.UserService')
.mixins('/helpers/base'))

// map with mixins as list
map('MyService'
.to('model.UserService')
.mixins('/helpers/base', '/helpers/mbdel'))

// map with mixins as array
map('MyService'
.to('model.UserService')
.mixins(['/helpers/base', '/helpers/moddl']))

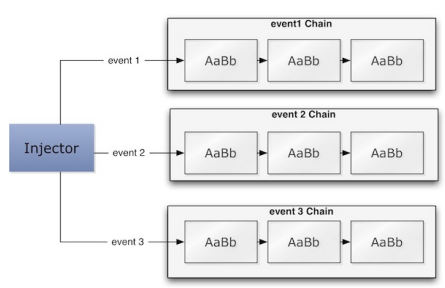
// Via annotation
component mixins=helpers/base
```

This will grab all the methods in the **base.cfm** and **model.cfm** templates and inject them into the target mapping as **public** methods. Awesome right?

The list of templates can include a **.cfm** or not

WireBox Event Model

WireBox also sports a very nice event model that can announce several object life cycle events. You can listen to these events and interact with WireBox at runtime very easily, whether you are in standalone mode or within a ColdBox application. Of course, if you are within a ColdBox application, you get the benefit of all the potential of **ColdBox Interceptors** and if you are in standalone mode, well, you just get the listener and that's it! Each event execution also comes with a structure of name-value pairs called **interceptData** that can contain objects, variables and all kinds of data that can be useful for listeners to use. This data is sent by the event caller and each event caller decides what this data sent is. Also, remember that WireBox also can be ran with a reference to **CacheBox**, which also offers lots of internal events that you can tap into. So let's start investigating first the object life cycle events.



WireBox Events

WireBox's offers a wide gamut of life cycle events that are announced at certain points in execution time. Below are the current events announced by the Injector *coldbox.system.ioc.Injector*.

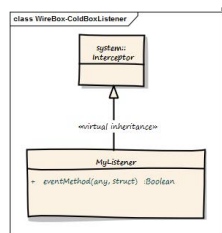
Event	Data	Description
afterInjectorConfiguration	<ul style="list-style-type: none"> ● injector: The calling injector reference 	Called right after the injector has been fully configured for operation.
beforeInstanceCreation	<ul style="list-style-type: none"> ● mapping: The mapping called to be created ● injector: The calling injector reference 	Called right before an object mapping is built via our internal object builders or custom scope builders.
afterInstanceInitialized	<ul style="list-style-type: none"> ● mapping: The mapping called to be created ● target: The object that just go constructed and initialized ● injector: The calling injector reference 	Called after an object mapping gets constructed and initialized. The mapping has NOT been placed on a scope yet and no DI/AOP has been performed yet.
afterInstanceCreation	<ul style="list-style-type: none"> ● mapping: The mapping called to be created ● target: The object that just go built, initialized and DI/AOP performed on it ● injector: The calling injector reference 	Called once the object has been fully created, initialized, stored, and DI/AOP performed on it. It is about to be returned to the caller via its <i>getInstance()</i> method.
beforeInstanceInspection	<ul style="list-style-type: none"> ● mapping: The mapping that is about to be processed. ● binder: The configuration binder processing the mapping ● injector: The calling injector reference 	Called whenever an object has been requested and its metadata has not been processed or discovered. In this interception point you can influence the metadata discovery.
afterInstanceInspection	<ul style="list-style-type: none"> ● mapping: The mapping that is about to be processed. ● binder: The configuration binder processing the mapping ● injector: The calling injector reference 	Called after an object mapping has been completely processed with its DI metadata discovery. This is your last chance to change or modify the DI data in the mapping before it is cached.
beforeInjectorShutdown	<ul style="list-style-type: none"> ● injector: The calling injector reference 	Called right before the Injector instance is shutdown.
afterInjectorShutdown	<ul style="list-style-type: none"> ● injector: The calling injector reference 	Called right after the Injector instance is shutdown.
beforeInstanceAutowire	<ul style="list-style-type: none"> ● injector: The calling injector reference ● mapping: The instance mapping ● target: The target object for wiring ● targetID: The unique target object ID used for wiring 	Called right after the instance has been created and initialized, but before DI wiring is done.
afterInstanceAutowire	<ul style="list-style-type: none"> ● injector: The calling injector reference ● mapping: The instance mapping ● target: The target object for wiring ● targetID: The unique target object ID used for wiring 	Called right after the instance has been created, initialized and DI has been completed on it.

Note: Please see our **CacheBox** documentation to see all of CacheBox's events.

WireBox Listeners

We have already seen in our previous section all the events that are announced by WireBox, but how do we listen? There are two ways to build WireBox listeners because there are two modes of operations, but the core is the same. Listeners are simple CFCs that must create methods that match the **same** name of the event they want to listen to. If you are running WireBox within a ColdBox application, listeners are **Interceptors** and you declare them and register them exactly the same way that you do with normal interceptors. Also, these methods can take up to two parameters depending on your mode of operation (standalone or coldbox). The one main difference between pure WireBox listeners and ColdBox interceptors are that *configure* method for the standalone WireBox is different. Please see samples.

ColdBox Mode Listener

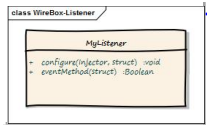


Argument	Type	Execution Mode	Description
event	<i>coldbox.system.web.context.RequestContext</i>	coldbox	The request context of the running request
interceptData	struct	standalone-coldbox	The data structure passed in the event

So let's say that we want to listen on the `beforeInjectorShutdown` and on the `afterInstanceCreation` event in our listener.

