ColdSpring Framework 1.0 Documentation

Dave Ross
Chris Scott
Kurt Wiersma
Sean Corfield
Simeon Bateman

Table Of Contents

I. Introduction To ColdSpring

ColdSpring is a inversion-of-control framework/container for CFCs (ColdFusion Components). Inversion of Control, or IoC, is synonymous with Dependency Injection, or DI. Dependency Injection is an easier term to understand because it's a more accurate description of what ColdSpring does. ColdSpring borrows its XML syntax from the java-based Spring Framework, but ColdSpring is not necessarily a "port" of Spring.

A dependency is when one piece of a program depends on another to get its job done. We'll use the example of a simple CFC designed to manage your user's shopping carts on an online store. The requirements dictate that it needs to calculate tax on order totals. You could build this functionality directly into your `ShoppingCartManager`, but it might be a better idea to create a `TaxCalculator` CFC to do this particular job. You might need tax calculation somewhere else in your program, outside of your shopping cart manager, and like most software developers, you want your code to be as reusable as possible. A `TaxCalculator`, which does nothing but calculate tax, is an example of highly cohesive (and thus highly reusable) code. However, you may have noticed in past projects that as the more cohesive things get, the more work there is in "keeping things together" (referred to as "coupling"). Along with greater cohesion, loosening the amount of coupling in your code is another hallmark of software development.

Every time a piece of your code instantiates and uses your tax calculator, two pieces of your application are effectively tied together (and thus they are known as "collaborators"). Collaboration is natural and understandable in an application, but you still want to do everything possible to keep coupling to a minimum. Your components need to know how to *use* the `TaxCalculator`, but should the knowledge of how to create and configure the `TaxCalculator` be sprinkled into each and every one? ColdSpring enables to you to remove those bits of code by directly managing your component's dependencies, but surprisingly your code won't have any idea that it's there. This means that your components don't have to create or find their collaborators, they are simply *given* them by ColdSpring! This act of giving is known as "injection", which is why "Dependency Injection" is a term that accurately describes what ColdSpring does.

Before, your `ShoppingCartManager` would just create its own TaxCalculator with a createObject() call. With ColdSpring, the `ShoppingCartManager` is instead injected with an instance of the `TaxCalculator` (also referred to as an "object reference"). In order for the ColdSpring to be able to do this, you will need one of two things:

1. A "setter" method that will accept the `TaxCalculator` instance (a method named `setTaxCalculator`).
   or

2. An argument to the ShoppingCartManager constructor that will accept the TaxCalculator instance.

Here's a before/after example of what your ShoppingCartManager constructor might look like:

Before ColdSpring:
```
<cffunction name="init" returntype="myApp.model.ShoppingCartManager" output="false" hint="constructor">
     <cfargument name="MaxItems" type="numeric" required="true" hint="max items for any of my user's carts"/>
     <cfset variables.TaxCalculator = createObject("component","myApp.model.TaxCalculator").init()/>
     <cfset variables.MaxItems = arguments.MaxItems/>
</cffunction>
```

After ColdSpring:
```
<cffunction name="init" returntype="myApp.components.ShoppingCartManager" output="false" hint="constructor">
     <cfargument name="TaxCalculator" type="myApp.model.TaxCalculator" required="true" hint="Dependency
     TaxCalculator"/>
     <cfargument name="MaxItems" type="numeric" required="true" hint="max items for any user's cart"/>
     <cfset variables.TaxCalculator = arguments.TaxCalculator/>
     <cfset variables.MaxItems = arguments.MaxItems/>
</cffunction>
```

Ok, so this doesn't *look* like any less code, nor does it look any cleaner. Maybe some added complexity will start to reveal the differences. Let's say our `TaxCalculator` needs to be told the current tax rate in order to function properly. Without ColdSpring, our constructor changes to:
```
<cffunction name="init" returntype="myApp.model.ShoppingCartManager" output="false" hint="constructor">
     <cfargument name="TaxRate" type="numeric" required="true" hint="Current Tax Rate"/>
     <cfargument name="MaxItems" type="numeric" required="true" hint="max items for any user's cart"/>
     <cfset variables.TaxCalculator =
     createObject("component","myApp.components.TaxCalculator").init(arguments.TaxRate)/>
     <cfset variables.MaxItems = arguments.MaxItems/>
</cffunction>
```

Does the "After ColdSpring" version of the constructor change? **It doesn't!** In fact, you might have noticed that the `ShoppingCartManager` is now receiving a tax rate (and passing it to the `TaxCalculator`), which to me is outside the scope of what the `ShoppingCartManager` is meant to do. In the ColdSpring example, the `TaxCalculator` comes in ready-to-use, and the `ShoppingCartManager` is not burdened with creating and configuring it.

So, what is better about the ColdSpring/dependency-injection approach?
- components aren't asked to do things outside of their scope or duty (known as "separation of concerns")
- components aren't completely tied to other implementations (again, less coupling)
- components are easier to configure and you can do so without changing code

- components become easier to test (we can dictate which collaborators they use, perhaps even creating dummy "stub" or "mock" objects to trick the component into thinking that it's running in a different environment).
- you can get a birds eye view of the dependencies among your components (and generate some neat documentation)
- components are not tied to ColdSpring at all. There should be very little plumbing required to use ColdSpring in any environment, and only "calling" code will be aware of its existence. In Model-View-Controller apps, this usually means that the Controller will have some knowledge of ColdSpring but nothing else will.

To use ColdSpring with your components, you simple create a configuration file (or files) that contain some simple xml tags which tell ColdSpring about your components (and their dependencies/configuration).

Take a look at the following xml snippet:

```
<bean id="ShoppingCartManager" class="myApp.model.ShoppingCartManager"/>
```

A ColdSpring "bean" tag is how you define your component(s), but don't read too much into the nomenclature. ColdSpring uses the `<bean/>` syntax because it relies heavily on the "Java-beans" specification to resolve your component's dependencies. All the above xml snippet says is "hey, ColdSpring… register `myApp.components.ShoppingCartManager` under the name "ShoppingCartManager". Doing so means at any time, you can ask ColdSpring to give you the "ShoppingCartManager" and it will return a `myApp.components.ShoppingCartManager` instance. Typically this instance is usually shared among all that ask (aka a "singleton"), however you can define your bean such that each time your code asks for ColdSpring form the component it will receive a new instance.

Most bean definitions will probably be a bit more complex, e.g. you might have some configuration details to pass to the `ShoppingCartManager` … perhaps into the constructor (your CFC's init() method), as shown in this `<bean/>` snippet:

```
<bean id="ShoppingCartManager" class="myApp.model.ShoppingCartManager">
    <constructor-arg name="MaxItems">
        <value>15</value>
    </constructor-arg>
</bean>
```

or as a property (e.g., a setterMethod – you would need to have a setMaxItems(items) method in your `ShoppingCartManager` for the following to work):

```
<bean id="ShoppingCartManager" class="myApp.components.ShoppingCartManager">
    <property name="MaxItems">
        <value>15</value>
    </property>
</bean>
```

Ok, let's look back at our problem… we need to "inject" a `TaxCalculator` into our `ShoppingCartManager`. This is simpler than you think – we have to define the `TaxCalculator` (and we'll supply it with the current tax rate):

```
<bean id="TaxCalculator" class="myApp.components.TaxCalculator">
    <constructor-arg name="TaxRate">
        <value>0.8</value>
    </constructor-arg>
</bean>
```

Then, in our `ShoppingCartManager` bean definition, we can reference the `TaxCalculator` by using the `<ref/>` tag:

```
<bean id="ShoppingCartManager" class="myApp.components.ShoppingCartManager">
    <constructor-arg name="MaxItems">
        <value>15</value>
    </constructor-arg>
    <constructor-arg name="TaxCalculator">
        <ref bean=TaxCalculator/>
    </constructor-arg>
</bean>
```

Now, when ColdSpring creates the `ShoppingCartManager`, it will pass in (inject) the `TaxCalculator` instance. The `ShoppingCartManager` has no idea where the `TaxCalculator` came from but it is perfectly happy to use it. You've now removed unnecessary code from the `ShoppingCartManager`, and loosened its coupling to the `TaxCalculator`, effectively making your code more reusable, testable, and maintainable.

II. BeanFactory Reference

Think of a ColdSpring BeanFactory as the container, or holder, for your application's components. It will instantiate, configure, and resolve the dependencies among the components (beans) you place inside it. ColdSpring is currently shipping with only one implementation of a BeanFactory. It's very possible that there will be others in the future, but the current implementation, coldspring.beans.DefaultXmlBeanFactory will be the one demonstrated primarily in this reference.

II.I Installing ColdSpring and creating the BeanFactory

To install ColdSpring, you must either place the source within your ColdFusion server's webroot, or create a ColdFusion mapping within the ColdFusion administrator named /coldspring that points to the location of the ColdSpring source code. To create a ColdSpring BeanFactory, you would simply use the createObject method in CFML. However, you may want to prepare two structures beforehand to pass in as arguments to the BeanFactory's constructor.

1. **"defaultProperties"**
   Currently, this is a simple way to pass in a struct of actual configuration data into the BeanFactory and then use a syntax like ${key} in place of using an actual value within the <bean/> definitions. Eventually this will be expanded/refactored into entire CFML expression support.
2. **"defaultAttributes"**
   A ColdSpring BeanFactory has the notion of bean attribute "defaults". This means that, for a given instance of DefaultXmlBeanFactory, you can configure default behavior that will be applied to all beans that don't explicitly override what you've set.

