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

For brevity's sake, I've cut out most of the output produced when running the test. But the punch line is the same: the test passes. Therefore, we know that our index service is working correctly (or at least within the expectations of the shouldIndexAndFindAJarFileObject() method). As we continue to develop the application, we'll know whether the changes we make break the index service, because this test will be the first to complain.

Our application is really starting to take shape. In this chapter, we added another bundle to the mix—this time with a service published in the OSGi service registry. And even though we haven't yet developed any bundles that consume that service, we've been able to test drive it with an integration test driven by Pax Exam.

But a service isn't any good unless someone uses it. Let's build something that uses the index service.

5.3 Consuming OSGi Services

As you'll recall from Chapter 3, *Dude, Where's My JAR?*, on page 47, the index service will ultimately have two consumers: the web front end and the repository spider. The web front end will use the index service to look search for JAR files that meet a user's criteria. The spider will use the index service to stock the search engine's index with the JAR files that it finds in Maven repositories. We'll get to the web front end later in Chapter 7, *Creating Web Bundles*, on page 131. But we'll go ahead and build the spider now.

First things first...the repository spider represents another module of our application and thus will be contained within its own bundle. Therefore, we'll need to create a new bundle project. Once again, we call on the pax-create-bundle script:

```
dwmj% pax-create-bundle -g com.dudewheresmyjar -p dwmj.spider -n spider \
?                       -v 1.0.0-SNAPSHOT
[INFO] Scanning for projects...
...
[INFO] Archetype created in dir: /Users/wallsc/Projects/projects/dwmj/spider
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 8 seconds
[INFO] Finished at: Sat Mar 07 16:43:22 CST 2009
[INFO] Final Memory: 10M/19M
[INFO] ------------------------------------------------------------------------
dwmj%
```

As usual, pax-create-bundle adds an example service, service interface, and activator to the generated project. Go ahead and remove them, and we'll be ready to develop the spider bundle.

Using Service Trackers

The first thing we'll need to do is to create the spider implementation class. Spidering a Maven repository is quite involved. For the purposes of our application, this involves several steps such as parsing POM files, reading a JAR file's contents, and extracting information from a JAR's META-INF/MANIFEST.MF file. For the most part, however, the functionality of the spider has nothing to do with OSGi. Therefore, in the interest of saving space and to keep our focus on consuming services, I'm going to show only the parts of the spider that are pertinent to the topic of consuming OSGi services.[1]

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

```
package dwmj.spider.internal;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;

import javax.swing.text.MutableAttributeSet;
import javax.swing.text.html.HTML;
import javax.swing.text.html.HTMLEditorKit;
import javax.swing.text.html.HTML.Tag;
import javax.swing.text.html.HTMLEditorKit.Parser;
import javax.swing.text.html.HTMLEditorKit.ParserCallback;

import org.osgi.util.tracker.ServiceTracker;

import dwmj.domain.JarFile;
import dwmj.index.IndexService;

public class MavenSpider implements Runnable {
   private JarFilePopulator[] jarFilePopulators = new JarFilePopulator[] {};
   private final ServiceTracker indexServiceTracker;
   private String repositoryUrl;
   private boolean active;

   public MavenSpider(ServiceTracker indexServiceTracker) {
      this.indexServiceTracker = indexServiceTracker;
   }
```

1. Remember, you can download the complete source code from http://www.pragprog.com/titles/cwosg/source_code.

```
    public void setRepositoryUrl(String repositoryUrl) {
        this.repositoryUrl = repositoryUrl;
    }

    // ...
    private void handleJarFile(String jarUrl) {
       // ...
             IndexService indexService =
                   (IndexService) indexServiceTracker.getService();

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

    // ...
}
```

The MavenSpider class is constructed by passing in a service tracker. You're probably wondering what this odd little class is for. Ultimately, doesn't MavenSpider need the index service? Why not just give it the index service straightaway? Why all of the indirection?

OSGi services are a tricky bunch. They can come and go at any time. There's no way to be sure that if we give an index service to the MavenSpider at creation that the index service will still be around when we're ready to use it. For that matter, there's no guarantee that the index service is even available when we create the MavenSpider.

Rather than putting MavenSpider in the awkward position of having to manage the comings and goings of the index service, we will use a service tracker. Service trackers contain all of the magic to keep track of whether a service is available, and they hide away the complexity of dealing with the OSGi service registry through lower-level APIs. MavenSpider is given a service tracker that keeps track of the index service and, upon request through the getService() method, provides the index service so that we can add a JarFile to the index.