```
component{
function configure(){
function beforeInjectorShutdown(event, interceptData){
var injector = arguments.interceptData.injector;
// Do my stuff here:
// I can use a log object because ColdBox is cool and injects one for me already.
log.info@UDE, I am going down!!"
}
function afterInstanceCreation(event, interceptData){
var injector = arguments.interceptData.injector;
var target = arguments.interceptData.target;
var mapping = arguments.interceptData.mapping;
log.info@The object #mapping.getName()# has just been built, performing my awesome AOP ppxressing on it."
// process awesome AOP on this target
processAwesomeAOP( target );
}
}
```

Standalone Mode Listener



Argument	Type	Execution Mode	Description
interceptData	struct	standalone-coldbox	The data structure passed in the event

```
component{
function configure(injector.properties){
variables.injector = arguments.injector;
variables.properties = arguments.properties;
log = variables.injector.getLogBox().getLogger();
}
function beforeInjectorShutdown(interceptData){
// Do my stuff here:
// I can use a log object because ColdBox is cool and injects one for me already.
log.info@UDE, I am going down!!"
}
function afterInstanceCreation(interceptData){
var target = arguments.interceptData.target;
var mapping = arguments.interceptData.mapping;
log.info@The object #mapping.getName()# has just been built, performing my awesome AOP ppxressing on it."
// process awesome AOP on this target
processAwesomeAOP( target );
}
}
```

Please note the `configure()` method in the standalone listener. This is necessary when you are using Wirebox listeners outside of a ColdBox application. The `configure()` method receives two parameters:

- **injector**: An instance reference to the calling Injector where this listener will be registered with.
- **properties**: A structure of properties that passes through from the configuration file.

As you can see from the examples above, each Listener component can listen to multiple events. Now you might be asking yourself, in what order are these listeners executed in? Well, they are executed in the order they are declared in either the ColdBox configuration file as interceptors or the WireBox configuration file as listeners.

Important: Order is EXTREMELY important for interceptors/listeners. So please make sure you order them in the declaration file.

Providers

Let's get funky now! We have seen how to inject objects and how to scope objects. However, we need to talk about a cool WireBox feature called *object providers*. We learned that when you request an object from WireBox it creates it and injects it immediately. However, sometimes we need more control like:

- Delay construction of the dependency until some point in time during your controlled execution. Maybe you don't want to construct some dependencies until some feature in your application is enabled.
- You need multiple instances of a class. Like a User service producing transient users, or our espresso machine creating espressos.
- You need to access scoped objects that might need reconstruction. Maybe you want to check the cache first for existence or a ColdFusion scope in order to avoid scope widening injection.
- You have some old legacy funkiness for building stuff that has to remain as its own factory

All of these areas is where WireBox Providers can really save the day. WireBox offers automatically a way to create providers for you by creating generic provider classes (`coldbox.system.ioc.Provider`) that will be configured to provide the mapping you want, then injected instead of the real object requested. This happens whenever you use the *provider* DSL injection namespace or annotate methods with a *provider* annotation. It also gives you an interface (`coldbox.system.ioc.IProvider`), which is very simple, that you can implement in order to register your own complex providers with WireBox. You would usually do the latter if you have legacy code you need to abstract out, had funky construction processes, etc. Let's start by looking at how registering custom providers work first and then how to use the automatic WireBox providers work.

Custom Providers

If you need to abstract old legacy code or have funky construction processes, we would recommend you build your own provider objects. This means that you will create a component that implements `coldbox.system.ioc.IProvider` (one `get()` method) and then you can map it. Once mapped, you can use it anywhere WireBox listens for providers:

- The injection DSL -> `inject="provider:[name]"`
- The mapping DSL
 - `toProvider('name')`
 - `setter.property.methodArg.initArg(name="",dsl="provider:[name]")`;

Here is the interface you need to implement:

