Wiring It All Together

Dependency Injection with ColdSpring

alagad

120 Nelson Lane , Clayton, NC 27527 | p 1 (888) ALAGAD4 | f 1 (888) 248.7836 | www.alagad.com | info@alagad.com

Doug Hughes, President

What You Will Learn

- The problem ColdSpring solves
- How to configure your classes using Dependency Injection
- How to autowire components into Model-Glue controllers
- The basics behind Service Oriented Architecture

What is ColdSpring?

- In simplest terms, a configuration framework
- It's a Factory for building your components
- Based on the Spring Framework from Java

Yet More Design Patterns

- The Factory Pattern
  - ColdSpring "builds" your objects
- Dependency Injection / Inversion of Control
  - Provides looser coupling between components
  - Dependencies are automatically passed to components when they are instantiated

The Factory Pattern

- A "Factory" component knows how to "build" another type of component.
- Think of an automotive factory...
  - You ask the factory to build you a nice new car.
  - The factory gives you your car.
  - You don't care how it was built.
- The CreateObject() method is a good example of the Factory pattern.

The Dependency Injection Pattern

- First, let's cover "Loose Coupling"…

Loose Coupling

- Cohesive Components are easier to write and maintain.
- Cohesive Components do one thing well.
- An uncohesive component is one that does a number of different (possibly related) things, but specializes in none.
  - One component that does all data access
- Cohesiveness is worthy goal…
  - But, what if a CFC needs to do more than one thing?

Dependencies

- The engine in a car doesn't provide the energy to run it, the fuel does.
- The engine is dependent on the fuel to do it's job.
- Components can, and often do, use other components to complete their jobs.
  - A security system might use different components as interfaces to LDAP, databases, or other information stores.

Getting Our Dependencies

- You could Instantiate needed components inside your component.

```
<cfcomponent displayName="Engine">
    <cfset Fuel = CreateObject(....) />
    <cffunction name="startEngine">
        <cfset var energy = Fule.burn() />
        ....
    </cffunction>
</cfcomponent>
```

- But what about these other component's dependencies and configuration values?
  - You have to repeatedly configure that component.
  - Changes to how the component is created impact every creation point.
  - The engine shouldn't decide what octane fuel to use.
- If you're creating a component in another, you're tightly coupling the two components.

Loosen The Coupling

- Pass your dependencies in during construction:

```
<cfset Fuel = CreateObject(....) />
<cfset Fuel.setOctane("High") />

<cfcomponent>
    <cffunction name="init">
        <cfargument name="Fuel" />
        ....
    </cffunction>

    <cffunction name="startEngine">
        <cfset var energy = Fule.burn() />
        .....
    </cffunction>
</cfcomponent>
```


Dependency Injection

- Also known as Inversion of Control
- A pattern where a Factory creates a component and provides it with all of its dependencies automatically
- Factory knows how to create objects based on its configuration

```
<cfset Engine = MyFactory.getObject("Engine") />
```


But Isn't That a Dependency?

- **No!**
- The object being created has no knowledge of the Factory.

Bean Pattern

- A bean is simply an object with getters and setters.
- DI/IOC Factories use the bean pattern and the init() convention

Random Note: Have you noticed that none of the patterns we've discussed stand alone? They all work together!

ColdSpring

- An IOC Factory for ColdFusion
- Configured via XML
- A very simple API
- There's more to the framework too, but we're not going to look beyond dependancy injection

Configuring ColdSpring

- Configured via ColdSpring.xml
- Located under /Config by default

The Basic ColdSpring.xml

- <beans> (only one)
  - <bean> (zero or more)
    - <constructor-arg> (zero or more)
    - <property> (zero or more)

<beans>

- The root tag for ColdSpring configuration

```
<beans>
   <!-- any number of beans -->
   <bean id="..." class="...">
      <!-- property and constructor-arg elements
      defined here -->
   </bean>
</beans>
```


<bean>

- Defines the configuration for an object which implements the Bean Pattern
- Construct-arg values are passed into init method arguments
- Property values are passed into setters

```
<bean id="..." class="...">
   <!-- any number of constructor-arg tags -->
   <constructor-arg name="...">
      <!-- value, ref, bean, map or list tag -->
   </constructor-arg>
   <!-- any number of property tags -->
   <constructor-arg name="username">
      <!-- value, ref, bean, map or list tag -->
   </constructor-arg>
</bean>
```


<bean> Attributes

(abridged)

Id

- A unique id for the object being configured

Class

- The fully qualified path for the configured CFC

Singleton

- Indicates if only one instance of this class can ever exist in your entire application.
- Defaults to true.

