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

Pragmatic Bookshelf

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

```
id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       ACTIVE      com.pragprog.HelloWorld_1.0.0

osgi>
```

That was pretty cool. But it shows only half of what our bundle can do. Now let's stop the bundle and see what happens:

```
osgi> stop 1
Goodbye World!

osgi>
```

As expected, stopping the bundle yielded a "Goodbye World!" message on the screen. And if we issue an ss command again, we'll see that its status is no longer in ACTIVE state:

```
osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       RESOLVED    com.pragprog.HelloWorld_1.0.0

osgi>
```

Were you a little surprised to see the bundle in RESOLVED state? Maybe you were expecting it to go back to INSTALLED state. For now, don't worry too much about bundle states—it's enough to just know that the bundle is no longer active. We'll examine the bundle life cycle in more detail later in Section 4.3, *Following the Bundle Life Cycle*, on page 80.

I couldn't be more excited! We've just built our first OSGi bundle, deployed it to an OSGi container, and seen it do its stuff. If you'd like, you can kick it around some more. Feel free to start it and stop it again as many times as you like. But don't get too carried away. . . there's more fun in store for the *Hello World* example.

2.3 A *Hello World* Service Bundle

A bundle can do a lot of things. It can simply act as a library, providing classes and interfaces for other bundles to use. Or, as we've already seen with the previous example, a bundle can contain an activator that performs some action when the bundle is started and stopped.

Another thing that a bundle can do is publish services to be consumed by other bundles. To illustrate, let's rip our *Hello World* example into two parts: a bundle that publishes a service that provides greetings and another bundle that contains a consumer of the service and prints those greetings.

Publishing a Hello Service

The first step in creating a service is deciding what its interface will look like. In OSGi, a service's interface defines not only how other components can interact with the service but also how the other components find the service. For our *Hello World* service, we'll need two methods: one to return some hello message and one to return a goodbye message. The following interface should do the trick:

Download hello-service/src/main/java/com/pragprog/hello/service/HelloService.java

```
package com.pragprog.hello.service;

public interface HelloService {
   String getHelloMessage();

   String getGoodbyeMessage();
}
```

Now we write the service implementation class. To keep things interesting, the following service implementation has an international flair:

Download hello-service/src/main/java/com/pragprog/hello/service/impl/HelloImpl.java

```
package com.pragprog.hello.service.impl;

import com.pragprog.hello.service.HelloService;

public class HelloImpl implements HelloService {
   public String getHelloMessage() {
      return "Bonjour!";
   }

   public String getGoodbyeMessage() {
      return "Arrivederci!";
   }
}
```

Take notice of the service implementation's package and how it differs from the interface's package. Although both could reside in the same package, it's a good practice to keep them separate. As we'll soon see, keeping them separate will make it possible to publish the service under an exported interface for other bundles to use, while keeping the implementation of the service hidden from its consumers.

In OSGi, services are published to a service registry within the container and are identified by the interface(s) that they implement. So, we'll need some way to register HelloImpl with the service registry. For that, let's create HelloPublisher:

Download hello-service/src/main/java/com/pragprog/hello/service/impl/HelloPublisher.java

```
package com.pragprog.hello.service.impl;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;

import com.pragprog.hello.service.HelloService;

public class HelloPublisher implements BundleActivator {
   private ServiceRegistration registration;

   public void start(BundleContext context) throws Exception {
      registration = context.registerService(HelloService.class.getName(),
                          new HelloImpl(), null);
   }

   public void stop(BundleContext context) throws Exception {
      registration.unregister();
   }
}
```

HelloPublisher is a bundle activator, much like the HelloWorld activator we created earlier. This activator, however, uses the BundleContext that it is given to register an instance of HelloImpl as a service. It does this by calling the BundleContext's registerService() method, passing the service's interface (as the String returned from a call to the interface's class.getName() method), an instance of HelloImpl, and a set of service properties to associate with the service (which, for our purposes, can be null).

The last thing we need to create is the bundle's META-INF/MANIFEST.MF file:

Download hello-service/src/main/resources/META-INF/MANIFEST.MF

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: com.pragprog.HelloWorldService
Bundle-Name: HelloWorldService
Bundle-Version: 1.0.0
Bundle-Activator: com.pragprog.hello.service.impl.HelloPublisher
Import-Package: org.osgi.framework
Export-Package: com.pragprog.hello.service
```

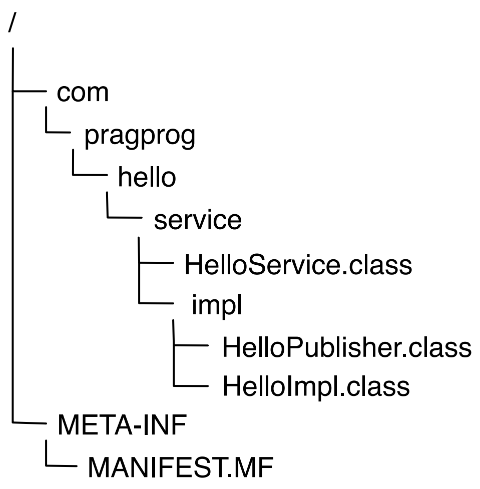

Figure 2.3: The structure of the HelloWorld service bundle

This bundle's manifest isn't dramatically different from the manifest we created before, but there is one new header to take note of. The Export-Package header publishes the contents of one or more packages for other bundles to use. Here, we've exported the com.pragprog.hello.service package so that consumers of our service can see and use the HelloService interface.

What's particularly interesting about Export-Package is the package that it doesn't export. Specifically, we're not exporting the com.pragprog.hello.service.impl package. That's because the service's implementation (and HelloPublisher, for that matter) are implementation details that are best kept secret. By not exporting them, we're effectively declaring them to be private, or unpublished. This prevents undesirable coupling that may occur if another bundle were to try to use HelloImpl directly instead of through its interface.

Now we're ready to compile and package everything up in a JAR file. In Figure 2.3, we can see the structure of the bundled JAR.

Finally, let's install it in Equinox:

```
osgi> install file:target/hello-service-1.0.0.jar
Bundle id is 2

osgi> ss
```

```
Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       ACTIVE      com.pragprog.HelloWorld_1.0.0
2       INSTALLED   com.pragprog.HelloWorldService_1.0.0

osgi>
```

The service bundle is now installed, alongside our first *Hello World* bundle that we deployed earlier. But the service won't be of any use to us until we start the bundle:

```
osgi> start 2

osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       ACTIVE      com.pragprog.HelloWorld_1.0.0
2       ACTIVE      com.pragprog.HelloWorldService_1.0.0

osgi>
```

When the service bundle is started, Equinox will invoke the start() method in HelloPublisher, consequently publishing the service in the service registry. To prove that the service has been published, we can issue Equinox's bundle command:

```
osgi> bundle 2
file:target/hello-service-1.0.0-SNAPSHOT.jar [2]
  Id=2, Status=ACTIVE
      Data Root=/Users/wallsc/osgi/configuration/org.eclipse.osgi/bundles/2/data
  Registered Services
    {com.pragprog.hello.service.HelloService}={service.id=21}
  No services in use.
  Exported packages
    com.pragprog.hello.service; version="0.0.0"[exported]
  Imported packages
    org.osgi.framework; version="1.4.0"<System Bundle [0]>
  No fragment bundles
  Named class space
    com.pragprog.HelloWorldService; bundle-version="1.0.0"[provided]
  No required bundles

osgi>
```

Notice that our service is found under the *Registered Services* heading. Also, notice that com.pragprog.hello.service is under the *Exported packages* heading.

Before we move on, it's worth noting that although HelloPublisher makes liberal use of the OSGi API in order to publish the service, HelloImpl and HelloService are completely OSGi-free.

At this point, the service has been deployed, but nobody is using it. Our original *Hello World* bundle is still in the container, but its activator is still printing hard-coded greetings. Let's put the service bundle to work by giving it a client.

Consuming the Service

Rather than create a new bundle to consume the hello service, let's revisit the original HelloWorld activator that we created earlier and change it to use the HelloWorld service:

Download hello-consumer/src/main/java/com/pragprog/hello/HelloWorld.java

```java
package com.pragprog.hello;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;

import com.pragprog.hello.service.HelloService;

public class HelloWorld implements BundleActivator {

   public void start(BundleContext context) throws Exception {
      HelloService helloService = getHelloService(context);
      System.out.println(helloService.getHelloMessage());
   }

   public void stop(BundleContext context) throws Exception {
      HelloService helloService = getHelloService(context);
      System.out.println(helloService.getGoodbyeMessage());
   }

   private HelloService getHelloService(BundleContext context) {
      ServiceReference ref = context.getServiceReference(HelloService.class
                        .getName());