```
<interface>The WireBox Provider Interface that follows the provider pattern*
<--- get --->
<functionname=get*output=false*access=public*returntype=any*hint=Get the provided object*
</functions>
</interface>
```

The CFC you build will need to be mapped so it can be retrieved by name and also so if it needs DI or any other WireBox funkiness, it can get it. So let's look at our *FunkyEspressoProvider* that we needed to create since we have some old legacy machines that we need to revamp:

```
component name=FunkyEspressoProvider implements=coldbox.system.ioc.IProvider singleton{
property name=log inject=logbox:logger:FunkyEspressoProvider*
public function init() {return this }
Espresso public function get(){
// log
log.canDebug(){ log.debug@usted funky espresso!
var espresso createObjectComponent*old.legacy.Espresso*init();
// add some sugar as the old legacy machine is not that great.
espresso.addSugar(1);
// returned provided object.
return espresso;
}
}
```

Finally we map to the provider using the `toProvider()` mapping method in the binder so anytime somebody requests an *Espresso* we can get it from our funky provider. Please note that I also map the provider because it also has some DI needed.

```
component extends=coldbox.system.ioc.config.Binder*
function configure(){
// map the provider first, so it can be constructed and DI performed on it.
map(FunkyEspressoProvider*
.toModel.legacy.FunkyEspressoProvider*
// map espresso's to the old funky provider for construction and retrieval.
map(Espresso*
.toProvider(FunkyEspressoProvider*
)
}
```

Cool! That's it, anytime you request an *Espresso*, WireBox will direct its construction to the provider you registered it with.

Virtual Provider Injection DSL

You can inject automatic object providers by using the *provider* injection dsl namespace. This will inject a WireBox provider class (`coldbox.system.ioc.Provider`) that follows our Provider pattern with one method on it: `get()` that will provide you with the requested mapped object. The difference between custom providers here is that WireBox will create a virtual provider object for you, configure it to retrieve a specific type of mapping and then use that for you.

```
// use the provider DSL namespace on a property
property name=searchCriteria inject=provider:requestCriteria*
// use the provider DSL namespace on a constructor argument
function init(coolObjectProvider inject=provider:HardToConstructObject*
variables.coolObjectProvider = arguments.coolObjectProvider;
return this
}
// To use it
searchCriteria.get().getCriteria();
coolObjectProvider.get().executeSomeMethod();
```

That's it! You basically use the *provider:[mapping]* injection DSL to tell a property, setter or argument that you want a provider object instead of the real deal. This will allow you to delay construction of such object or avoid the nasty pitfall of scope widening injection.

Virtual Provider Mapping DSL

You can also use our cool mapping DSL to create mappings that refer to provided objects by using the *dsl* construction type:

```
// map an object to a virtual provided object
map<fooObjectProvider>
.toDSL(provider:HardToConstructObject)

// map an object an set the explicit DI arguments or DI setters to virtual provided objects
map<SearchService>
.toModel(search.SearchService)
.initArg(name=searchCriteria:dsl#provider:requestCriteria)
```

You can use the following mapping methods to map to virtual providers by using their *dsl* arguments:

- mapDSL()
- initArg(name="", dsl="")
- property(name="", dsl="")
- setter(name="", dsl="")
- methodArg(name="", dsl="")

Virtual Provider Lookup Methods

This is a ColdFusion 9 feature only, where you can mark methods in your components with a special *provider* annotation so they can serve the objects you requested automatically for you. This is an amazing feature as it will take the original method signature and replace the method for you with one that will serve the provided objects for you automatically. How insane is that! You deserve some [quality sleep](#) *wait it damnning!*

