Extracted from:

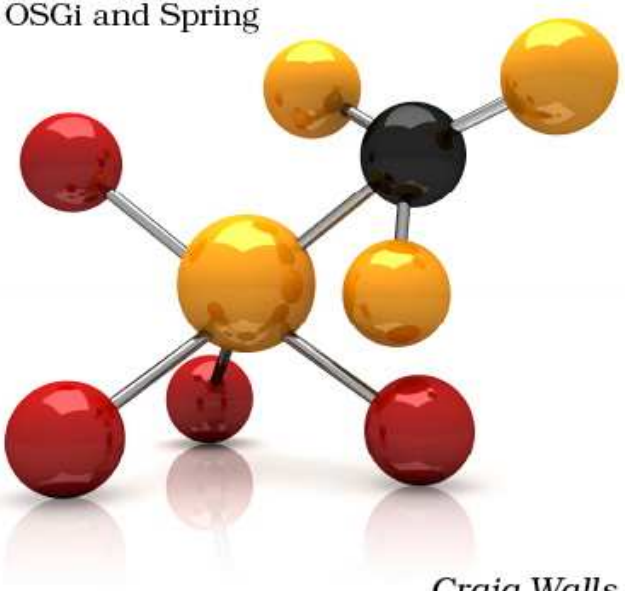
Modular Java

Creating Flexible Applications with OSGi and Spring

The Pragmatic Programmers

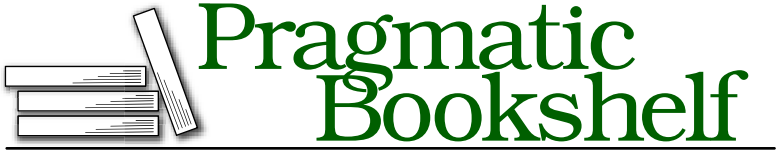
Modular Java

Creating Flexible Applications with OSGi and Spring



Craig Walls

Edited by Jacquelyn Carter



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-40-9

ISBN-13: 978-1934356-40-1

Printed on acid-free paper.

P1.0 printing, May 2009

Version: 2009-6-18

How to Not Publish the Spring Context as a Service

If you'd rather not have a bundle's Spring context published as a service, you'll need to say so with the Spring-Context: header:

```
Spring-Context: META-INF/spring/*.xml;publish-context:=false
```

By setting the publish-context directive to false, we're asking Spring-DM to go ahead and load the Spring context using XML files in META-INF/spring, but not to publish the context in the OSGi service registry.

baker's dozen of interfaces, any of which you can use to retrieve the bundle's Spring context.

Now that we've converted the index bundle to use Spring-DM, let's turn our attention to the spider bundle to see whether Spring-DM can help us eliminate all of the code that we wrote to consume the index service.

6.3 Injecting Services into Consumers

As you'll recall, there's much more to consuming a service than publishing it. A service consumer must carefully deal with the transitivity of services to make sure that it's not trying to use a service that has gone away or that has been replaced with a newer version. All of that service management resulted in a lot of code in both the spider bundle's activator and in the spider implementation class.

Spring-DM was able to eliminate OSGi-specific code in our index bundle. Can it do the same for the spider bundle? You bet! In fact, as you'll soon see, consuming a service with Spring-DM isn't much different from publishing a service.

First things first. . . just as with the index bundle, we're no longer going to need the bundle activator for the spider bundle. So, let's go ahead and ditch it:

```
dwmjs% cd spider
spider% rm src/main/java/dwmj/spider/impl/SpiderActivator.java
```

Be sure to remove the Bundle-Activator: entry from osgi.bnd, too.

Now that the spider's bundle activator is gone, we no longer have a way to give the MavenSpider a service tracker to look up the index service.

But that's OK, because instead of giving MavenSpider a way to get the index service, we're going to inject the index service into MavenSpider:

Download dwmjs/spider/src/main/java/dwmj/spider/internal/MavenSpider.java

```
public class MavenSpider {
   private static final Logger LOGGER = Logger.getLogger("MavenSpider");
   private JarFilePopulator[] jarFilePopulators = new JarFilePopulator[] {};
   private String repositoryUrl;
   private boolean active = true;
   private IndexService indexService;

   public MavenSpider(IndexService indexService) {
      this.indexService = indexService;
   }

}
```

We've traded a ServiceTracker for a reference to an IndexService. Actually, we're going to inject MavenSpider with a proxy to the index service (that automatically handles the transitive nature of services). But for all intents and purposes, you can pretend that it's the real index service—MavenSpider won't know the difference.

Notice that MavenSpider no longer implements java.lang.Runnable. Originally, we had to start MavenSpider in a separate thread so that it would not hold up the spider bundle from starting. But now Spring is going to start MavenSpider, so it no longer needs to implement Runnable.

