ColdSpring Reference v0.01

David Ross

I: Introduction

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.
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 ColdSpring version change? Actually, it **does not!** 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, why is the dependency-injection approach better?

- 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 and 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 spec to resolve its dependencies (especially public properties exposed via a setter-method). All the above line says is "hey, ColdSpring… register `myApp.components.ShoppingCartManager` under the name `ShoppingCartManager`. This 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 the bean it will receive a new instance.

Your bean definition 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), 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 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 tax calculator into our `ShoppingCartManager`. This is simpler than you think – we have to define the TaxCalculator first (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`, making your code more reusable, testable, and maintainable.

II: Beans + BeanFactory reference guide

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).

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 I'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, a 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 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`	If implemented would cause 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.	Planned (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="argName"/>`
   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 bean=""/>`	`<ref bean="myBeanId"/>`	Used to pass in a reference to another bean defined within the BeanFactory.

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

`<list/>`	`<list>`
	`        <value>5</value>`
	`        <ref id="barBean"/>`
	`</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"/>
```

III. Misc. Development 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.

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:
    a. Define your NotificationService
    ```
    <bean id=”NotificationService”
          class=”myApp.model.NotificationService”/>
    ```
    b. 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:

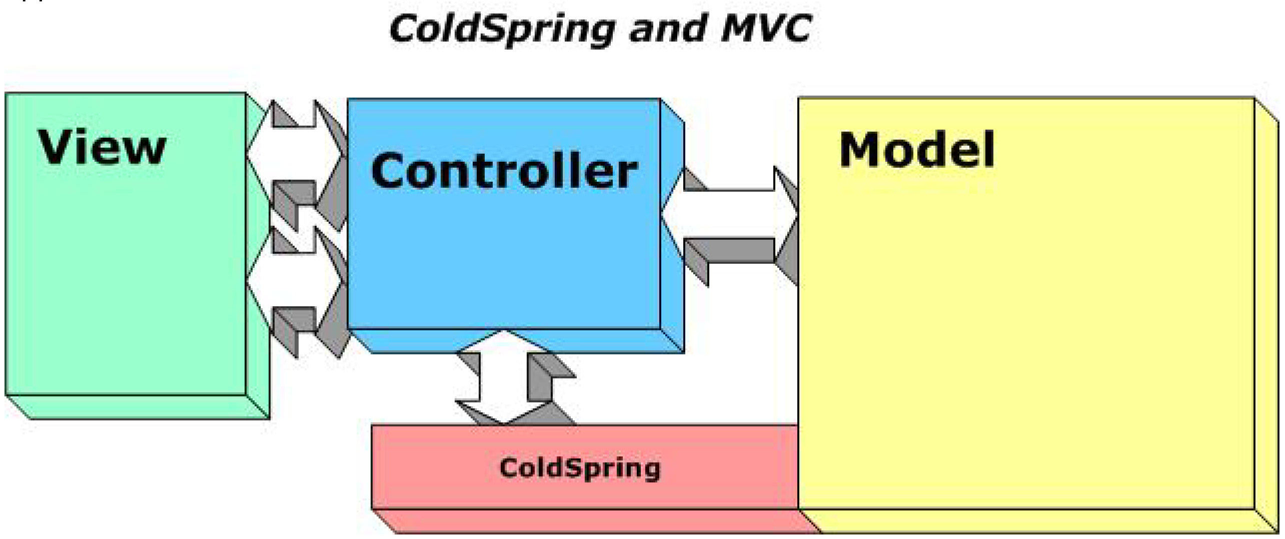
```
<cffunction name="setNotificationService" returntype="void" output="false" hint="Dependency: Notifcation Service">
      <cfargument name="NotificationSerivce" type="myApp.model.NotificationService" required="true"/>

