

What is OSGi?

OSGi is the Open Service Gateway initiative. It is a specification that describes a system which provides a dynamic component model. In a previous lab, you were tasked with implementing something very similar, though vastly less feature-rich. In addition, you also actually used OSGi when you developed your Eclipse plug-in, as the Eclipse platform is built on OSGi. Equinox, the OSGi implementation developed by the Eclipse team, is considered to be the reference implementation of the standard. There are several other implementations, though they tend to not be fully compliant. Apache Felix, for instance, implements only a subset of the specification.

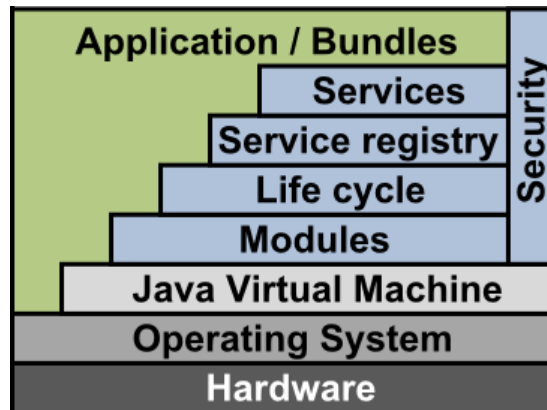


Figure 1: The Architecture of OSGi

OSGi applications are built upon bundles. On a basic level, a bundle is just a normal jar with a few extra manifest headers. You could quite easily take any bundle and import classes from it like any other jar if you use javac to compile your program. When properly run through OSGi, however, the only parts of a bundle you see are the parts that are explicitly exposed to you. The bundle may have 100 public classes, but only one is exposed. Within the bundle itself, interacting with the other classes is as normal.

One particularly interesting feature of OSGi is the support for multiple versions of the same bundle being run at the same time. This solves a huge dependency issue where one bundle X depends on version 1 of bundle Y and bundle Z depends on version 2 of bundle Y. In an ideal world, we would be able to update all dependent bundles when new versions are released, but the world is not ideal.

Hello World!

Now it's time to get down to business and write write some code! Fire up that dreaded beast, Eclipse. Being the provider of the reference OSGi implementation, Eclipse is pretty well geared up to do OSGi dev work.

Creating your first bundle

Creating the template for a bundle is pretty easy, and in fact it's a lot like creating an eclipse plugin template (because an eclipse plugin IS a bundle).

1. Click on **File** → **New** → **Project**.
2. Select **Plug-in Project** and click **Next**.
3. For project name, put `edu.rosehulman.HelloOSGi`. For target platform select **OSGi framework** → **Standard**