There's just one more tweak we must make to MavenSpider to make it ready for Spring-DM. Now that we're injecting an IndexService reference into MavenSpider, we'll need to change it to just use the IndexService and not try to look it up from the ServiceTracker. Previously, MavenSpider had a snippet of code that looked like this:

Download dwmj/spider/src/main/java/dwmj/spider/internal/MavenSpider.java

```
if (jarFile.isIndexable()) {
   IndexService indexService =
         (IndexService) indexServiceTracker.getService();

   if(indexService != null) {
      indexService.addJarFile(jarFile);
   }
}
```

But with the ServiceTracker gone, it's much simpler:

Download dwmjs/spider/src/main/java/dwmj/spider/internal/MavenSpider.java

```
if(jarFile.isIndexable()) {
   indexService.addJarFile(jarFile);
}
```

Awesome! We've managed to turn MavenSpider into a POJO, eliminating all hints of the OSGi API. To wrap up the conversion of the spider bundle to use Spring-DM, we need to wire MavenSpider as a Spring bean, injecting it with a reference to the index service. First, we'll configure a reference to the index service:

Download dwmjs/spider/src/main/resources/META-INF/spring/spider-osgi.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/osgi"
       xmlns:beans="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/osgi
            http://www.springframework.org/schema/osgi/spring-osgi.xsd
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <reference id="indexService"
       interface="dwmj.index.IndexService" />

</beans:beans>
```

The <*reference*> element isn't much different from the <*service*> element, only in reverse. The interface= attribute tells Spring to look up a service from the OSGi service registry with the dwmj.index.IndexService interface. The id= attribute is effectively the flipside of the <*service*> element's ref= element—but, instead of referencing another Spring bean, the id= attribute gives the index service proxy a name with which we can inject it into the MavenSpider.

Speaking of injecting the index service into the MavenSpider, let's wire up the spider bean:

Download dwmjs/spider/src/main/resources/META-INF/spring/spider-context.xml

```
<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-2.5.xsd">

  <bean class="dwmj.spider.internal.MavenSpider"
      init-method="run" destroy-method="stop">
      <constructor-arg ref="indexService" />
```

```
      <property name="repositoryUrl" value="http://repo2.maven.org/maven2/" />
      <property name="jarFilePopulators">
        <list>
         <bean class=
           "dwmj.spider.internal.PomBasedJarFilePopulator" />
         <bean class=
          "dwmj.spider.internal.JarContentBasedJarFilePopulator" />
        </list>
      </property>
  </bean>
</beans>
```

The third <*constructor-arg*> injects the bean named *indexService* into the MavenSpider as it's constructed. Once it has been constructed, Spring will start the spider by calling the run() method, as indicated by the <*bean*> element's init-method= attribute. Later, when the Spring context is shut down (when the bundle is stopped), Spring will invoke the stop() method to stop the spider, as indicated by the destroy-method= attribute.

The spider bundle is now converted from a bundle whose Java code is strewn with bits of the OSGi API to one containing simple POJOs that are managed by Spring. Before we move on, there's one more thing left to do. . . let's build the spider bundle and see whether it works. First, the build. . .

```
spider% mvn install
[INFO] Scanning for projects...
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 7 seconds
[INFO] Finished at: Fri Mar 20 18:00:13 CDT 2009
[INFO] Final Memory: 14M/31M
[INFO] ------------------------------------------------------------------------
spider%
```

. . . and then the provision:

```
spider% cd ..
dwmjs% pax-provision
[INFO] Scanning for projects...
...
osgi> ss

Framework is launched.
```

Joe Asks...

What If a Service Isn't Available?

When we used the <*reference*> element to consume the index service, we assumed that the service would be readily available. But what if it isn't?

By default, Spring-DM will wait five minutes for the service to become available before an unchecked ServiceUnavailableException is thrown. If you want to change the timeout period, you have two options.

First, you can set the timeout= attribute on the <*reference*> element to adjust the timeout on a reference-by-reference basis:

```
<reference id="indexService"
  interface="com.dudewheresmyjar.index.IndexService"
  timeout="60000"/>
```

Or you can change the default timeout value by setting the osgi:default-timeout= at the root of the XML file:

```
<beans:beans xmlns="http://www.springframework.org/schema/osgi"
    xmlns:osgi="http://www.springframework.org/schema/osgi"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/osgi
       http://www.springframework.org/schema/osgi/spring-osgi.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/
              spring-beans-2.5.xsd"
    osgi:default-timeout="60000">

  <reference id="indexService"
       interface="com.dudewheresmyjar.index.IndexService" />