Both structures can be ignored and the BeanFactory will use its own internal defaults. An example of creating the BeanFactory follows:

```
<cfset myBeanFactory = createObject("component","coldspring.beans.DefaultXmlBeanFactory").init()/>
```

You should be able to run the above line of code without error if ColdSpring is installed correctly on your server.

II.II Supplying the BeanFactory with your bean definitions

The DefaultXmlBeanFactory implementation can only read bean definitions from xml. There is no way to programmatically add bean definitions to this implementation (however one could construct the necessary xml on the fly and give that to the DefaultXmlBeanFactory - as shown below).

Currently, there are 3 ways to add bean definitions to the DefaultXmlBeanFactory:

1. Pass the DefaultXmlBeanFactory a fully qualified path to a bean definition xml file.
```
<!--- void loadBeansFromXmlFile(string beanDefinitionFile, boolean ConstructNonLazyBeans)
--->
<cfset myBeanFactory = createObject("component","coldspring.beans.DefaultXmlBeanFactory").init()/>
<cfset myBeanFactory.loadBeansFromXmlFile("/path/to/file.xml",true)/>
```

2. Pass the DefaultXmlBeanFactory a string containing raw unparsed xml.
```
<!--- void loadBeansFromXmlRaw(string beanDefinitionXml, boolean ConstructNonLazyBeans) --
->
<cfset myBeanFactory = createObject("component","coldspring.beans.DefaultXmlBeanFactory").init()/>
<cfsavecontent variable="beanConfigs">
<beans>
<bean id="myFirstBean" class="myApp.model.myFirstBean"/>
</beans>
</cfsavecontent>
<cfset myBeanFactory.loadBeansFromXmlRaw(beanConfigs,true)/>
```

3. Pass the DefaultXmlBeanFactory a parsed Coldfusion xml object.
```
<!--- void loadBeansFromXmlObj(any beanDefinitionXmlObj, boolean ConstructNonLazyBeans) --
->
<cfset myBeanFactory = createObject("component","coldspring.beans.DefaultXmlBeanFactory").init()/>
<cffile action="read" file="/path/to/file.xml" variable="xmlContent"/>
<cfset someXml = xmlParse(xmlContent)/>
<cfset myBeanFactory.loadBeansFromXmlObj(someXml,true)/>
```

You're probably wondering what "ConstructNonLazyBeans" does, but first we'll explain the basics of configuring the DefaultXmlBeanFactory and the beans you put in it.

II.III <bean/> tag attributes

To explore all of the attributes of a ColdSpring bean definition, one could look at the J2EE Spring framework's DTD (which ColdSpring expects you to adhere to). However, not every attribute or tag is fully implemented in ColdSpring, and there are some that aren't applicable to CFC development, so they are simply ignored.
ColdSpring beans are defined via the `<bean/>` tag, and here are the attributes of the `<bean/>` tag worth mentioning (attributes in **`bold`** are required):

Attribute Name	**Description and Use**	**Implemented/ Planned/ Won't Implement**
`id`	This is the identifier used to store your bean. When you ask ColdSpring to give you a reference to one of its beans, you'll use this same identifier.	Implemented
`name`	Serves the same purpose as id, however can accept multiple identifiers via a comma separated list. This effectively allows you to define your bean as having several aliases.	Planned
`class`	The actual CFC type to create for this bean definition.	Implemented
`singleton`	true|false – When true, indicates whether one shared instance of your bean will be kept by the BeanFactory and returned to all retrieval requests. When false, aImplemented new instance will be created and returned to each retrieval request.	Implemented
`init-method`	A name of a method that ColdSpring will call on a bean after all its dependencies have been set. Since we use "init" as a constructor in CFCs, "setup" or "configure" are good alternatives. If your CFC needs to do something with one or more of its dependencies immediately after receiving them, init-method is the easiest way to do it.	Implemented
`lazy-init`	true|false – When true, ColdSpring won't create the bean (or any dependencies of the bean that haven't been created) until it is asked for the bean. When false, ColdSpring will create the bean immediately upon receiving its definition (unless the method used to populate the BeanFactory tells ColdSpring not to via. the ConstructNonLazy beans argument - see II.II)	Implemented
`destroy-method`	If implemented, ColdSpring would call this method on a bean	Won't Implement

	before it is destroyed. Won't Implement	
`autowire`	no|byName|byType -Tells ColdSpring to autowire in dependencies by looking at the public properties (setters) of your bean and seeing if it knows about a bean that would match the signature. It will look for a match either by the name (or id) of a bean, or by the bean's type.	Implemented (except for "constructor" and autodetect values).
`depends-on`	If implemented, would explicitly tell the beanFactory to fully instantiate the bean specified by this attribute before creating the bean that defines depends-on.	Won't Implement
`factory-method`	Causes ColdSpring to call this method on the class defined in the bean to return the actual instance to use for this bean.	Won't Implement (by itself, see factory-bean)
`factory-bean`	id of a bean, known to Coldspring, on which the specified factory-method would be called to obtain an instance.	Implemented (use with factory-method)

Those are the standard attributes of a bean tag. There are a few others that are seriously outside the scope of ColdSpring due to the differences between CFCs and Java classes, so I won't mention them here.

II.IV The <bean/> tag's children

There are only two available child-tags of `<bean/>`, typically used to express dependencies among your beans or to supply the bean with some type of data (usually configuration information, or placeholders for configuration).

The `<bean/>` child tags implemented in ColdSpring are:

1. `<constructor-arg name="argumentName"/>`
   This tag will cause Coldspring to supply your bean with a value or object reference when it is instantiated (during a CFC's init() method), passed as an argument named via the name attribute.

2. `<property name="propertyName" />`
   Similar in nature to constructor arg, however in this case ColdSpring will pass some value or object reference into your bean as an argument to a setter method, identified via the name attribute. Thus, your CFC must have a setter method

name that matches the property tag's name attribute (for example if your property is named "foo" then your CFC needs a setFoo() method).

The `<lookup-method />` tag has yet to be implemented in ColdSpring. If you are interested in what it does, it is a way of injecting a method into a CFC that can then be used by that CFC to retrieve a bean from the factory. Think of it like mailing someone a cellphone with your number punched in rather then calling them directly.

II.V Children of <constructor-arg/> and <property/>

Both `<constructor-arg/>` and `<property/>` can accept a wide range of child tags, used to define what values or object references need to be passed into the constructor argument or property setter, respectively.
The table below lists all currently available child tags to both `<constructor-arg/>` and `<property/>`

Tag	Example Usage	Description
`<value/></value>`	`<value>15</value>` `<value>${key.subKey}</value>`	Used to pass in an arbitrary value, defined either directly in the xml or in the defaultProperties supplied to the BeanFactory.
`<ref/>`	`<ref bean="myBeanId"/>`	Used to pass in a reference to another bean defined within the BeanFactory. The bean="" attribute references the ID of the other bean.
`<bean />`	`<bean id="foo" class="foo" …` `</bean>`	Can be used to define an entire bean to be used only for the purpose of injecting into another bean. All attributes and child tags will be available.
`<map/>`	`<map>` `<entry key="foo">` `<value>5</value>` `</entry>` `<entry key="bar">`	Will pass a struct into your bean. Each entry within the map will correspond to a key within the passed-in struct. The child tags of <entry/> are any tags

	`<ref id="barBean"/>`
	`</entry>`
	`</map>`
		listed here, including map. Since CF only supports simple values for struct keys, only the key="" attribute of entry is supported.

`<list/>`	`<list>`
	`<ref id="barBean"/>`
	`<value>5</value>`
	`</list>`
		Like <map/>, but an array instead of a struct. Child tags can include anything listed here (including <map/> and <list/>)

The other tags specified by the Spring DTD, <null/>, <props/>, and <set/> are either yet-to-be implemented or won't be implemented in ColdSpring.
There is no limit to the "depth" of your bean definitions, demonstrated by this example snippet (which will work provided the CFC's exist):

```
<bean id="bean1" class="path.to.bean1">
    <constructor-arg name="bean2">
        <bean id="bean2" class="path.to.bean2">
<property name="bean3">
                <bean id="bean3" class="path.to.bean3">
                    <property name="bean4">
                        <ref bean="bean4"/>
                    </property>
                </bean>
</property>
        </bean>
    </constructor>
</bean>
<bean id="bean4" class="path.to.bean4"/>
```

II.VI Autowiring

The term "autowire" refers to the ability of the ColdSpring beanFactory to automatically wire dependent objects together without necessarily having to define those dependencies in the xml bean definitions. When you define your CFCs within a ColdSpring beanFactory, it inspects each CFC's metadata and sees if there are properties or constructor-arguments that match other CFC's it knows about. By default, autowiring is turned off within the BeanFactory, but if you turn it on (you can do it bean-by-bean or for an entire set of beans), ColdSpring will automatically inject dependencies (automatically wire things together).

There are two autowiring settings, `"byName"` and `"byType"`. As you can probably guess, `"byName"` matches components by their name, meaning if you have a `setSomeService(...)` method in a component or a constructor-argument named `"SomeService"`, and ColdSpring knows a `<bean/>` with an `id="SomeService"`, it will inject whatever `"SomeService"` is into your component. `"byType"` works by using the `"type"` attribute in your ColdFusion code, meaning if you have a setter-method with an argument which is `type="type.of.SomeService"` (or a constructor-argument), and ColdSpring knows a bean with `class="type.ofSomeService"` it will inject that `<bean/>` into your component.