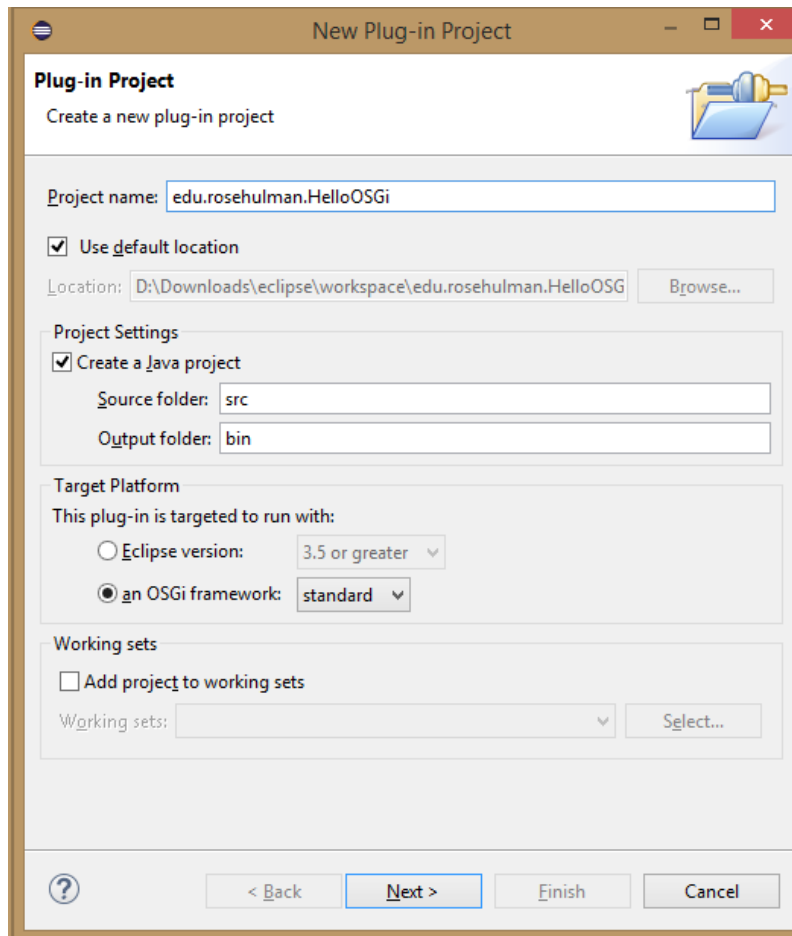
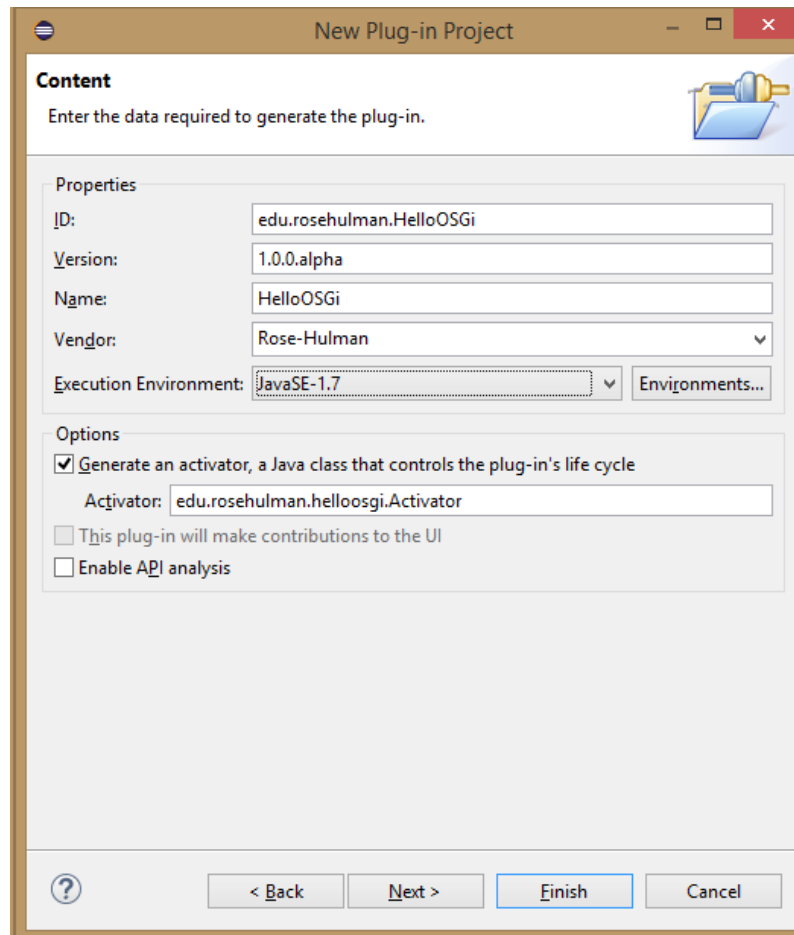


Figure 2: It should look something like this.

4. Click **Next**.
5. Enter the following values:
 - ID: `edu.rosehulman.HelloOSGi`

- Version: 1.0.0.alpha
- Name: HelloOSGi
- Vendor: Rose-Hulman
- Execution Environment: JavaSE-1.8 (Note that even though the picture shows 1.7, please use JavaSE-1.8)



New Plug-in Project

Content
Enter the data required to generate the plug-in.

Properties

ID:

Version:

Name:

Vendor:

Execution Environment:

Options

☒ Generate an activator, a Java class that controls the plug-in's life cycle
Activator:

☐ This plug-in will make contributions to the UI

☐ Enable API analysis

Figure 3: Now it should look like this.