      HelloService helloService = (HelloService) context.getService(ref);
      return helloService;
   }
}
```

This new HelloWorld activator is a bit more interesting than the first one. Rather than printing hard-coded greetings, the new start() and stop() methods use the HelloService returned from getHelloService(). The getHelloService() method is just a convenience method that looks up the service in the OSGi service registry. It does this by using the service's class

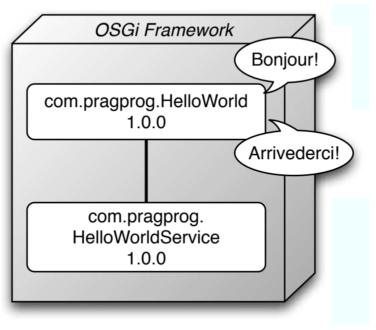
OSGi Framework
Bonjour!
com.pragprog.HelloWorld 1.0.0
Arrivederci!
com.pragprog. HelloWorldService 1.0.0

Figure 2.4: The new service-oriented HelloWorld activator relies on a service (deployed in a separate bundle) to provide its greetings.

name to get a service reference. With the service reference in hand, it then asks the BundleContext for the service, as shown in Figure 2.4.

It may not be apparent at first glance, but the getHelloService() method is rather naive. In OSGi, services can come and go as bundles are installed, updated, started, stopped, and uninstalled. What will the BundleContext's getService() return if the service's bundle isn't active or installed? As it turns out, if the service isn't available, then getHelloService() will return null, and the calls to getHelloMessage() and getGoodbyeMessage() will fail in a splendid fashion with a NullPointerException.

For now we'll just pretend that the service will always be available. We'll examine some strategies for dealing with missing services more gracefully in Chapter 5, *OSGi Services*, on page 82.

The only thing left to do is modify the manifest to account for the changes made to the HelloWorld activator. Since the activator now uses the HelloService interface, we must add its package to the Import-Package header:

Download hello-consumer/src/main/resources/META-INF/MANIFEST.MF

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: com.pragprog.HelloWorld
Bundle-Name: HelloWorld
Bundle-Version: 1.0.1
Bundle-Activator: com.pragprog.hello.HelloWorld
Import-Package: org.osgi.framework,
 org.osgi.util.tracker,
 com.pragprog.hello.service
```

In addition to importing the service's interface package, I have also bumped up the Bundle-Version to 1.0.1, just to indicate that this is a slightly different bundle than the one we've already installed.

All the pieces are now in place. To see it in action, first compile and JAR up the bundle, and then install it to Equinox. Assuming that the new version of the JAR file is in the same location as before, we can issue the update command:

```
osgi> update 1
Goodbye World!
Bonjour!
```

A lot of stuff happens when we ask Equinox to update the bundle. It first stops the bundle—that's why we see the "Goodbye World!" message. Then it uninstalls the old bundle and reinstalls the new bundle from the original location. Finally, it starts the bundle, resulting in the hello message being printed. And, it does all of this without having to restart Equinox!

Did you notice that the hello message is now "Bonjour!"? That proves that the activator is using the service and not simply printing the old hard-coded "Hello World!" message. If you need any further evidence that the bundle has been updated, check the status by issuing the ss command:

```
osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       ACTIVE      com.pragprog.HelloWorld_1.0.1
2       ACTIVE      com.pragprog.HelloWorldService_1.0.0

osgi>
```

The thing to spot is that the version number is now 1.0.1 and not 1.0.0 as it was previously.

For proof that the activator is actually using the service, the bundle command again comes in handy:

```
osgi> bundle 1
file:../hello/target/hello-activator-1.0.0.jar [1]
  Id=1, Status=ACTIVE
       Data Root=/Users/wallsc/osgi/configuration/org.eclipse.osgi/bundles/1/data
  No registered services.
  Services in use:
    {com.pragprog.hello.service.HelloService}={service.id=21}
  No exported packages
  Imported packages
    org.osgi.framework; version="1.4.0"<System Bundle [0]>
    org.osgi.util.tracker; version="1.3.3"<System Bundle [0]>
    com.pragprog.hello.service; version="0.0.0"
                <file:../hello-service/target/hello-service-1.0.0.jar [2]>
  No fragment bundles
  Named class space
    com.pragprog.HelloWorld; bundle-version="1.0.1"[provided]
  No required bundles

osgi>
```

If you look under the *Services in use:* heading, you'll find that this bundle is using the service published by the service bundle. And, it imports the com.pragprog.hello.service package that is exported by the service bundle.

We've seen the hello message. Now let's complete the story by stopping the bundle and seeing the goodbye message:

```
osgi> stop 1
Arrivederci!

osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.4.0.v20080605-1900
1       RESOLVED    com.pragprog.HelloWorld_1.0.1
2       ACTIVE      com.pragprog.HelloWorldService_1.0.0

osgi>
```

With that, we conclude our first adventure in OSGi. Although we've kept things simple, we've covered a lot of ground. We've become acquainted with two different OSGi containers (Equinox and Felix). We've also deployed a simple *Hello World* bundle in an OSGi container and seen it in action. And, we've expanded the *Hello World* example to be split across two bundles, one consuming a service published by the other. All of this serves as the basis for more OSGi adventure to come.