To set autowiring on an entire set of `<beans/>`, use the `default-autowire` attribute:

```
<beans default-autowire="byName">
    <bean id="cfc1" class="..."/>
    <bean id="cfc2" class="..."/>
    ...
</beans>
```

To set autowiring on an individual bean (this will override any `default-autowire` setting), use the `autowire` attribute:

```
<beans>
    <bean id="cfc1" class="..." autowire="byType"/>
    <bean id="cfc2" class="..."/>
    <bean id="cfc3" class="..." autowire="byName"/>
    ...
</beans>
```

II.VII Using Your Own Factories

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

To use this functionality, you must first define the factory you want to use:

```
<bean id="someDAOFactory" class="myapp.components.SomeDAOFactory">
    <constructor-arg name="databaseType">
        <value>MySQL</value>
    </constructor-arg>
</bean>
```

Then, you can use your factory to define other ColdSpring beans, by using the `factory-bean` attribute to point at your factory and the `factory-method` attribute to indicate which method ColdSpring should call on your factory to obtain an instance from it. Assuming `someDAOFactory` above has a `getDAO()` method that returns the proper object, you could define the resulting components as `<bean/>`'s in ColdSpring like this:

```
<bean id="someDAO" factory-bean="someDAOFactory" factory-method="getDAO"/>
```

Remember, you won't define a `class=""` attribute on these `<bean/>` because the bean definition indicates the result of a method call on another bean. Also, you may need to supply arguments to your factory's method - perhaps `someDAOFactory.getDAO()` needs the type of DAO you want e.g. `someDAOFactory.getDAO('person')` returns a `personDAO`. You use the `<constructor-arg/>` tag to pass arguments to a factory method, like so:

```
<bean id="personDAO" factory-bean="someDAOFactory" factory-method="getDAO">
    <constructor-arg name="DAOType">
        <value>person</value>
    </constructor-arg>
</bean>
```

If you use <property/> on a <bean/> defined with factory-bean and factory-method, it will be set on the resulting component just like any other property. Take the following bean definition:

```
<bean id="personDAO" factory-bean="someDAOFactory" factory-method="getDAO">
    <constructor-arg name="DAOType">
        <value>person</value>
    </constructor-arg>
    <property name="DatabaseName">
        <value>MyDatabase</value>
    </property>
</bean>
```

When ColdSpring is asked for the bean named `"personDAO"`, it will first call `getDAO('person')` on the `someDAOFactory`, and then call `setDatabaseName('MyDatabase')` on the component returned from `getDAO('person')`. <property/> will only work if your factory's method is returning a component, as factory-method can be used to define arbitrary values such as strings, structs, etc. For example, if you needed to explicitly pass the returning value of a method call into another component, you can use the factory-bean/method approach like so:

```
<bean id="someComponent" class="type.of.SomeComponent">
    <property name="DatasourceName">
        <bean id="dsn" factory-bean="ConfigService" factory-method="getConfigValue">
<constructor-arg name="ValueName">
                <value>dsn</value>
</constructor-arg>
        </bean>
    </property>
</bean>
```

In the above example, when asked for `someComponent`, ColdSpring would inject the result of calling `ConfigService.getConfigValue('dsn')`

II.VIII Hierarchical Bean Factories

ColdSpring provides a way to link BeanFactories together in a hierarchy, allowing you to define `<bean/>`'s in one BeanFactory and then have a "child" BeanFactory resolve dependencies from its parent. A typical example would be "system-wide" components like a "LoggerService":

```
<beans>
    <bean id="LoggerService" class="some.logger.component"/>
</beans>
```

Say we create our `GlobalBeanFactory` and supply it with the above definitions. We could then create a new BeanFactory (we'll call it `SomeAppBeanFactory`) and supply it with these definitions:

```
<beans>
    <bean id="SomeComponent" class="some.application.component">
        <property name="LoggerService">
<ref bean="LoggerService"/>
```

```
        </property>
    </bean>
</beans>
```

Of course, if you just create `SomeAppBeanFactory` and supply it with the above XML, `SomeComponent` would not get injected with a `LoggerService` because ColdSpring has no idea of a bean named `"LoggerService"`. However, if you call the method `setParent()` on a BeanFactory, and pass in a reference to another BeanFactory, ColdSpring will resolve dependencies that exist in the "parent" BeanFactory:

```
<cfset SomeAppBeanFactory.setParent(GlobalBeanFactory)/>
```

After running the above line of code, `SomeAppBeanFactory` would be able to resolve the `LoggerService <bean/>`. Also, lets say you have several "child" BeanFactories, and one of them is having issues using the system-wide `"LoggerService"`. You could define your own `"LoggerService" <bean/>` in that specific BeanFactory and ColdSpring would use that one instead of the one from the parent.

IV. Aspect Oriented Programming w/ ColdSpring

IV.I. Introduction

Aspect Oriented Programming is a programming model that allows you to think about parts of an application in terms of generalized concerns, as apposed to working with a hierarchy of objects as you would in Object Oriented Programming. Although the concept may seem like a big shift in thinking, it is actually highly complementary to traditional OOP architecture. Using AOP is primarily a process of breaking apart your application into primary business concerns, those that lend themselves to traditional OOP style modeling, and generalized concerns, things like logging, security, and caching, which can be written in a abstracted, reusable manner, and then applied globally across your model with an AOP framework. Concerns like these can be thought of as aspects of your application, functionality that cuts across many of the components in your primary business logic. For this reason it's often said that AOP deals with crosscutting concerns. Working with AOP in the context of an IoC container such as ColdSpring is particularly easy and powerful, as you will primarily be dealing with programming the functionality of your aspects, and use the bean factory configuration file to configure your model components with this added functionality.

IV.II. Concepts

Many new technologies often come with new and sometimes confusing terminology, and AOP is certainly no exception. Below are some of the new terms you will be need to become familiar with, you may want to reference this list as you gain knowledge working with AOP.

Aspects: Aspects are the generalized, crosscutting concerns that you would like to apply to your model components. Your will primarily implement these aspect through the programming of Advice objects which will be joined to your model components through a process called Weaving.

Joinpoints: Joinpoints are areas of your application where you would like to apply your Advice objects to. Joinpoints are an abstract concept in AOP, there are no Joinpoint components that you will use, they are specific places in your model where you would like the functionality in your aspects to occur. For all intents and purposes, Joinpoints are methods in your cfcs where you will be adding functionality.

Pointcuts: Pointcuts are special components that you can configure to identify Joinpoints for you. Pointcuts are configured with the names of methods, which can contain the '*' wildcard character, and are the primary mechanism with which your Aspects are applied to methods in your model component.

Advice: Advice objects are the primary implementation of your Aspects. Working with AOP is mainly a process of adding functionality to existing methods in a cfc. You can think of this as adding advice to those methods.

Advisors: Advisors are special components that you configure to create pointcuts for you, and associate Advice objects with them. Advisors are used during the weaving process to build proxy objects, they are the mechanism for defining the how your Advice components should be applied to target objects.

Target Object: The target object is the cfc that you would like to add functionality to. ColdSpring will create a new object based on this target object that you can use in place of it elsewhere in your model.

Proxy Object: A proxy object is an object that ColdSpring creates to accept method calls intended for your target object, which will then look to see if there are any Advice objects to use before forwarding the method call to your target. The proxy object actually contains all the aspects and pointcuts it needs along with your target object. ColdSpring creates this object for you, which will have the same method signatures and type of your original object, making its use transparent to the rest of your model components.

Weaving: Weaving is the process of generating a Proxy Object, providing it with any Aspects and Pointcuts you have configured for a target object. This is performed by a ProxyBeanFactory, which is configured in your ColdSpring xml configuration file.

Although there are a few confusing new concepts to swallow here, it is important to understand that most of these are just that, concepts. The primary objects that you will be concerning your self with are Advice components, as they are where you will actually be writing the functionality that make up your Aspects, and Advisors, which you will define in your ColdSpring configuration file with rules that it will use to create Pointcuts for your advice. You will also configure ProxyFactoryBeans that will take care take care of building proxy objects for you (through weaving).

IV.III. Advice Types

ColdSpring AOP offers four types of Advice to work with. You create an advice component of one of the following types by extending a ColdSpring.aop component, and implementing a method. This allows ColdSpring to know how to use your component.

BeforeAdvice: BeforeAdvice components will be executed before a method that you have determined to be a Joinpoint. They have access to the method name being called, the component that the call is targeted to, and all the arguments being passed to that method. BeforeAdvice should not alter the method or arguments, as that could cause unexpected results, and possibly hard to debug exceptions.

AfterReturningAdvice: AfterReturningAdvice components are called after a method has completed execution and possibly returned a result. They also have access to the method, target object and arguments to the method call, as well as any returned value. Like BeforeAdvice, best practices would dictate that you should not alter the return value of a method call. If you intend to actually effect method execution in some way, you should be using the more powerful AroundAdvice.

AroundAdvice: AroundAdvice is a far more powerful type of advice than before or after advice, as it gives you complete control over the method execution. ColdSpring AOP implements this type of advice through the use of a component called MethodInterceptor, which in turn uses a MethodInvocation object to yield control over the method call. Although these component names probably sound strange, then are implementations of interfaces developed by a group of Java developers called the AOP Alliance, which consists of developers from the Spring Framework, Nannings Aspects, and AspectJ. These interfaces are used to provide consistency and portability between AOP implementations, and give you the benefit of having a familiarity with the concepts if you find yourself working with one of these frameworks.

ThrowsAdvice: ThrowsAdvice, as the name implies, are called when a method call throws an exception. You have complete access to the exception that occurred and the ability to do whatever you want with it.