6. Click **Next**.
7. Select the **Hello OSGi Bundle** and click **Finish**

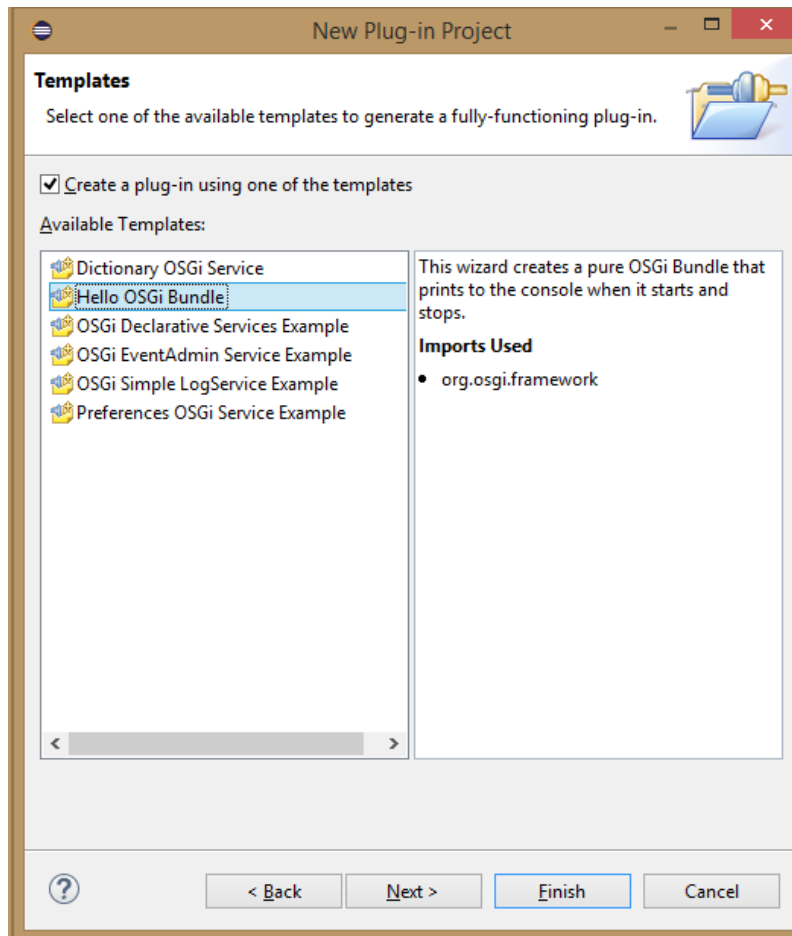


Figure 4: That one.

The Activator

When you generate an OSGi plug-in, Eclipse creates for you three files: `Activator.java`, `MANIFEST.MF`, and `build.properties`. `Activator.java` will come in looking like this:

```

package edu.rosehulman.helloosgi;

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

public class Activator implements BundleActivator {

    public void start(BundleContext context) throws
        Exception {
        System.out.println("Hello World!!");
    }

    public void stop(BundleContext context) throws
        Exception {
        System.out.println("Goodbye World!!");
    }

}

```

Figure 5: Plugin Entry Point.

You can easily guess what those methods are for. OSGi bundles have a very simple life cycle. When started, a `BundleContext` object is passed in, providing access to various OSGi-container services and information. You typically use the `start` to setup any necessary resources, and then use `stop` to clean them up. Easy peasy.