      <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(…) />

III.II ColdSpring and MVC frameworks

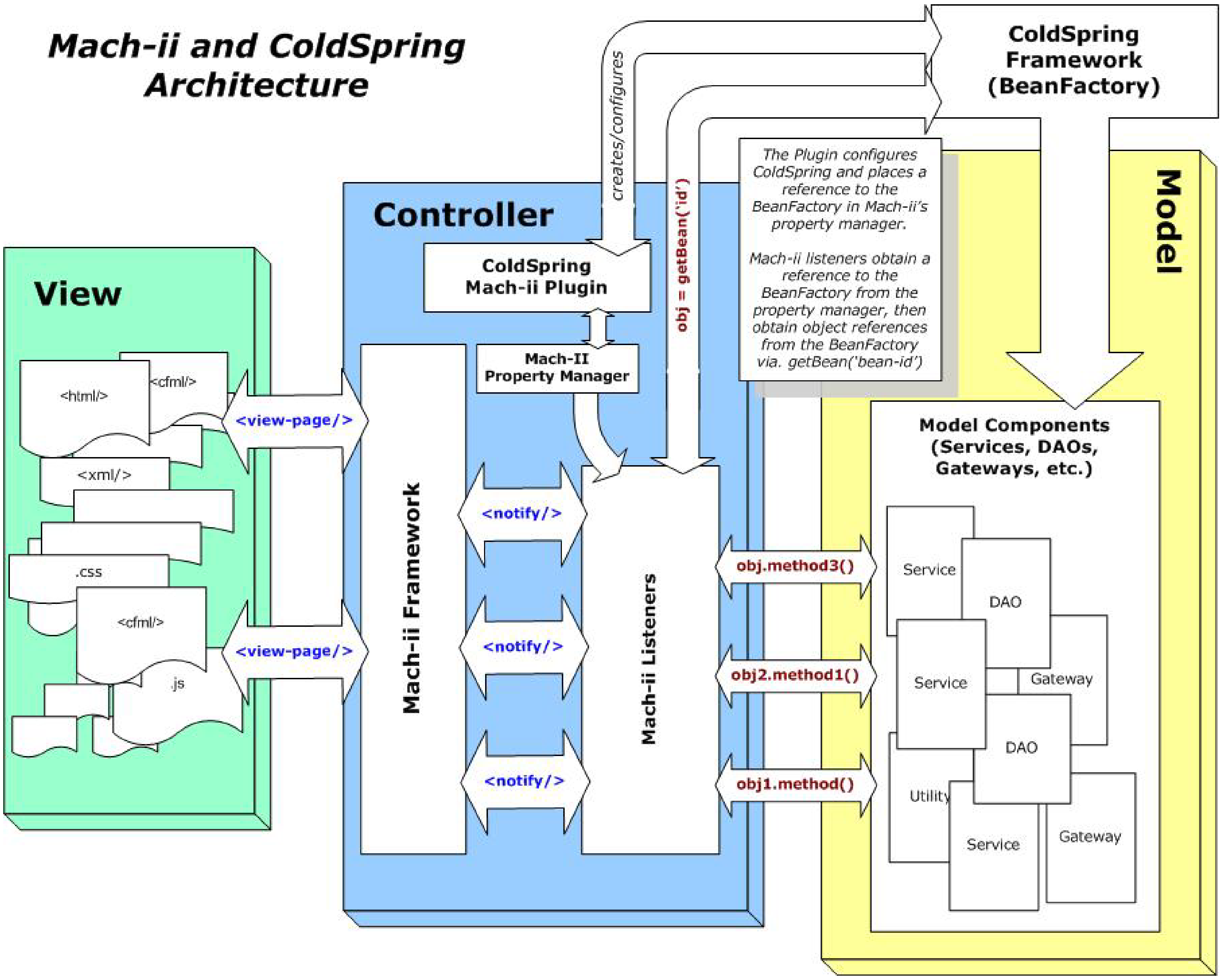
ColdSpring was also developed to fit in well with existing MVC (Model View Controller) frameworks, such as Fusebox 4 and Mach-II. To use ColdSpring with one of these frameworks, it's important to understand the "big picture", as in where ColdSpring sits in relation to the rest of the application. The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