IV.III.I BeforeAdvice

BeforeAdvice is a simple type of advice that is used to insert behavior before a method is executed. BeforeAdvice components are automatically inserted before a method call, and don't need to do anything to cause the target method to be invoked. This gives it the advantage of being very easy to use.

To create a before advice component, you extend the pseudo-abstract component `coldspring.aop.BeforeAdvice`. This will enable ColdSpring to recognize your Advice and insert it before a method call on a proxied object. To implement a before advice, you must define the following method in your component:

```
before(method, args, target)
```

Argument	Type	Description
`method`	`coldspring.aop.Method`	The Method object wraps the original method call. `Proceed()` will automatically be called for you on the Method object after your BeforeAdvice executes (to execute the original method call)
`args`	`Struct`	The argument collection that will be passed to the method call in the Method object
`target`	`Component`	The target object of the method call in the Method object (useful if your advice is very generic and needs to inspect the component that it is being applied to)

The `coldspring.aop.Method` component exposes the following methods during within `before(...)`

Method Signature	Return Type	Description
`getMethodName()`	`String`	The name of the original method that's being called.

Example

The following example defines a simple logging advice to run before a method is executed. This Advice uses a `LoggingService`, and will interrogate the args and target arguments to send information about the method call to the `LoggingService`. By separating out the implementation of the actual logging, we gain the ability to easily enable/disable logging, alter the log file location, or switch implementations at will.

```
<cfcomponent name="LoggingBeforeAdvice" extends="coldspring.aop.BeforeAdvice">
```

```
    <!--- setters for dependencies --->
    <cffunction name="setLoggingService" returntype="void" access="public" output="false"
                hint="Dependency: logging service">
        <cfargument name="loggingService" type="net.klondike.service.LoggingService"
                    required="true"/>
        <cfset variables.m_loggingService = arguments.loggingService />
    </cffunction>
    <cffunction name="before" access="public" returntype="any">
        <cfargument name="method" type="coldspring.aop.Method" required="true" />
        <cfargument name="args" type="struct" required="true" />
        <cfargument name="target" type="any" required="true" />
        <cfset var arg = '' />
        <cfset var argString = '' />
        <cfset var objName = getMetadata(arguments.target).name />
        <cfloop collection="#arguments.args#" item="arg">
            <cfif isSimpleValue(arguments.args[arg])>
                <cfif len(argString)>
                    <cfset argString = argString & ', ' />
                </cfif>
                <cfset argString = argString & arg & '=' & arguments.args[arg] >
            </cfif>
        </cfloop>
        <cfset variables.m_loggingService.info("[" & objName & "] "
& method.getMethodName()
            & "(" & argString & ") called!") />
    </cffunction>
</cfcomponent>
```

The ColdSpring bean definitions for your advice would look like this:

```
<!-- logging service -->
<bean id="loggingService" class="net.klondike.component.LoggingService" />
<!-- set up the logging advice -->
<bean id="loggingAdvice" class="net.klondike.aspects.LoggingBeforeAdvice">
    <property name="loggingService">
        <ref bean="loggingService" />
    </property>
</bean>
```

You would then supply your advice to an advisor. The advisor is configured with either a property called `"mappedName"`, which indicated a string pattern for which method names we want our advice applied to, or `"mappedNames"` which can be a comma-separated list. You can use a wildcard character ("*") in either to match multiple methods. In the example here we use mappedNames="*", which will match all methods:

```
<bean id="loggingAdvisor" class="coldspring.aop.support.NamedMethodPointcutAdvisor">
    <property name="advice">
        <ref bean="loggingAdvice" />
    </property>
    <property name="mappedNames">
        <value>*</value>
    </property>
</bean>
```

This advisor can configured to intercept the method calls on any object in the BeanFactory by using ProxyFactoryBean. Seen below, we set up the `CatalogDAO` as normal but we give it an `id="catalogDAOTarget"`. We then create a ProxyFactoryBean with the proper `id="catalogDAO"`. Thus anyone asking ColdSpring for the bean `"catalogDAO"` (or `<bean/>`'s that are wired to `catalogDAO`) get our ProxyFactoryBean, however it looks and smells like a regular `catalogDAO` (but in actuality has our advisor snooping method calls and applying advice as needed):

```
<!-- set up a proxy for the dao -->
<bean id="catalogDAOTarget"
        class="net.klondike.component.catalogDAO">
    <property name="dsn">
        <value>klondike</value>
    </property>
</bean>
<bean id="catalogDAO" class="coldspring.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref bean="catalogDAOTarget" />
    </property>
    <property name="interceptorNames">
        <list>
            <value>loggingAdvisor</value>
        </list>
    </property>
</bean>
```

IV.III.II AfterReturningAdvice

AfterReturningAdvice is used to insert behavior after a method has executed. These advice types are automatically inserted after a method is called, and although this type of advice can read any return value from the called method, it does not need to do anything to ensure normal method execution. As in BeforeAdvice, this also gives it an advantage of being simple to use.
To create an after returning advice component, you extend the pseudo-abstract component `coldspring.aop.AfterReturningAdvice`. This enables ColdSpring to recognize the advice and insert it after the method has executed. To implement an after retuning advice, you must define the following method in your component:

```
afterReturning(returnVal, method, args, target)
```

Argument	Type	Description
`returnVal`	`any`	If the method you're advising returned a value, it will be defined in the arguments scope as `returnVal`. If your method did not return a value `returnVal` will not be defined.
`method`	`coldspring.aop.Method`	The Method object wraps the original method call. `Proceed()` will automatically be called for you on the Method object after your BeforeAdvice executes (to execute the original method call)
`args`	`Struct`	The argument collection that will be passed to the method call in the Method object
`target`	`Component`	The target object of the method call in the Method object (useful if your advice is very generic and needs to inspect the component that it is being applied to)

The `coldspring.aop.Method` component exposes the following methods during within `afterReturning(...)`

Method Signature	Return Type	Description
`getMethodName()`	`String`	The name of the original method that's being called.

Example

The following example defines a simple logging advice to run after a method is executed. Again, this Advice uses a `LoggingService`, and will interrogate the returnVal, args and target arguments to send information about the method call and return value to the `LoggingService`.

```
<cfcomponent name="LoggingAfterAdvice" extends="coldspring.aop.AfterReturningAdvice">
     <!--- setters for dependencies --->
     <cffunction name="setLoggingService" returntype="void" access="public" output="false"
                    hint="Dependency: logging service">
          <cfargument name="loggingService" type="net.klondike.service.LoggingService"
                         required="true"/>
          <cfset variables.m_loggingService = arguments.loggingService />
     </cffunction>
     <cffunction name="afterReturning" access="public" returntype="any">
          <cfargument name="returnVal" type="any" required="false" />
          <cfargument name="method" type="coldspring.aop.Method" required="true" />
          <cfargument name="args" type="struct" required="true" />
          <cfargument name="target" type="any" required="true" />
          <cfset var arg = '' />
          <cfset var argString = '' />
          <cfset var objName = getMetadata(arguments.target).name />
          <cfloop collection="#arguments.args#" item="arg">
               <cfif isSimpleValue(arguments.args[arg])>
                    <cfif len(argString)>
                         <cfset argString = argString & ', ' />
                    </cfif>
                    <cfset argString = argString & arg & '=' & arguments.args[arg] >
               </cfif>
          </cfloop>
          <cfif StructKeyExists(arguments,"returnVal")
and isSimpleValue(arguments.returnVal)>
               <cfset variables.m_loggingService.info("[" & objName & "] "
                         & method.getMethodName()
                         & "(" & argString & ") called! Returning: " & arguments.returnVal) />
          <cfelse>
               <cfset variables.m_loggingService.info("[" & objName & "] "
                         & method.getMethodName()
                         & "(" & argString & ") called!") />
          </cfif>

     </cffunction>
```

```
</cfcomponent>
```

The ColdSpring bean definitions for your advice would look like this:

```
<!-- logging service -->
<bean id="loggingService" class="net.klondike.component.LoggingService" />
<!-- set up the logging advice -->
<bean id="loggingAdvice" class="net.klondike.aspects.LoggingAfterAdvice">
    <property name="loggingService">
        <ref bean="loggingService" />
    </property>
</bean>
```

You would then supply your advice to an advisor. The advisor is configured with either a property called `"mappedName"`, which indicated a string pattern for which method names we want our advice applied to, or `"mappedNames"` which can be a comma-separated list. You can use a wildcard character ("*") in either to match multiple methods. In the example here we use mappedNames="*", which will match all methods:

```
<bean id="loggingAdvisor" class="coldspring.aop.support.NamedMethodPointcutAdvisor">
    <property name="advice">
        <ref bean="loggingAdvice" />
    </property>
    <property name="mappedNames">
        <value>*</value>
    </property>
</bean>
```

Same as before, this advisor can configured to intercept the method calls on any object in the BeanFactory by using ProxyFactoryBean. Seen below, we set up the `CatalogDAO` as normal but we give it an `id="catalogDAOTarget"`. We then create a ProxyFactoryBean with the proper `id="catalogDAO"`. Thus anyone asking ColdSpring for the bean `"catalogDAO"` (or `<bean/>`'s that are wired to `catalogDAO`) get our ProxyFactoryBean, however it looks and smells like a regular `catalogDAO` (but in actuality has our advisor snooping returning method calls and applying advice as needed):

```
<!-- set up a proxy for the dao -->
<bean id="catalogDAOTarget"
        class="net.klondike.component.catalogDAO">
    <property name="dsn">
```

```
        <value>klondike</value>
    </property>
</bean>
<bean id="catalogDAO" class="coldspring.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref bean="catalogDAOTarget" />
    </property>
    <property name="interceptorNames">
        <list>
            <value>loggingAdvisor</value>
        </list>
    </property>
</bean>
```

IV.III.III AroundAdvice aka MethodInterceptor

Where BeforeAdvice and AfterAdvice give you're the ability to inject functionality either before or after method execution, AroundAdvice gives you both total control over and responsibility for method invocation. In around advice you are given a command object called a `MethodInvocation` which not only gives you access to the method name, target object, and arguments to the method call, but also access to the `proceed()` method of the `MethodInvocation` command object containing the method call. It is actually your responsibility to call `proceed()`, or the method call you are advising will not execute at all. For this reason, AroundAdvice is considerably more powerful than before or after advice.