```
public Expresso function getEspresso() provider=presso {}
```

Wow! That's it! Yep, just create an empty method signature and annotated with *provider={mapping}* and then WireBox will read these annotated methods and replace them for you at runtime so when you call *getEspresso()* it actually calls the WireBox injector and requests a new *Espresso* instance and it returns it.

Important: Please note that the visibility of provided methods does not matter to WireBox. It can provide public, private, or packaged visibilities with no problem at all.

Provider onMissingMethod Proxy

Thanks to our friend Brad Wood, we have a feature in our Providers that you can leverage its *onMissingMethod()* to proxy calls into the provided object itself. So let's say our provided object has a method called *sayHello()*, then with an injected provider you must do this:

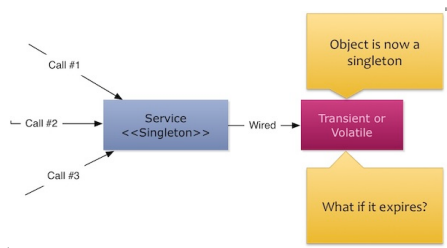
```
property name=chatter inject:provider=Chat
function useChatter(){
    return chatter.get().sayHello();
}
```

That is great, but you can proxy calls into the provider itself by doing this:

```
property name=chatter inject:provider=Chat
function useChatter(){
    return chatter.sayHello();
}
```

The WireBox provider object (*coldbox.system.ioc.Provider*) has an *onMissingMethod()* function that will take all missing method calls and proxy them to the provided object. Now, this is great but be ready to lose on performance if you use this approach. That is the only caveat to this approach, is that you will be impacted by performance, not crazy, but try it.

Scope Widening Injection

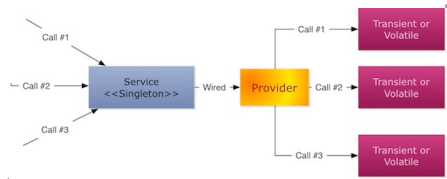


An object that is scoped into request, session, server, cachebox or application scopes and if wired into a persisted object will remain around even when this object has expired from the scope. This is called *scope-widening injection* and is a problem that must be addressed by NOT injecting them into persisted objects directly but by using WireBox's provider approach. This guarantees that the object's scope lifecycle will be maintained and your singleton or other persistent objects will be decoupled from the scope by accessing the target object via its provider.

For example, let's say you have a handler that wires in a user object that is scoped into session scope, but the handler itself is scoped as a singleton:

```
component name=handler singleton{
    property name=user inject:userId=user
}
//user component
component name=user scope=session{
}
```

So when the handler is created and persisted as a singleton, the user object gets created, stored in session and also referenced into the lifecycle of the handler object. So now, if the user expires from session, the handler does not know about it, because all it knows is that a direct reference to that out of context object still remains. So if the user needed things in session to exist, this will now fail. This problem is much how hibernate and detached objects works. Objects are no longer in session, they are detached. This scope widening issue is resolved by **NOT** injecting the user object directly into the handler but by using a provider.



Scope Widening Injection Solution: Object Providers

Below is my favorite approach to solving the issue which is by using provided methods:

```
component name=handler singleton{
    function getUser() provider=user {}
}
```

That's it! My *getUser()* method will be replaced by WireBox with a proxy provider method that will request from the WireBox injector the *user* mapping instance.

Object Persistence & Thread Safety

While the injector can help in many ways to secure the creation of your objects, it is ultimately up to you to create code that is both thread safe and tested. It is always a great idea to design your objects without the injector in mind for threading and concurrency. DI is not a silver bullet, but a tool to relieve object creation and not to relieve the burden of good object design. Thread safety is much more complex and can be compromised when using persistent scopes like singleton, session, server, application and cache box, as more than one thread will be trying to access your code and dependencies. The only guarantee the injector can provide is the constructor and constructor dependency creation to be completely locked. The following object is to be guaranteed to be locked when created and wired with dependencies:

```
component {
    /**
     * @log inject logbox:logger:{this}
     * @dao inject id:MyDAO
     */
    function init(required log, required dao){
        variables.log = arguments.log;
        variables.dao = arguments.dao;
        return this;
    }
}
```

Important: The inject annotations are done in comments as ColdFusion 9 has a bug when adding annotations on scripted arguments.

An example of a flawed object could be the following:

```
component {
    property name=dao inject:MyDAO
    property name=log inject:logbox:logger:{this}
    function init(){
        return this;
    }
}
```

Why is this object flawed? It is flawed because the majority of DI engines, including WireBox, will lock for constructing the object and its constructor arguments. However, once it is constructed, it will store the object in the persistence scope of choice in order to satisfy the potential of circular dependencies in the object graph. After it is placed in the storage, the DI engines will wire up setter and property mix injections and WireBox *onDIComplete()* method. With this normal approach, the wiring of dependencies and *onDIComplete()* have the potential of mixups due to concurrency. This is a normal side-effect and risk that developers take due that Java makes no guarantees that any thread other than the one that set its dependencies will see the dependencies. The memory between threads is not final or immutable so properties can enter an altered state.

¹The subtle reason has to do with the way Java Virtual Machines (JVM) are designed to manage threads. Threads may keep local, cached copies of non-volatile fields that can quickly get out of sync with one another unless they are synchronized

correctly." From Dependency Injection by Dhanji R. Prasanna

Note: This side effect of concurrency will only occur on objects that are singletons or persisted in scopes like session, server, application, server or cachebox. It does not affect transient or request scoped objects.

WireBox, can help you lock and provide thread safety to setter and property injections by providing you with the **ThreadSafe** annotation or our binder **threadSafe()** tagging method. So if we wanted to make the last example thread safe for property and setter wiring then we would do the following:

```
component threadSafe{
    property name=dao inject:MyDAO
    property name=log inject:logbox:logger:{this}
    function init(){
        return this
    }
}
// or
component threadSafe{
    property name=dao inject:MyDAO
    property name=log inject:logbox:logger:{this}
    function init(){
        return this
    }
}
// or you can bind it as a thread safe component
map(MyObject) to {path.model.MyObject::Singleton()}.threadSafe();
```

Note: All objects are marked as **non thread safe** for dependency wiring by default in order to allow for circular dependencies. Please note that if you mark an object as **threadSafe**, then it will not be able to support circular dependencies unless it uses WireBox providers. (See Providers Section)

Our **threadSafe** annotation and binder tagging property will allow for these objects to be completely locked and synchronized for object creation, wiring and *onDiComplete()*. However, circular dependencies will now fail as persistence cannot be guaranteed for the setter or property dependencies. However, since WireBox is so awesome, you can still use circular dependencies by wiring instead our object providers. (Please see providers section). In conclusion, constructing and designing a CFC that is thread safe is often a very arduous process. It is also very difficult to test and recreate threading issues in your objects and applications. So don't feel bad, as even the best of us can get into some nasty wormholes when dealing with concurrency and thread safety. However, always try to design for as much concurrency as possible and test test!

Custom DSL Builder

WireBox allows you to create your own DSL object builders and then register them via your configuration binder. This allows you to create a namespace or override an internal namespace with your own object builder. By now we have seen our injection DSL and noticed that we have internal namespaces. With this feature we can alter it or create new ones so our annotations and injection DSLs can be customized to satisfaction. This is easily done in the following process

1. Create a CFC that implements *coldbox.system.ioc.dsl.IDSLBuilder*
2. Register your custom DSL builder in your configuration binder

The DSL Builder Interface

<interface> hint="The main interface to produce WireBox namespace DSL Builders"

```
<--- init --->
<function name=init output=false access=public returnType=any>
<argument name=injector type=any required=true>
<argument name=targetObjectType type=any required=false>
</function>
<--- process --->
<function name=process output=false access=public returnType=any>
<argument name=definition required=true>
</function>
</interface>
```

The main method here to notice is the *process()* method. This method receives the injection dsl annotation value and it will be up to the builder to parse it and decide what to do with it. If this method returns *null* then WireBox will log it for you at that specific injection dependency resolution. However, we also encourage you to use best practices and log from your builders also.

Registering a custom DSL builder

If you refer back to the configuration section you can see the **customDSL** structure or **mapDSL()** method that you can use for registering custom DSL namespaces.