Even though the service tracker abstracts away any unpleasantness of dealing with the service registry's low-level APIs, getService() could still return null if the service is unavailable. So, we will need to check for a null service before calling addJarFile(). But if you'd rather wait for the service to become available, we could call waitForService() instead of getService().

Caution: Don't Dawdle in an Activator

The waitForService() method will block until a service is available or the specified timeout has passed. For that reason, avoid specifying a long timeout when using waitForService() in an activator's start() or stop() method. If the service isn't available, the bundle will get stuck in STARTING or STOPPING state while transitioning to or from an ACTIVE state.

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

```
try {
   IndexService indexService =
      (IndexService) indexServiceTracker.waitForService(10000);

   // ...
}
catch (InterruptedException e) {
   // handle exception
}
```

Unlike getService(), which returns immediately, waitForService() will wait for a service to become available, up to a specified timeout (in milliseconds). In this case, waitForService() will wait up to ten seconds for the service to become available before giving up. A timeout of zero tells waitForService() to wait indefinitely.

Now that we've spent some time looking at how to use a service tracker to look up a service from the OSGi registry, you're probably wondering where that service tracker comes from. For the answer to that, look no further than SpiderActivator, the spider bundle's activator:

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

```
package dwmj.spider.internal;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.util.tracker.ServiceTracker;

import dwmj.index.IndexService;

public final class SpiderActivator implements BundleActivator {
   private ServiceTracker indexServiceTracker;

   private static String[] REPOSITORIES = new String[] {
         "http://www.dudewheresmyjar.com/repo/" };
```

```
    private static JarFilePopulator[] POPULATORS = new JarFilePopulator[] {
        new PomBasedJarFilePopulator(), new JarContentBasedJarFilePopulator()
    };

    private final MavenSpider[] spiders = new MavenSpider[REPOSITORIES.length];

    public void start(BundleContext context) throws Exception {
        indexServiceTracker = new ServiceTracker(context, IndexService.class
                    .getName(), null);
        indexServiceTracker.open();

        for (int i = 0; i < REPOSITORIES.length; i++) {
            MavenSpider spider = new MavenSpider(indexServiceTracker);
            spider.setRepositoryUrl(REPOSITORIES[i]);
            spider.setJarFilePopulators(POPULATORS);

            Thread thread = new Thread(spider);
            thread.start();
        }
    }

    public void stop(BundleContext context) throws Exception {
        for (int i = 0; i < spiders.length; i++) {
            spiders[i].stop();
        }
        indexServiceTracker.close();
    }
}
```

SpiderActivator's main job is to create an instance of MavenSpider for each Maven repository that will be crawled (in this case, an artificial repository). But first, it creates a service tracker to track the index service. The constructor for ServiceTracker takes three parameters:

- The bundle context
- The name of the service to be tracked
- An optional service tracker customizer (org.osgi.util.tracker.ServiceTrackerCustomizer)

Since we need to track the index service, we pass in the bundle context and the fully qualified name of the IndexService interface.

As for the third parameter, ServiceTrackerCustomizer is an odd little interface that lets us hook into the service tracker to monitor when services are added, removed, or modified. We won't need a service tracker customizer, though—so we'll give it a null service tracker customizer.

The last thing that the activator does is create a MavenSpider instance for each of the repositories and sends them off to crawl. So that the

Please Don't Crawl IBiblio

As a consequence of crawling a repository, the spider generates a lot of traffic. Maven repositories are geared toward serving occasional requests for Java libraries but may not be prepared to handle a barrage of requests from our spider.

Please be a good citizen, and do not configure the spider to crawl the central repository at IBiblio or any other repository that you do not have express permission to crawl. Or better yet, set up a local repository, and set the spider bundle to crawl it.

start() method can finish without waiting for the crawlers (Maven repositories are large—it might take awhile), SpiderActivator fires off a thread for each spider to crawl in.

The spider bundle is almost complete. The only thing left to do is to register SpiderActivator as the bundle's activator by adding a line in the BND instruction file:

Download dwmj/spider/osgi.bnd

```
Bundle-Activator: dwmj.spider.internal.SpiderActivator
```

All of the bundle's pieces are in place. We're almost ready to build and deploy the spider bundle and watch it crawl a repository.