AroundAdvice is implemented through a component called `MethodInterceptor`, as it gives you the ability to do just that, intercept a method call, and perform any operation necessary. When ColdSpring finds a `MethodInterceptor` for a method, it will relinquish all control to that object, and provide it with a `MethodInvocation` object, which will be used to interrogate the method call, and then trigger it's operation. Because Advice components can be chained together, it is worth noting that the object that receives the call to `proceed()` may not actually be the target method, it may be another `MethodInterceptor`. Although the internal processing may seem quite complex, the `MethodInvocation` object is built with all the necessary information to make this transparent to the user. It's important to realize that the `MethodInterceptor` is responsible for returning whatever needs to be returned from the actual method call - perhaps it's just whatever is returned from `proceed()`, or a completely different value all together.

To create a `MethodInterceptor`, you will extend the pseudo-abstract component `coldspring.aop.MethodInterceptor`. This will enable ColdSpring to recognize your advice and add it to the Advice chain used internally by the `MethodInvocation` object. To implement a `MethodInterceptor`, you must define the following method in your component:

```
invokeMethod(methodInvocation)
```

Argument	**Type**	**Description**
`methodInvocation`	`coldspring.aop.MethodInvocation`	The `MethodInvocation` object (described in detail below) gives you access to the method name, target object, and argument collection, as well as control over method execution through the `proceed()` method

The `methodInvocation` component exposes the following methods:

Method	**Return Type**	**Description**
`proceed()`	`void`	Invokes the next MethodInterceptor or the target method.
`getMethod()`	`coldspring.aop.Method`	Returnes the Method object containing the target method.
`getArguments()`	`struct`	Returns the argument collection from the original method call.
`getTarget()`	`Component`	Returnes the target object for the method call

The `coldspring.aop.Method` component exposes the following methods:

Method Signature	**Return Type**	**Description**
`getMethodName()`	`String`	The name of the original method that's being called.

Example

The following example defines a simple logging advice to run as method interceptor. As you can see, the advice calls proceed() on the MethodInvocation object, and inspect the return value for logging purposes.

```
<cfcomponent name="LoggingAroundAdvice" extends="coldspring.aop.MethodInterceptor">
     <!--- setters for dependencies --->
     <cffunction name="setLoggingService" returntype="void" access="public" output="false"
                    hint="Dependency: logging service">
          <cfargument name="loggingService" type="net.klondike.service.LoggingService"
                         required="true"/>
          <cfset variables.m_loggingService = arguments.loggingService />
     </cffunction>
     <cffunction name="invokeMethod" access="public" returntype="any">
          <cfargument name="methodInvocation" type="coldspring.aop.MethodInvocation"
                         required="false" />

          <cfset var arg = '' />
          <cfset var argString = '' />

          <cfset var args = arguments.methodInvocation.getArguments() />
          <cfset var methodName = arguments.methodInvocation.getMethod().getMethodName() />
          <cfset var objName = getMetadata(arguments.methodInvocation.getTarget()).name />

          <cfloop collection="#args#" item="arg">
               <cfif isSimpleValue(args[arg])>
                    <cfif len(argString)>
                         <cfset argString = argString & ', ' />
                    </cfif>
                    <cfset argString = argString & arg & '=' & args[arg] >
               </cfif>
          </cfloop>

          <cfset rtn = arguments.methodInvocation.proceed() />

          <cfif isDefined('rtn') and isSimpleValue('rtn')>
               <cfset variables.m_loggingService.info("[" & objName & "] "
                    & methodName & "(" & argString & ") complete! Returning: " & 'rtn') />
          <cfelse>
<cfset variables.m_loggingService.info("[" & objName & "] "
                    & methodName & "(" & argString & ") complete!") />
```

```
        </cfif>

        <cfif isDefined('rtn')>
            <cfreturn rtn />
        </cfif>

    </cffunction>
</cfcomponent>
```

The ColdSpring bean definitions for your MethodInterceptor would look like this:

```
<!-- logging service -->
<bean id="loggingService" class="net.klondike.component.LoggingService" />
<!-- set up the logging advice -->
<bean id="loggingAdvice" class="net.klondike.aspects.LoggingAroundAdvice">
    <property name="loggingService">
        <ref bean="loggingService" />
    </property>
</bean>
```

You would then supply your advice to an advisor. The advisor is configured with either a property called "mappedName", which indicated a string pattern for which method names we want our advice applied to, or "mappedNames" which can be a comma-separated list. You can use a wildcard character ("*") in either to match multiple methods. In the example here we use mappedNames="*", which will match all methods:

```
<bean id="loggingAdvisor" class="coldspring.aop.support.NamedMethodPointcutAdvisor">
    <property name="advice">
        <ref bean="loggingAdvice" />
    </property>
    <property name="mappedNames">
        <value>*</value>
    </property>
</bean>
```

Same as before, this advisor can configured to intercept the method calls on any object in the BeanFactory by using ProxyFactoryBean. Seen below, we set up the CatalogDAO as normal but we give it an id="catalogDAOTarget". We then create a ProxyFactoryBean with the proper id="catalogDAO". Thus anyone asking ColdSpring for the bean "catalogDAO" (or <bean/>'s that are wired to catalogDAO) get our ProxyFactoryBean, however it looks and smells like a regular catalogDAO (but in actuality has our advisor snooping method calls and applying our `MethodInterceptor` as needed):

```
<!-- set up a proxy for the dao -->
<bean id="catalogDAOTarget"
        class="net.klondike.component.catalogDAO">
    <property name="dsn">
        <value>klondike</value>
    </property>
</bean>
<bean id="catalogDAO" class="coldspring.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref bean="catalogDAOTarget" />
    </property>
    <property name="interceptorNames">
        <list>
            <value>loggingAdvisor</value>
        </list>
    </property>
</bean>
```

IV.III.IV ThrowsAdvice

ThrowsAdvice is another simple type of advice that is used to insert behavior when method execution throws an exception. ColdSpring's AOP framework will gather the details of the exception and pass them to your advice. It's important to note that you cannot suppress the exception unless you throw a different one (if you want that level of control you may want to use AroundAdvice).

To create a ThrowsAdvice component, you extend the pseudo-abstract component `coldspring.aop.ThrowsAdvice`. This will enable ColdSpring to recognize your Advice and insert it when a method call on a proxied object throws an exception. To implement a ThrowsAdvice, you can do one of two things:

- implement an afterThrowing() or afterThrowingAny() method, which coldspring will route all exceptions to

- implement methods for the exception types you are interested in - for instance if you wanted to catch exceptions with a type="SqlException", you would implement a afterThrowingSqlException() method.

It's up to you, because you could always interrogate the exception type in a afterThrowing() method and then call the appropriate method yourself, ColdSpring just has some extra machinery to automate the mapping of exception types to methods in your advice.

The syntax for an afterThrowing*() method(s) are as follows

```
afterThrowing(method, args, target, exception)
```

Argument	**Type**	**Description**
`method`	`coldspring.aop.Method`	The Method object wraps the original method call. `Proceed()` will automatically be called for you on the Method object after your BeforeAdvice executes (to execute the original method call)
`args`	`Struct`	The argument collection that will be passed to the method call in the Method object
`target`	`Component`	The target object of the method call in the Method object (useful if your advice is very generic and needs to inspect the component that it is being applied to)
`exception`	`coldspring.aop.Exception`	A component containing the details of the exception that occurred (see below)

The `coldspring.aop.Exception` component exposes the following methods during `afterThrowing*(...)`

Method Signature	**Return Type**	**Description**
`getType()`	`string`	returns the type of the exception
`getMessage()`	`string`	returns the message within the exception
`getDetail()`	`string`	returns the detail within the exception

`getTagContext()`	`array`	returns an array of tags within which the exception occured
`getExtendedInfo()`	`string`	returns extended info, if present
`getNativeErrorCode()`	`string`	returns a native database error code, if present
`getSqlState()`	`string`	returns sqlstate, if present
`getSql()`	`string`	returns the sql attempting to execute, if present
`getQueryError()`	`string`	returns the query error, if present
`getWhere()`	`string`	returns the where clause of the query attempting to execute, if present
`getErrNumber()`	`string`	returns the database-specific error number, if present
`getMissingFileName()`	`string`	returns the name of a missing file, if present (exists when a template can't be found)
`getLockOperation()`	`string`	when getType() == "lock", returns operation that failed (Timeout, Create Mutex, or Unknown, if present
`getLockName()`	`string`	when getType() == "lock", returns name of affected lock (if the lock is unnamed, the value is "anonymous").
`getErrorCode()`	`string`	returns errorcode of exception, when present
`getBaseException()`	cf exception (aka `cfcatch`)	Returns the actual exception

The `coldspring.aop.Method` component exposes the following methods during `afterThrowing*(...)`

Method Signature	**Return Type**	**Description**
`getMethodName()`	`String`	The name of the original method that's being called.