```
customDSL = {
    ortus => path.model.dsl.OrtusBuilder
};
```

```
or
mapDSL(ortus) => path.model.dsl.OrtusBuilder
```

This will register a new injection DSL namespace called **ortus** that maps to that instantiation component **path.model.dsl.OrtusBuilder**. Here is a very simple DSL Builder:

```
<component implement=coldbox.system.ioc.dsl.IDSLBuilder output=false>
<--- init --->
<function name=init output=false access=public returnType=any hint=Configure the DSL for operation and returns coldbox.system.ioc.dsl.IDSLBuilder>
<argument name=injector type=any required=true hint=The linked WireBox injector to use for injection>
<script>
instance = { injector = arguments.injector };
instance.log = instance.injector.getLogBox().getLogger();
return this
</script>
</function>
<--- process --->
<function name=process output=false access=public returnType=any hint=Process an incoming DSL definition and produce an object with it.>
<argument name=definition required=true hint=The injection dsl definition structure to process. Keys: name, dsl>
<script>
var thisType = arguments.definition.dsl;
var thisTypeLen = listen(thisType);
var utilName = "";
// DSL stages
switch(thisTypeLen){
// Ortus
case 1 : {return instance.injector.getInstance( util ); }
// Ortus utility
case 2 : {
utilName = getToken(thisTypeLen, 2);
// Verify that cache exists
if( instance.injector.containsInstance( utilName ) ){
return instance.injector.getInstance( utilName );
}
elseif( instance.log.isDebugEnabled() ){
instance.log.debug( "not find named ortus utility #utilName# using definition: #arguments.definition.toString()#" );
}
break
} // end level 2 main DSL
</script>
</function>
</component>
```

As you can see from the sample, creating your own DSL builder is fairly easy. The benefits of a custom DSL builder is that you can very easily create and extend the injection DSL language to your own benefit and if you are funky enough, override the behavior of the internal DSL Namespaces.

Custom Scopes

WireBox allows you to create your own object persistence scopes so you can have full control on their lifecycle. This is easily done in the following process:

1. Create a CFC that implements *coldbox.system.ioc.scopes.IScope*
2. Register your custom scope in your configuration binder

You can create your own persistence scope or if you are getting funky, override the internal persistence scopes with your own logic.

The Scope Interface

<interface> hint="The main interface to produce WireBox storage scopes"

```
<--- init --->
<function name=init output=false access=public returnType=any hint=Configure the scope for operation and returns coldbox.system.ioc.scopes.IScope>
<argument name=injector type=any required=true hint=The linked WireBox injector to use for injection>
</function>
<--- getFromScope --->
<function name=getFromScope output=false access=public returnType=any hint=Retrieve an object from scope or create it if not found in scope>
<argument name=mapping type=any required=true hint=The object mapping to use for injection>
<argument name=initArguments type=any required=false hint=The constructor structure of arguments to passthrough when initializing coldbox.system.ioc.scopes.IScope>
</function>
</interface>
```

Scoping Process

The scoping process must be done by using some of the referenced injector's methods:

- **buildInstance(mapping, initArguments)**
- **autoWire()**

These methods must be called sequentially in order to avoid circular reference locks. The first method **buildInstance** is used to construct and initialize an object instance. The **autoWire** method is used then to process DI and AOP on the targeted object. Let's look at my Ortus Scope:

```
<component output=false implement=coldbox.system.ioc.scopes.IScope>
<--- init --->
<function name=init output=false access=public returnType=any hint=Configure the scope for operation>
<argument name=injector type=any required=true hint=The linked WireBox injector to use for injection>
<script>
instance = {
    injector = arguments.injector,
    ortus = {}
};
```

```

    });
    return this
  </cfscript>
</cffunction>

<!-- getFromScope -->
<cffunction name=getFromScope output=false access=public returntype=any hint=Retrieve an object from scope or create it if not found in scope
  <cfargument name=mapping type=any required=true hint=The object mapping idoc generated by coldbox.system.ioc.config.Mapping
  <cfargument name=initArguments type=any required=false hint=The constructor structure of arguments to passthrough when initializing the object structure
  </cfargument>
  <cfscript>
    var name = arguments.mapping.getName();
    if ( structKeyExists(instance.ortus, name) ){
      return instance.ortus[name];
    }

    lock name=ortus.scope.#arguments.mapping.getName()#
    instance.ortus[name] = instance.injector.buildInstance( arguments.mapping, arguments.initArguments );
  }

  // wire it
  instance.injector.autowire(target=instance.ortus[name], mapping=arguments.mapping);

  // send it back
  return instance.ortus[name];
  </cfscript>
</cffunction>
</cfcomponent>

```

Important: Always make sure that you use the `buildInstance` method and then store the results in the scope before wiring is done to avoid endless loops errors.

Registering A Custom Scope

If you refer back to the configuration section you can see the `customScopes` structure or `mapScope()` method that you can use for registering custom DSL namespaces.

