Japanese Using Spring Factories with Groovy

Spring is an open-source framework created to address the complexity of Java enterprise application development. One of Spring's goals is to help developers write simple, testable and loosely coupled systems while reducing the amount of scaffolding code required. In this respect, Groovy has a common goal. So, for simple systems, Groovy alone may be sufficient for your needs. However, as your system grows in size and complexity, and especially in hybrid Java/Groovy environments, you might find Spring's facilities provide great value to your Groovy system development.

Here we look at using Spring's Bean Factory mechanisms within Groovy. These facilities allow beans to be managed within a Spring container. The beans are normally Java objects, but since Groovy objects are Java objects, Spring can just as easily manage Groovy objects for you. In particular, in mixed Java/Groovy environments, you can leverage any existing domain objects or services already have defined in your Spring wiring from the Java part of your application. You can immediately start using those within your Groovy scripts and code, allowing a great mix of strongly-typed Java domain objects and dynamically typed Groovy code.

Let's start by exploring a simple calculator application.

Bare Bones Approach

Suppose we have the following implementation class:

```
class CalcImpl {
    def doAdd(x, y) { x + y }
}
```

We can make use of that class from a script as follows:

```
import org.springframework.beans.factory.support.*

def configure(factory) {
    def bd = new RootBeanDefinition(CalcImpl)
    factory.registerBeanDefinition('calcBean', bd)
}

def factory = new DefaultListableBeanFactory()
configure(factory)

def calc = factory.getBean('calcBean')
println calc.doAdd(3, 4) // => 7
```

This script relies on no external wiring files. Everything is configured in the script itself. If we wish to alter our system at a later time, we simply alter the configuration inside the `configure()` method. In the Java world, this wouldn't be very flexible, but in the Groovy world, this script may be executed from source code (even dynamically loaded when it changes) so altering the configuration doesn't necessarily require a new build.

Classical Spring Approach

Probably the most common way to use Spring is to use an XML *wiring* file. As systems grow larger, wiring files allow configuration to be centralised in easy to change 'groupings' of beans. Let's assume our calculator needs to eventually be expanded to have additional functionality. An approach to handling complexity as the system grows is to delegate functionality off to other components. Here is how we might code up an *ad der* component:

```
class AdderImpl {
    def add(x, y) { x + y }
}
```

Then, applying the delegate design pattern we mentioned earlier would result in the following refactored calculator:

```
class CalcImpl2 {
    def AdderImpl adder
    def doAdd(x, y) { adder.add(x, y) }
}
```

To capture our software system configuration, we will use an XML wiring file (we have two beans):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="calcBean" class="CalcImpl2" autowire="byType"/>
    <bean class="AdderImpl"/>
</beans>
```

Now, our script code looks like:

```
import org.springframework.context.support.ClassPathXmlApplicationContext

def ctx = new ClassPathXmlApplicationContext('calcbeans.xml')
def calc = ctx.getBean('calcBean')
println calc.doAdd(3, 4) // => 7
```

Annotation Approach

If we wish, we can remove the need for an XML file by using annotations. Note that we need to think carefully before using this approach extensively in a large system for two reasons. Firstly, we are more tightly coupling our system to Spring. Secondly, we are associating configuration and deployment information with our source code. Perhaps separating those concerns will have enormous benefits for large systems. The good news is that Spring lets you take on board just those annotations that you are happy to use.

Here is how we might code up our adder component:

```
import org.springframework.stereotype.Component

@Component class AdderImpl {
    def add(x, y) { x + y }
}
```

Here is our modified calculator:

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Component

@Component class CalcImpl3 {
    @Autowired private AdderImpl adder
    def doAdd(x, y) { adder.add(x, y) }
}
```

And here is our script code (note no XML file is required):

```
import org.springframework.context.support.GenericApplicationContext
import org.springframework.context.annotation.ClassPathBeanDefinitionScanner

def ctx = new GenericApplicationContext()
new ClassPathBeanDefinitionScanner(ctx).scan('') // scan root package for components
ctx.refresh()
def calc = ctx.getBean('calcImpl3')
println calc.doAdd(3, 4) // => 7
```

This example uses features in Spring 2.1 (currently at a Milestone release) and Groovy 1.1 (currently in beta release) on a Java 5 or greater JVM.

BeanBuilder Approach

We can also use the BeanBuilder from Grails to avoid writing an XML file. It will look something like this (after adding the latest Grails jar to your classpath):

```
def bb = new grails.spring.BeanBuilder()
bb.beans {
    adder(AdderImpl)
    calcBean(CalcImpl2) { delegate.adder = adder } // need to use delegate may change
}
def ctx = bb.createApplicationContext()
def calc = ctx.getBean('calcBean')
println calc.doAdd(3, 4) // => 7
```

Further Information

- The Spring Example in Groovy and JMX
- Spring Documentation