<constructor-arg>

- The constructor-arg tag maps a configured value to an argument in an object's init method.

```
<constructor-arg name="datasource">
   <!-- value, ref, bean, map or list tag -->
   <value>Fortune</value>
</constructor-arg>
```

<property>

- The property tag maps a configured value to a setter on the object.

```
<property name="datasource">
  <!-- value, ref, bean, map or list tag -->
  <value>Fortune</value>
</property>
```


<value />

- Specifies a simple value

```
<value>15</value>
```


<ref />

- Reference another object configured with the bean tag by it's Id.

```
<ref bean="myBeanId"/>
```


<map/>

- Specifies a simple structure of name/value pairs

```
<map>
  <entry key="foo">
    <!-- value, ref, bean, map or list tag -->
    <value>5</value>
  </entry>
  <entry key="bar">
    <!-- value, ref, bean, map or list tag -->
    <ref id="barBean"/>
  </entry>
</map>
```


<list/>

- Specifies an array of values

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


Creating The BeanFactory

- The ColdSpring.beans. DefaultXmlBeanFactory component is used to load XML configuration.

```
<cfset ColdSpring = CreateObject("component",
   "coldspring.beans.DefaultXmlBeanFactory").init() />
<cfset ColdSpring.loadBeans("/path/to/ColdSpring.xml") />
```


Creating Beans

- Use getBean() to create an instance of a configured object
- Returns singletons by default
  - Yet another design pattern

```
<cfset CategoryGateway =
   ColdSpring.getBean("CategoryGateway") />
```


How CS and MG Fit Together

- ColdSpring used to configure the entire Model-Glue framework.
  - The nice side effect is that you can modify the behavior of Model-Glue without touching any of the core files.
- Model-Glue will automatically "wire" your controllers with configured objects.
  - Looks for setters on controller and their type.
    - setXyz() – accepts model.Xyz
  - Looks in ColdSpring configuration for a component with the same id and type.

```
<bean id="Xyz" type="model.Xyz" />
```


Getting Beans From Model-Glue Controllers

- You can use this code to load any configured bean in a controller:

```
<cfset var FortuneService =
  getModelGlue().getBean("FortuneService") />
```

- This has it's place, but it's cleaner to autowire beans

alagad

The Facade Pattern

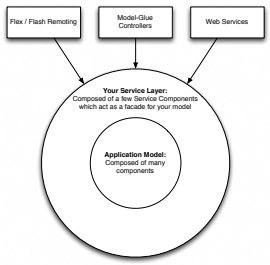
- Using a component to hide complexity in other components.
- Provides a simpler interface than using a set of components together.
- Wouldn't it be nice to just have one CFC that we could ask for a Fortune rather than several objects that work together?

alagad

Service Oriented Architecture

- An implementation of the Facade Pattern to simplify interactions with your model.
- Often one method per system use case.
- Can be exposed as a web service, to flash remoting, etc.
- Services are not tied (at all) to Model-Glue or ColdSpring

alagad



Service Layers

alagad

What You Learned

- ColdSpring is an Inversion of Control framework used to configure objects
- ColdSpring allows for more loosely coupled cohesive components
- How to configure objects with ColdSpring
- How to write Services which act as a facade for your model

Question and Answer

Exercise

- Make your Fortune CFC into a Fortune Service
- Configure your model using ColdSpring
- Update your unit tests to use ColdSpring to create objects (where appropriate)
- Expose your Fortune service as a Web Service

Discussion

- How did the exercise "feel"?
- Does this seem like more code or less?
- Do you think this could be more easily maintained?
- What if you were to switch database platforms?