Example

The following example defines a simple logging advice to run when a method throws an exception. This Advice uses a `LoggingService`, and will interrogate the args, target, and exceptions arguments to send information about the method exception to the `LoggingService`. By separating out the implementation of the actual logging, we gain the ability to easily enable/disable logging, alter the log file location, or switch implementations at will.

```
<cfcomponent name="LoggingThrowsAdvice" extends="coldspring.aop.ThrowsAdvice">
     <!--- setters for dependencies --->
     <cffunction name="setLoggingService" returntype="void" access="public" output="false"
                    hint="Dependency: logging service">
          <cfargument name="loggingService" type="net.klondike.service.LoggingService"
                         required="true"/>
          <cfset variables.m_loggingService = arguments.loggingService />
     </cffunction>

     <cffunction name="afterThrowing" access="public" returntype="any">
          <cfargument name="method" type="coldspring.aop.Method" required="true" />
          <cfargument name="args" type="struct" required="true" />
          <cfargument name="target" type="any" required="true" />
          <cfargument name="exception" type="coldspring.aop.Method" required="true" />

          <cfset var arg = '' />
          <cfset var argString = '' />
          <cfset var objName = getMetadata(arguments.target).name />
          <cfloop collection="#arguments.args#" item="arg">
               <cfif isSimpleValue(arguments.args[arg])>
                    <cfif len(argString)>
                         <cfset argString = argString & ', ' />
                    </cfif>
                    <cfset argString = argString & arg & '=' & arguments.args[arg] >
               </cfif>
          </cfloop>
          <cfset variables.m_loggingService.info("[" & objName & "] "
               & method.getMethodName()
               & "(" & argString & ") threw an exception:
               & arguments.exception.getType()
               & "(" & arguments.exception.getMessage()
               & "|" & arguments.exception.getDetail() & ")" />
     </cffunction>
</cfcomponent>
```

The ColdSpring bean definitions for your advice would look like this:

```
<!-- logging service -->
<bean id="loggingService" class="net.klondike.component.LoggingService" />
<!-- set up the logging advice -->
<bean id="loggingAdvice" class="net.klondike.aspects.LoggingThrowsAdvice">
    <property name="loggingService">
```

```
        <ref bean="loggingService" />
    </property>
</bean>
```

You would then supply your advice to an advisor. The advisor is configured with either a property called `"mappedName"`, which indicated a string pattern for which method names we want our advice applied to, or `"mappedNames"` which can be a comma-separated list. You can use a wildcard character ("*") in either to match multiple methods. In the example here we use mappedNames="*", which will match all methods:

```
<bean id="loggingAdvisor" class="coldspring.aop.support.NamedMethodPointcutAdvisor">
    <property name="advice">
        <ref bean="loggingAdvice" />
    </property>
    <property name="mappedNames">
        <value>*</value>
    </property>
</bean>
```

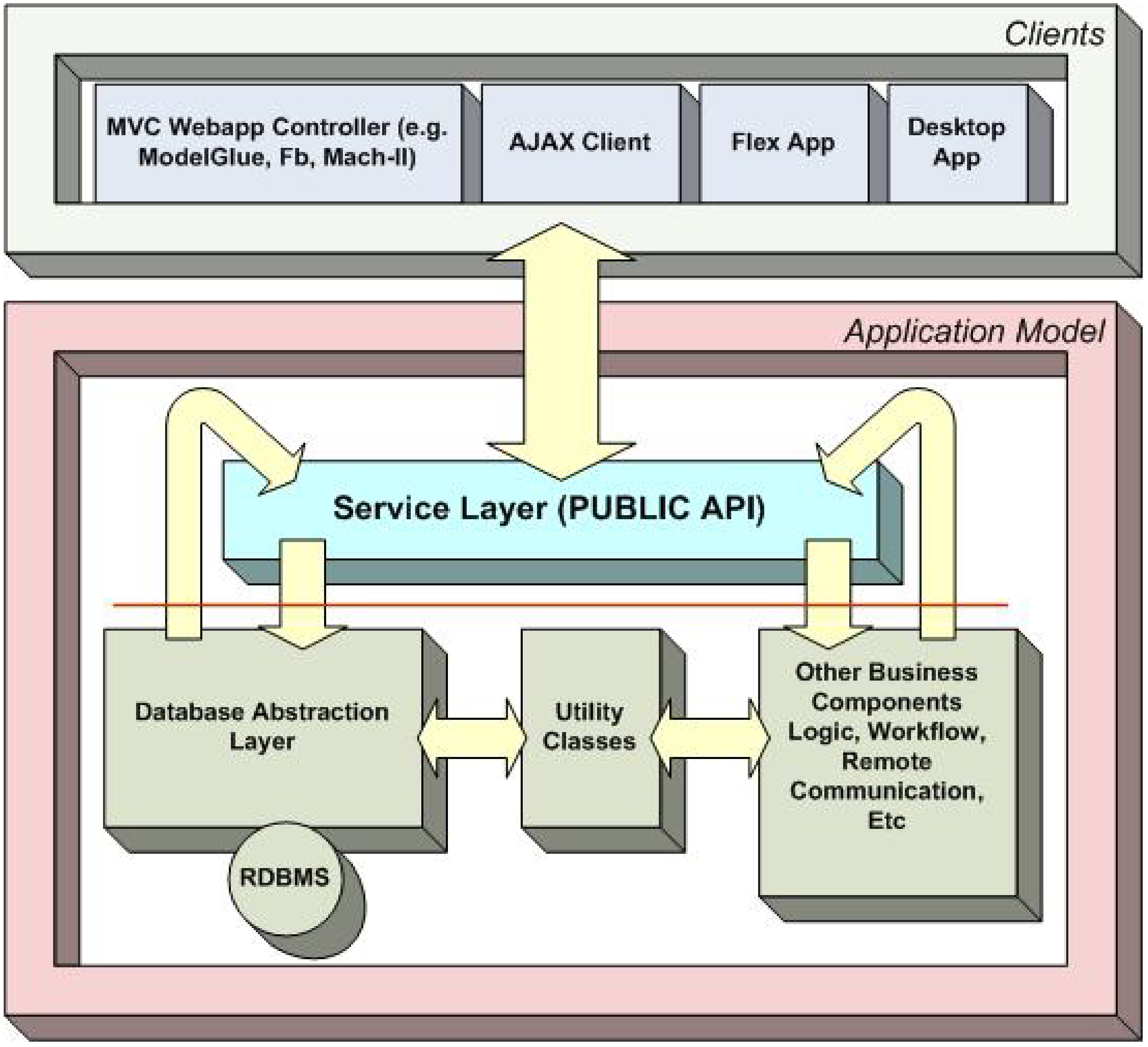
This advisor can configured to intercept the method calls on any object in the BeanFactory by using ProxyFactoryBean. Seen below, we set up the `CatalogDAO` as normal but we give it an `id="catalogDAOTarget"`. We then create a ProxyFactoryBean with the proper `id="catalogDAO"`. Thus anyone asking ColdSpring for the bean `"catalogDAO"` (or `<bean/>`'s that are wired to `catalogDAO`) get our ProxyFactoryBean, however it looks and smells like a regular `catalogDAO` (but in actuality has our advisor snooping method calls and applying advice as needed (when exceptions occur)):

```
<!-- set up a proxy for the dao -->
<bean id="catalogDAOTarget"
        class="net.klondike.component.catalogDAO">
    <property name="dsn">
        <value>klondike</value>
    </property>
</bean>
<bean id="catalogDAO" class="coldspring.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref bean="catalogDAOTarget" />
    </property>
    <property name="interceptorNames">
        <list>
            <value>loggingAdvisor</value>
        </list>
    </property>
</bean>
```

III. Developing w/ ColdSpring

III.I Service Layers and ColdSpring

ColdSpring was designed to work exceptionally well with a piece of application architecture known as a "service-layer". What this means is that the functionality comprised within many of the application's components is separated into logical units and each is abstracted behind a clean interface (interface as in api). This is interface is often called a "service". In some applications you might have a few components that make up one logical unit of functionality… a DAO to fetch and store object instances in a database, maybe a gateway for aggregating multiple objects of that same type into recordsets. The idea of a service layer is that you group that functionality together so that other pieces of the application that depend on those components can speak to them through a clean, well documented api (the service). You'll do your best to define that api as early as possible, because the less it changes the easier your life will be. The abstraction a service layer provides also makes it a lot easier to manage your dependencies, which is where ColdSpring comes in. The diagram below shows the importance of the service layer (which could also be referred to as your "public API". Not only do "clients" such as web-application controller layers, ajax, and flex depend on the service layer's api, but also components within your model. This is why having the dependency resolution features of ColdSpring at your disposal make it easy to provide *any* component in your application's model with *any* service api.