```

customScopes = {
  ortus = "path.model.dsl.OrthusScope"
};

or
mapScope(ortus,"path.model.dsl.OrthusScope"

```

Now I can use the `ortus` scope in my mappings DSL and even my annotations, isn't that cool!

```

component scope=ortus{
}

// map it
map(Luis)
  to(model.path.LuisService*
  .into(ortus);

```

WireBox Injector Interface

We also provide an interface to create objects that adhere to our injector interface: `coldbox.system.ioc.Injector`. Then these objects can be used as parent injectors, great for legacy factories or creating hierarchies according to your specs. All you have to do is implement the following interface:

```

<cfinterface hint=An interface that enables any CFC to act like a parent injector within WireBox>
  <!-- getParent -->
  <cffunction name=getParent output=false access=public returntype=void hint=Link a parent injector with this injector
  <cfargument name=injector required=true hint=A WireBox injector to assign as a parent to this>injector
  </cffunction>

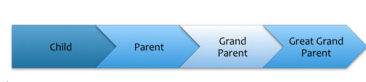
  <!-- getParent -->
  <cffunction name=getParent output=false access=public returntype=any hint=Get a reference to the parent injector instance, else an empty simple string meaning nothing is set
  </cffunction>

  <!-- getInstance -->
  <cffunction name=getInstance output=false access=public returntype=any hint=Locates, Creates, Injects and Configures an object model instance
  <cfargument name=name required=false hint=The mapping name or CFC instance path to try to build up
  <cfargument name=dsl required=false hint=The dsl string to use to retrieve the instance model object, mutually exclusive with 'name'
  <cfargument name=initArgument required=false hint=The constructor structure of arguments to passthrough when initializing the instance
  </cffunction>

  <!-- containsInstance -->
  <cffunction name=containsInstance output=false access=public returntype=any hint=Checks if this injector can locate a model instance or not
  <cfargument name=name required=true hint=The object name or alias to search for if this container can locate it or has knowledge of it
  </cffunction>

  <!-- shutdown -->
  <cffunction name=shutdown output=false access=public returntype=void hint=Shutdown the injector gracefully by calling the shutdown events internally.
  </cffunction>
</cfinterface>

```



Once you create this CFC that implements this interface then you can call on the injector's `setParent()` method and you are ready to roll.

WireBox Object Populator

WireBox also comes packaged with our handy ColdBox object populator that has been so successful in our ColdBox applications. The object populator object can populate objects with data from XML, JSON, WDDX, structures, queries and much more. So we highly encourage you to check it out as it will really help out in your applications. The way to retrieve it is to use the `getObjectPopulator()` method on the injector and then call one of our populate methods you can see below. Please check out the [cfc-docs-xml](#) for the latest and greatest.