</beans:beans>
```

```
id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.2.R34x_v20080826-1230
1       ACTIVE      org.eclipse.osgi.util_3.1.300.v20080303
2       ACTIVE      org.eclipse.osgi.services_3.1.200.v20070605
3       ACTIVE      org.ops4j.pax.logging.pax-logging-api_1.3.0
4       ACTIVE      org.ops4j.pax.logging.pax-logging-service_1.3.0
5       ACTIVE      org.springframework.osgi.extender_1.2.0
6       ACTIVE      org.springframework.osgi.core_1.2.0
7       ACTIVE      org.springframework.osgi.io_1.2.0
8       ACTIVE      com.springsource.slf4j.org.apache.commons.logging_1.5.0
9       ACTIVE      com.springsource.slf4j.api_1.5.0
                    Fragments=10
10      RESOLVED    com.springsource.slf4j.log4j_1.5.0
                    Master=9
11      ACTIVE      org.springframework.aop_2.5.6
12      ACTIVE      org.springframework.beans_2.5.6
13      ACTIVE      org.springframework.context_2.5.6
14      ACTIVE      org.springframework.core_2.5.6
15      ACTIVE      org.springframework.test_2.5.6
16      ACTIVE      com.springsource.org.aopalliance_1.0.0
17      ACTIVE      org.springframework.transaction_2.5.6
18      ACTIVE      com.dudewheresmyjar.domain_1.0.0.SNAPSHOT
19      ACTIVE      org.compass-project.compass_2.1.1
20      ACTIVE      com.dudewheresmyjar.index_1.0.0.SNAPSHOT
21      ACTIVE      com.dudewheresmyjar.spider_1.0.0.SNAPSHOT

osgi>
```

So far so good. There were no exceptions thrown, and everything seems to be working. Let's check the spider bundle to make sure that it's using the index service:

```
osgi> bundle 21
initial@reference:file:com.dudewheresmyjar.spider_1.0.0.SNAPSHOT.jar/ [21]
  Id=21, Status=ACTIVE      Data Root=/Users/wallsc/Projects/projects/dwmjs/runner/
      equinox/org.eclipse.osgi/bundles/21/data
  No registered services.
  Services in use:
    {org.springframework.beans.factory.xml.NamespaceHandlerResolver}={service.id=24}
    {org.xml.sax.EntityResolver}={service.id=25}
    {dwmj.index.IndexService}={org.springframework.osgi.bean.name=indexService,
      Bundle-SymbolicName=com.dudewheresmyjar.index, Bundle-Version=1.0.0.SNAPSHOT,
      service.id=26}
  Exported packages
    dwmj.spider.impl; version="1.0.0.SNAPSHOT"[exported]
  Imported packages
    dwmj.domain; version="1.0.0.SNAPSHOT"<initial@reference:file:
        com.dudewheresmyjar.domain_1.0.0.SNAPSHOT.jar/ [18]>
    dwmj.index; version="1.0.0.SNAPSHOT"<initial@reference:file:
        com.dudewheresmyjar.index_1.0.0.SNAPSHOT.jar/ [20]>
    javax.swing.text; version="0.0.0"<System Bundle [0]>
```

Spring-DM to Become Part of the OSGi Specification

As I write this, a draft of the OSGi 4.2 Specification is available. Within this specification is RFC 124: A Component Model for OSGi. It's commonly being referred to as the Blueprint Service.

What's curious about the Blueprint Service is that it looks suspiciously like Spring-DM. A few names and terms have been changed, but for the most part, Blueprint Service is Spring-DM. This should come as no surprise to anyone who looks closely as the specification—it is written by employees of SpringSource, the company behind the Spring Framework and Spring-DM.

See Appendix C, on page 218, for more information on how the Blueprint Service compares to Spring-DM.

```
    javax.swing.text.html; version="0.0.0"<System Bundle [0]>
    javax.xml.parsers; version="0.0.0"<System Bundle [0]>
    javax.xml.xpath; version="0.0.0"<System Bundle [0]>
    org.w3c.dom; version="0.0.0"<System Bundle [0]>
  No fragment bundles
  Named class space
    com.dudewheresmyjar.spider; bundle-version="1.0.0.SNAPSHOT"[provided]
  No required bundles

osgi>
```

And there it is. The third entry under the *Services in use:* header tells us that the spider bundle is using the index service—thanks to Spring-DM and the <*reference*> element.

At this point we've developed all the bundles of the *Dude, Where's My JAR?* application twice: once using the core OSGi API and again using Spring-DM. Before we move on, let's take a moment to reflect on what Spring-DM has done for us.

Without Spring-DM, both publication and consumption of a service required writing directly to the OSGi API. In both cases, we had to write an activator class to either register or retrieve a service in the OSGi service registry. On the other hand, with Spring-DM we were able to publish and consume services in a declarative fashion, with no need to interact directly with the OSGi API.

Now we're ready to put a face on this application. Coming up in the next chapter, we're going to develop the web front end of the application.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Modular Java's Home Page
http://pragprog.com/titles/cwosg
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/cwosg.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com