Clients
MVC Webapp Controller (e.g. ModelGlue, Fb, Mach-II)
AJAX Client
Flex App
Desktop App
Application Model
Service Layer (PUBLIC API)
Database Abstraction Layer
RDBMS
Utility Classes
Other Business Components Logic, Workflow, Remote Communication, Etc

Even though CFML provides many rich abstractions of complicated programming tasks as simple tags, it still may be a good idea to put them behind a service layer. Take CFMAIL, for example. Sure you could sprinkle email notifications throughout an application by using CFMAIL, but the day requirements change you'll be happier if you put it behind a service (we'll call it our "NotificationService"). So say that you've been given the chore of making sure that the application sends out SMS messages as well as emails to anyone who's listed as having SMS. Well, if you used CFMAIL everywhere, you'd now have to go through and add this functionality, which could end up being a lot of code (and a lot of duplication). Alternatively, if you have a NotificationService that is used everywhere CFMAIL was supposed to be, you could make your changes in one place and the rest of the application would never even know about it. The problem is that before your application depended on CFMAIL, which is simply "available" from anywhere within CFML code. Now we need to provide the same level of ubiquity with our NotificationService, and that's not as easy. However, with ColdSpring, bringing the NotificationService into a component is an easy two step process:

1. Provide a way for the NotificationService to be supplied to the component that needs it. This is done by either an argument to its init method that is named NotificationService and of the same CFC type as the NotificationService, or you provide a public setNotificationService method, with one argument, with the same name and type just like the constructor argument.

2. Describe the dependency in ColdSpring's bean definition or make sure `autowire` is set to `"byName"` or `"byType"`. Without autowiring, the bean definitions would look like this:

    1. Define your NotificationService
       `<bean id="NotificationService" class="myApp.model.NotificationService"/>`

    2. Then define your component that needs the NotificationService, and pass in the reference via constructor-arg or property (property is shown)

```
        <bean id="ComponentThatNeedsNotificationService"
        class="myApp.model.ComponentThatNeedsNotificationService">
<property name="NotificationService">
                <ref bean="NotificationService"/>
</property>
        </bean>
```

The setter method within "ComponentThatNeedsNotificationService" would look like this:

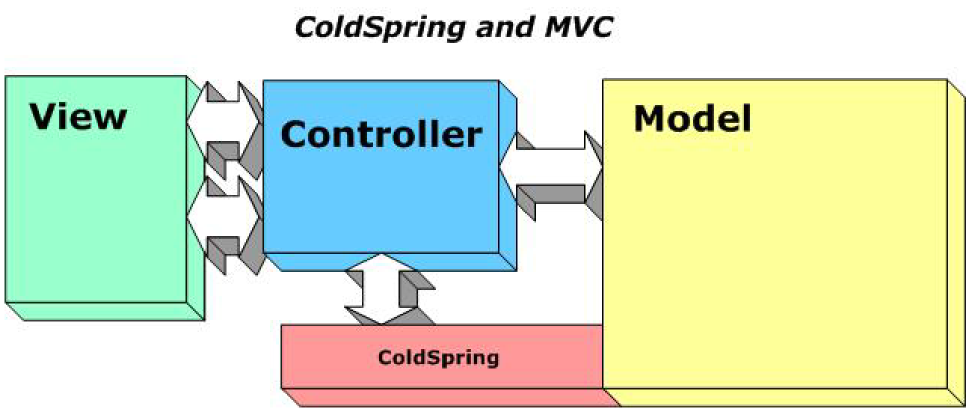
```
<cffunction name="setNotificationService" returntype="void" output="false"
hint="Dependency: Notifcation Service">
```

```
    <cfargument name="NotificationSerivce" type="myApp.model.NotificationService"
         required="true"/>
    <cfset variables.notificationService = arguments.notificationService/>
</cffunction>
```

The `<cfset variables.notificationService = arguments.notificationService/>` line makes sure `"ComponentThatNeedsNotificationService"` will retain a reference to the notification service so that it can be used until overwritten or `"ComponentThatNeedsNotificationService"` is destroyed. Thus anywhere `"ComponentThatNeedsNotificationService"` needs to send a notification it simply says something like: `<cfset variables.notificationService.send(…) />` instead of using `<CFMAIL/>`, and you can be happy that all Notfications are passing through one concise component in your application.

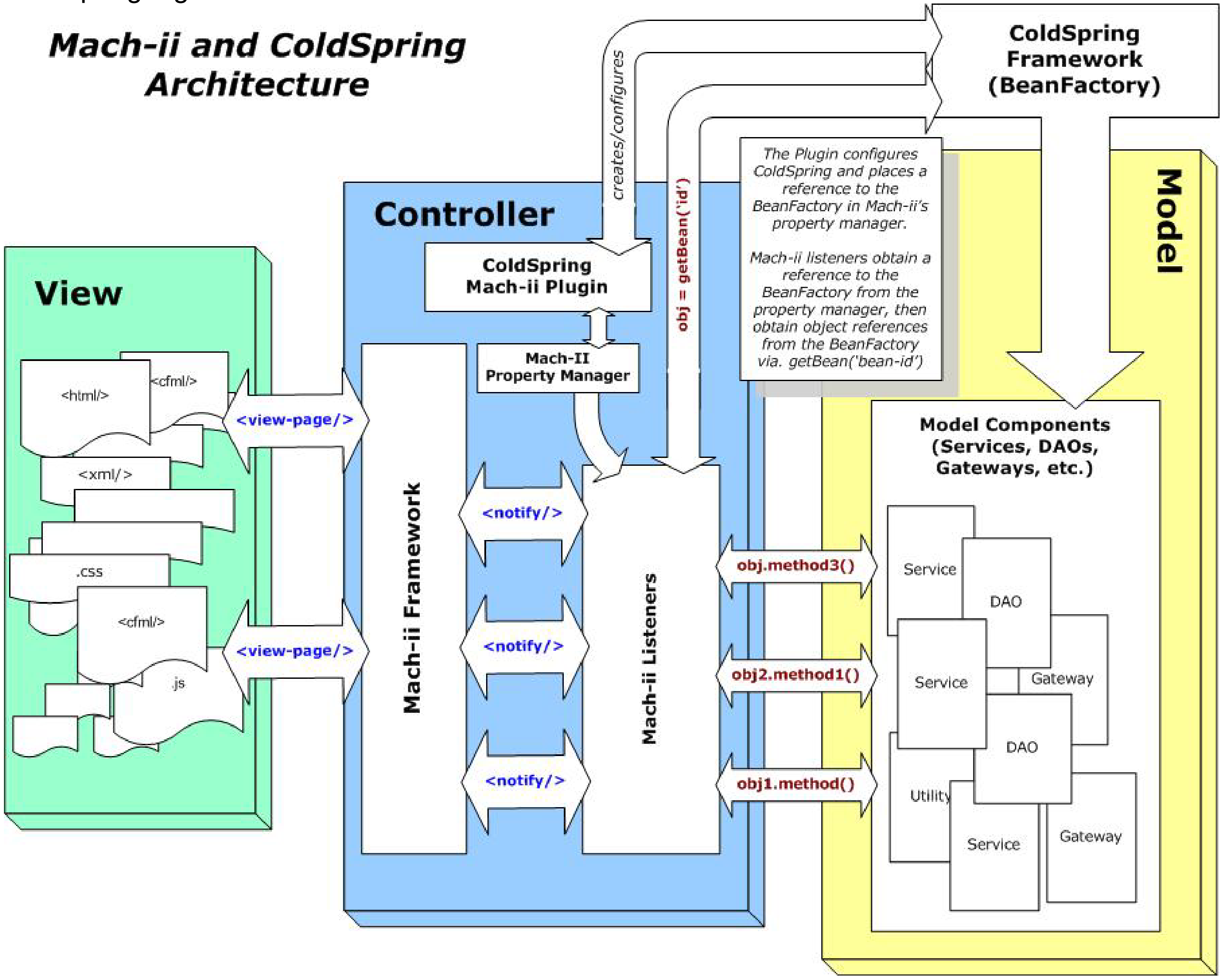
III.II ColdSpring and MVC frameworks

ColdSpring was also developed to fit in well with existing MVC (Model View Controller) frameworks, such as Fusebox 4 and Mach-II. To use ColdSpring with one of these frameworks, it's important to understand the "big picture", as in where ColdSpring sits in relation to the rest of the application. The following diagram illustrates ColdSpring's relationship in a MVC web application:



Why does the Controller layer need to communicate with ColdSpring? In some cases, it may only be to retrieve objects (beans) from the ColdSpring bean factory. In others, the Controller may handle setting up and configuring the bean factory. Either way, it's important to note that once the controller obtains a reference to a bean, it does not communicate "thru" ColdSpring.

For those who use the Mach-II framework for a controller, ColdSpring ships with a Mach-II plugin (coldspring.machii.coldspringPlugin.cfc) that automates the creation of the bean factory. The following diagram details MachII and ColdSpring together:


Mach-ii and ColdSpring Architecture
ColdSpring Framework (BeanFactory)
creates/configures
obj = getBean('id')
The Plugin configures ColdSpring and places a reference to the BeanFactory in Mach-ii's property manager.
Mach-ii listeners obtain a reference to the BeanFactory from the property manager, then obtain object references from the BeanFactory via. getBean('bean-id')
Model
Controller
ColdSpring Mach-ii Plugin
Mach-II Property Manager
View
<html/>
<cfml/>
<xml/>
.css
<cfml/>
.js
<view-page/>
<view-page/>
Mach-ii Framework
<notify/>
<notify/>
<notify/>
Mach-ii Listeners
obj.method3()
obj2.method1()
obj1.method()
Model Components (Services, DAOs, Gateways, etc.)
Service
DAO
Service
Gateway
DAO
Utility
Service
Gateway

ColdSpring bean factories can be used completely transparently in Mach ii applications via the `coldspring.machii.ColdspringPlugin`. The Mach ii plugin handles initialization and storage of the ColdSpring bean factory, as well as 'auto-matically' wiring up all of your Mach ii plugins, filters, and listeners with any ColdSpring managed components they may require. All that is necessary is really to define the plugin in the plugins section of your Mach ii config file, preferably as the first plugin, so it has access to all subsequent plugins for auto-wiring. The plugin also provides relatively fine grained control over its behavior through configuration parameters.