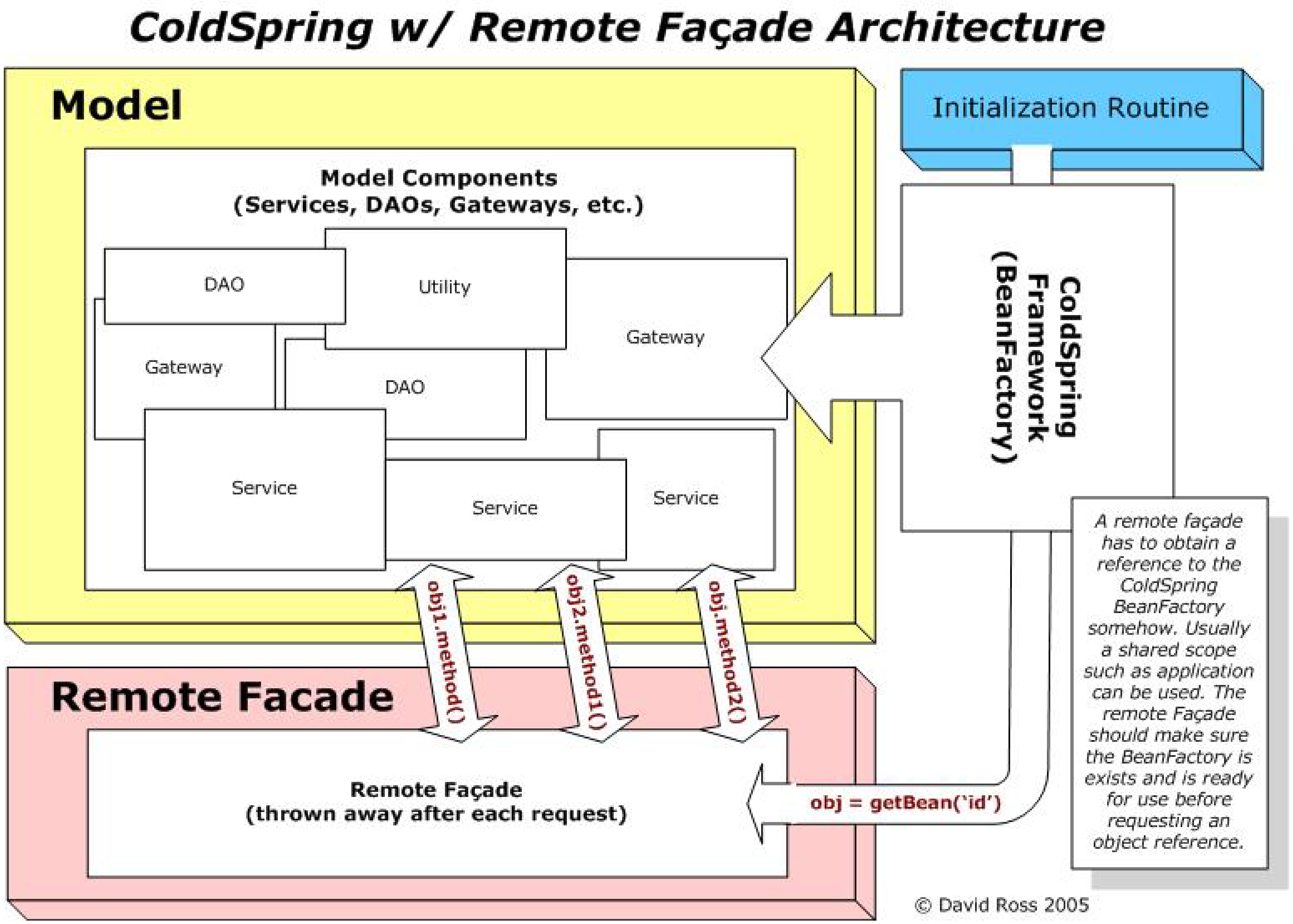
For those who use the Mach-II framework for a controller, ColdSpring ships with a Mach-II plugin (coldspring.machii.coldspringPlugin.cfc) that automates the creation of the bean factory. The following diagram details MachII and ColdSpring together:

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')
Controller
ColdSpring Mach-ii Plugin
Mach-II Property Manager
Model
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

You must let the plugin know the location of your bean definitions xml file, and the plugin will handle the rest: creating the bean factory, passing in the bean definitions, and then placing the whole thing in Mach-II's property manager. For usage detail, consult the source of the plugin.

III.III ColdSpring and Remoting


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

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”.

To see an example of a Service Layer, MVC framework integration (Mach-II and FuseBox 4) and an example of a Remote Façade, please refer to the ColdSpring sample app, located in the /examples directory within the ColdSpring source code.