Deploying the Spider Bundle

There's only one more thing to do before we can build the spider bundle. Since the spider directly depends on classes and interfaces from the domain and index bundles, we'll need to make sure that they're in the compile-time classpath. For that, we'll use Pax Construct's pax-add-dependency script. First, we'll add the domain bundle as a dependency to the spider bundle:

```
spider% pax-import-bundle -g com.dudewheresmyjar -a domain -v 1.0.0-SNAPSHOT
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building com.dudewheresmyjar.spider [dwmj.spider]
[INFO]    task-segment: [org.ops4j:maven-pax-plugin:1.4:import-bundle]
          (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] [pax:import-bundle]
[INFO] Adding com.dudewheresmyjar.domain [dwmj.domain] as dependency to
          com.dudewheresmyjar:spider:bundle:1.0.0-SNAPSHOT
```

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5 seconds
[INFO] Finished at: Sat Mar 07 21:52:40 CST 2009
[INFO] Final Memory: 8M/18M
[INFO] ------------------------------------------------------------------------
spider%
```

Then we'll add the index bundle:

```
spider% pax-import-bundle -g com.dudewheresmyjar -a index -v 1.0.0-SNAPSHOT
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building com.dudewheresmyjar.spider [dwmj.spider]
[INFO]    task-segment: [org.ops4j:maven-pax-plugin:1.4:import-bundle]
          (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] [pax:import-bundle]
[INFO] Adding com.dudewheresmyjar.index [dwmj.index] as dependency to
          com.dudewheresmyjar:spider:bundle:1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5 seconds
[INFO] Finished at: Sat Mar 07 21:53:00 CST 2009
[INFO] Final Memory: 8M/18M
[INFO] ------------------------------------------------------------------------
spider%
```

The pax-add-dependency script should have added the domain and index bundles as <*dependency*>s in the spider bundle's pom.xml file. Now that the spider bundle is set dependency-wise, let's try building it:

```
spider% mvn install
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building com.dudewheresmyjar.spider [dwmj.spider]
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------------------
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 9 seconds
[INFO] Finished at: Sat Mar 07 22:01:03 CST 2009
[INFO] Final Memory: 14M/31M
[INFO] ------------------------------------------------------------------------
spider%
```

Good deal! The spider bundle was successfully built.

Now we're ready to provision it and see whether it works:

```
dwmj% pax-provision
[INFO] Scanning for projects...
...
osgi> ss

Framework is launched.

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      com.dudewheresmyjar.domain_1.0.0.SNAPSHOT
6       ACTIVE      org.compass-project.compass_2.1.1
7       ACTIVE      com.dudewheresmyjar.index_1.0.0.SNAPSHOT
8       ACTIVE      com.dudewheresmyjar.spider_1.0.0.SNAPSHOT

osgi>
```

After running pax-provision and using the Equinox ss command, you'll see that the spider bundle was installed and started. Moreover, if you issue the bundle command to view the spider bundle's information...

```
osgi> bundle 8
initial@reference:file:com.dudewheresmyjar.spider_1.0.0.SNAPSHOT.jar/ [8]
  Id=8, Status=ACTIVE      Data Root=/Users/wallsc/Projects/projects/dwmj/runner/
          equinox/org.eclipse.osgi/bundles/8/data
  No registered services.
  Services in use:
    {dwmj.index.IndexService}={service.id=24}
  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/ [5]>
    dwmj.index; version="1.0.0.SNAPSHOT"<initial@reference:file:
            com.dudewheresmyjar.index_1.0.0.SNAPSHOT.jar/ [7]>
    javax.swing.text; version="0.0.0"<System Bundle [0]>
    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.osgi.framework; version="1.4.0"<System Bundle [0]>
    org.osgi.util.tracker; version="1.3.3"<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>
```

. . . you'll find that the spider bundle uses the service identified as dwmj.index.IndexService (look under the *Services in use:* header). Also, if you wait a moment or two, you'll see the spider interacting with the index service as it finds JAR files in the Maven repository.

Finally, as one more bit of proof that the index service is indexing JarFiles on behalf of the spider, go dig around in the index directory (probably /tmp/dudeindex on Unix or c:\temp\dudeindex on Windows). This directory contains a set of files that comprise a Lucene index. While the spider is running, the selection of files and the sizes of those files will fluctuate, indicating that new entries are being written to the index.

In this chapter, we've developed two of the central bundles of our application. The index bundle publishes a service through which consumers can add and search for JarFile entries in an index. The spider bundle is one such consumer of the index service, crawling a Maven repository and submitting what it finds to the index service for indexing.

We'll write some code to search that index when we develop the web front end in Chapter 7, *Creating Web Bundles*, on page 131. But before we get there, let's push rewind on the project and see how Spring Dynamic Modules for OSGi (Spring-DM) brings a POJO-based programming model to OSGi, simplifying some of the OSGi plumbing code we've written so far.

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, new titles and other offerings.

Buy the Book

If you liked this eBook, 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

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