```
populator = injector.getObjectPopulator();
```

populateFromXML

Populate a bean from an XML packet

Returns

- This function returns *any*

Arguments

Key	Type	Required	Default	Description
target	any	Yes	---	The target to populate
xml	any	Yes	---	The XML string or packet
root	string	No	---	The XML root element to start from
scope	string	No	---	Use scope injection instead of setters population. Ex: scope=variables.instance.
trustedSetter	boolean	No	false	If set to true, the setter method will be called even if it does not exist in the bean
include	string	No	---	A list of keys to include in the population
exclude	string	No	---	A list of keys to exclude in the population

populateFromQuery

Populate a bean from a query

Returns

- This function returns *Any*

Arguments

Key	Type	Required	Default	Description
target	any	Yes	---	The target to populate
qry	query	Yes	---	The query to populate the bean object with
rowNumber	Numeric	No	1	The query row number to use for population
scope	string	No	---	Use scope injection instead of setters population. Ex: scope=variables.instance.
trustedSetter	boolean	No	false	If set to true, the setter method will be called even if it does not exist in the bean
include	string	No	---	A list of keys to include in the population
exclude	string	No	---	A list of keys to exclude in the population

populateFromStruct

Populate a bean from a structure

Returns

- This function returns *any*

Arguments

Key	Type	Required	Default	Description
target	any	Yes	---	The target to populate
memento	struct	Yes	---	The structure to populate the object with.
scope	string	No	---	Use scope injection instead of setters population.
trustedSetter	boolean	No	false	If set to true, the setter method will be called even if it does not exist in the bean
include	string	No		A list of keys to include in the population
exclude	string	No		A list of keys to exclude in the population

populateFromQueryWithPrefix

Populates an Object using only specific columns from a query. Useful for performing a query with joins that needs to populate multiple objects.

Returns

- This function returns *any*

Arguments

Key	Type	Required	Default	Description
target	any	Yes	---	This can be an instantiated bean object or a bean instantiation path as a string. If you pass an instantiation path and the bean has an 'init' method. It will be executed. This method follows the bean contract (set{property_name}). Example: setUsername(), setName()
qry	query	Yes	---	The query to populate the bean object with
rowNumber	Numeric	No	1	The query row number to use for population
scope	string	No		Use scope injection instead of setters population. Ex: scope=variables.instance.
trustedSetter	boolean	No	false	If set to true, the setter method will be called even if it does not exist in the bean
include	string	No		A list of keys to include in the population
exclude	string	No		A list of keys to exclude in the population
prefix	string	Yes	---	The prefix used to filter. Example: 'user_' would apply to the following columns: 'user_id' and 'user_name' but not 'address_id'.

populateFromJSON

Populate a bean from a json string

Returns

- This function returns *any*

Arguments

Key	Type	Required	Default	Description
target	any	Yes	---	The target to populate
JSONString	string	Yes	---	The JSON string to populate the object with. It has to be valid JSON and also a structure with name-key value pairs.
scope	string	No		Use scope injection instead of setters population. Ex: scope=variables.instance.
trustedSetter	boolean	No	false	If set to true, the setter method will be called even if it does not exist in the bean
include	string	No		A list of keys to include in the population
exclude	string	No		A list of keys to exclude in the population

Mapping DSL Examples

```
mapPath(path)
mapDSL(cool; "model.CoolFactory")

map("SecurityService"
  .to(model.security.SecurityService*
  .onDICOComplete{@art:"executeRole"})

mapDirectory(shared/model)

// Eager initialized objects
map("luis, joe"; to(model.Luis; in this.SCOPE.SINGLETON).asEagerInit())
map("luis; joe").to(model.Luis; in this.SCOPE.SINGLETON).asEagerInit()

// map a property to a mapping id via DSL
map("Lui").toDSL(coldbox:setting:luis)

// using initWith() for passing name-value pairs or positional arguments for direct initialization of a mapping
map("transferConfig"
  .to(transfer.com.config.Configuration*
  .initWith(datasourcePath=getProperty("datasourcePath"
    configPath=getProperty("configPath"
    definitionPath=getProperty("definitionPath");

// Now doing with setter injection
map("transferConfig"
  .to(transfer.com.config.Configuration*
  .setter(name=datasourcePath, value=getProperty("datasourcePath")
  .setter(name=configPath, value=getProperty("configPath")
  .setter(name=definitionPath, value=getProperty("definitionPath");

// Map with constructor arguments
map("transfer"
  .to(transfer.com.Transfer*
  .in(SCOPE.SINGLETON)
  .noAutowired()
  .asEagerInit()
  .initArg(name=configuration, ref="transferConfig") //ref = name by default, or have an explicit name

// RSS Integration With Caching.
map("googleNews"
  .toRSS(url="http://news.google.com/news?pz=1&ned=us&hl=en&topic=h&num=3&output=rss"
  .asEagerInit()
  .inCacheBox(timeout=20, lastAccessTimeout=30, provider=key#google-news);

// Java Integration with init arguments
map("Buffer"
  toJava(java.lang.StringBuffer*
  initArg(value=0; javaCast@long);

// Java integration with initWith() custom arguments and your own casting.
map("Buffer"
  toJava(java.lang.StringBuffer*
  initWith( javaCast@long; 500) );
```