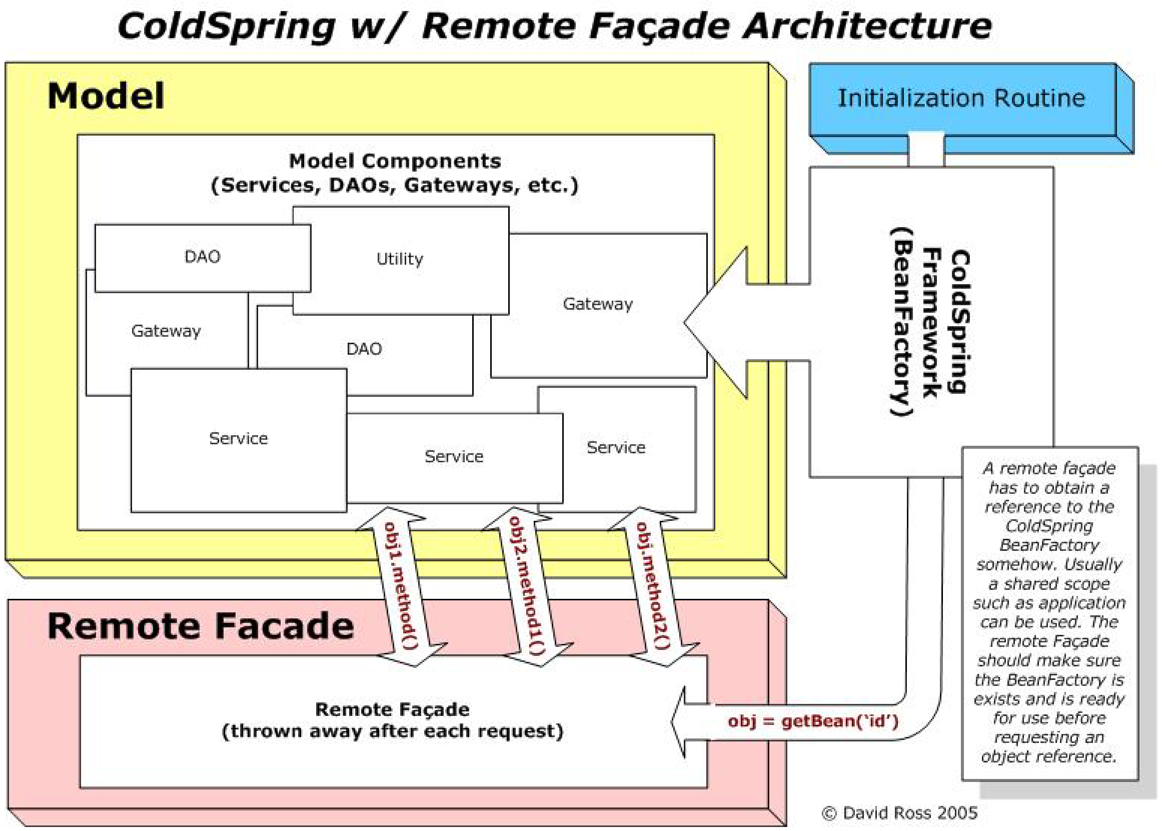
The following parameters are available:

Parameter	**Values**	**Description**
`configFilePropertyName`	any string	The name of the ColdSpring configuration file, defined in the Mach ii properties section
`configFilePathIsRelative`	true/false	Defines whether the file is relative to the application index.cfm file
`resolveMachiiDependencies`	true/false	When set to true all Mach II listeners, filters, and plugins will be auto-wired with ColdSpring managed components, this is the preferred behavior
`placeFactoryInApplicationScope`	true/false	When set to true the ColdSpring bean factory will be placed in Application scope, under the key defined in `localBeanFactoryKey` if it is defined. The bean factory is always added to the propertyManager in the same key
`localBeanFactoryKey`	any string	Defines the key that the bean factory is saved under in the propertyManager, and possibly Application scope. If this parameter is not set, the default bean factory key will be used
`parentBeanFactoryKey`	any string	When working with sub-apps in Mach II, each application's bean factory can set a parent's bean factory as it's own parent. If defined, this string should be the `localBeanFactoryKey` of the parent you wish to set.

III.III ColdSpring and Remoting

III.III.I Remote Facades

ColdSpring also provides a good foundation for exposing your application model to remote method calls. Currently, the primary way to do this is to write remote facades, which expose ColdSpring beans to remote calls by containing methods with `access="remote"`.


ColdSpring w/ Remote Façade Architecture
Model
Model Components (Services, DAOs, Gateways, etc.)
DAO
Utility
Gateway
Gateway
DAO
Service
Service
Service
Initialization Routine
ColdSpring Framework (BeanFactory)
obj1.method()
obj2.method1()
obj.method2()
Remote Facade
Remote Façade (thrown away after each request)
obj = getBean('id')
A remote façade has to obtain a reference to the ColdSpring BeanFactory somehow. Usually a shared scope such as application can be used. The remote Façade should make sure the BeanFactory is exists and is ready for use before requesting an object reference.
© David Ross 2005

There is a good example of a Remote Facade in the "FeedViewer" example application (located within the /examples/ directory within the ColdSpring distribution).

III.III.II Using AOP to create remote proxies

You can also use ColdSpring's AOP framework to automatically create Remote Facade(s) for your components. As of the 1.0 Release, this approach will only work on CFMX 7 and higher. We are working on support for MX6.1 as well.

To use this functionality, first gain some understanding of ColdSpring's AOP framework. The overall approach is this:

- Define your components normally as `<bean/>`'s
- Create a new `<bean/>` using the `coldspring.aop.framework.RemoteFactoryBean` class.
    - In the `RemoteFactoryBean`'s definition, we must define the following properties:
        - `target`, the actual `<bean/>` we are creating a remote interface cfc for
        - `serviceName`, the name of the resulting remote inteface cfc
        - `absolutePath`, the filesystem location where the remote interface cfc should be placed **or** `relativePath`, a path relative from your webroot
        - `remoteMethodNames`, a matching pattern for which methods in our target component we want to remote proxy.

A simple example definition would be:

```
<bean id="someComponent" class="type.of.SomeComponent"/>

<bean id="someComponent_Remote" class="coldspring.aop.framework.RemoteFactoryBean">
    <property name="target">
        <ref bean="someComponent" />
    </property>
    <property name="serviceName">
        <value>RemoteCatalogService</value>
    </property>
    <property name="relativePath">
        <value>/remote/</value>
    </property>
    <property name="remoteMethodNames">
        <value>get*</value>
    </property>
</bean>
```

In order for this to work, we have to tell our remote proxy to actually create itself, so you can do one of two things.

1) You can call `getBean('someComponent_Remote')`on the BeanFactory and ColdSpring will create the proxy for you (you would see a `remoteCatalogService.cfc` in the /remote/ directory on your server. So you would want to add this call during application startup.

2) You can use the `coldspring.aop.framework.RemoteFactoryBean` api to control when the remote proxy is created. First you must obtain the RemoteFactoryBean itself from ColdSpring, and this is done by preceding the bean name with a "&", meaning, `getBean("&someComponent_Remote")` would return your `RemoteFactoryBean`, which exposes the following api for you to use:

Method	**Usage**
`createRemoteProxy()`	Creates the remote proxy
`destroyRemoteProxy()`	Removes the remote proxy
`isConstructed()`	Returns true/false as to whether the proxy has been created.

III.III.III Automatic CFC to ActionScript object conversion

ColdSpring also ships with a few utility classes which can be to automatically marshall and unmarshall between CFCs and ActionScript objects. These classes are the `coldspring.remoting.flash.FlashUtilityService` and `coldspring.remoting.flash.FlashMappings`. The FlashUtilityService uses the FlashMappings component to accomplish this task. You would configure them like so:

```
<bean id="flashMappings" class="coldspring.remoting.flash.FlashMappings">
    <constructor-arg name="mappings">
        <list>
<map>
                <entry key="cfcType">
                    <value>some.type.of.Component</value>
                </entry>
                <entry key="asType">
                    <value>some.actionscript.Type</value
                </entry>
</map>
        </list>
    </constructor-arg>
</bean>
```

```
<bean id="flashUtilityService" class="coldspring.remoting.flash.FlashUtilityService">
    <property name="flashMappings">
        <ref bean="flashMappings"/>
    </property>
</bean>
```

So, you simply provide the mappings as a list of maps (array of structs), each containing both a cfcType and a corresponding actionScript type. In this case any `some.type.of.Component` would be converted to the ActionScript *some.actionscript.Type* class.

You can use the FlashUtilityService by itself, but if you supply a RemoteProxyFactory with it, ColdSpring will automatically use AOP to apply the flashUtilityService to your remote method calls (see III.III.II Using AOP to create remote proxies):

```
<bean id="someComponent" class="type.of.SomeComponent"/>

<bean id="someComponent_Remote" class="coldspring.aop.framework.RemoteFactoryBean">
    <property name="target">
        <ref bean="someComponent" />
    </property>
    <property name="serviceName">
        <value>RemoteCatalogService</value>
    </property>
    <property name="relativePath">
        <value>/remote/</value>
    </property>
    <property name="remoteMethodNames">
        <value>get*</value>
    </property>
    <property name="flashUtilityService">
        <ref bean="flashUtilityService" />
    </property>
</bean>
```