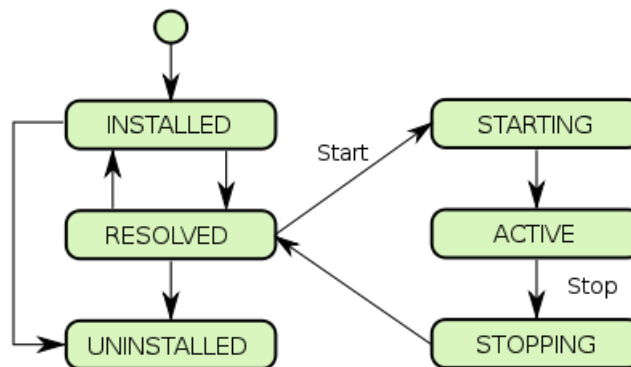


Figure 6: See, simple

The Manifest

And then you have the `MANIFEST.MF`. This file lists out a set of headers defined by the OSGi specification. You won't have to modify this directly if you're using Eclipse, since it has forms that make things simpler for you, but go ahead and make sure it looks something like this:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: HelloOSGi
Bundle-SymbolicName: edu.rosehulman.HelloOSGi
Bundle-Version: 1.0.0.alpha
Bundle-Activator: edu.rosehulman.helloosgi.Activator
Bundle-Vendor: Rose-Hulman
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: org.osgi.framework;version="1.8.0"
```

Figure 7: MANIFEST.MF

Running the bundle

Eclipse has Equinox embedded, which makes it relatively easy to run bundles. Don't just click run though! Chances are, all kinds of errors will pop up. This is because, in its infinite wisdom, Eclipse has decided to include every single environment bundle it knows about by default. This, as you might imagine, doesn't not play out terribly well. Lucky for you, the slings and arrows of outrageous dependencies have been endured, and a working subset has been discovered. Here's how you do it:

1. Click **Run** → **Run Configuration**
2. You should see OSGi Framework in the list on the left. Right click it and then click **New**.
3. Set Name to OSGi or whatever you want.
4. Click **Deselect All**
5. Select edu.rosehulman.HelloOSGi and the following bundles:
 - org.apache.felix.gogo.command
 - org.apache.felix.gogo.runtime
 - org.apache.felix.gogo.shell
 - org.eclipse.equinox.console
 - org.eclipse.osgi
 - org.eclipse.osgi.compatibility.state

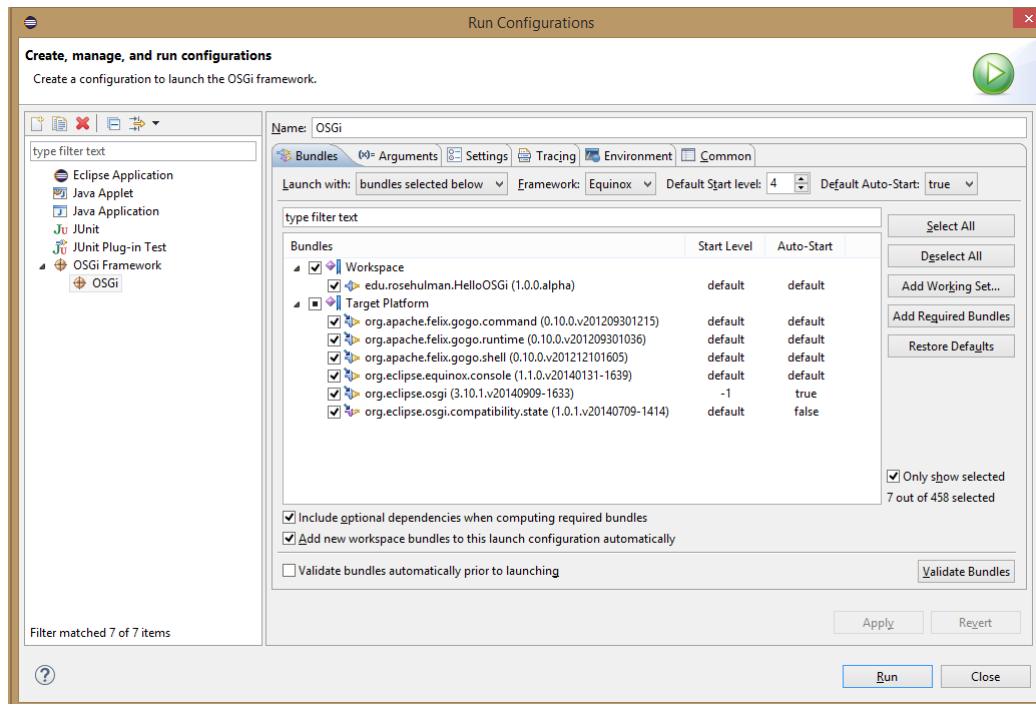


Figure 8: Run Configuration

Now feel free to click **Run**. You should see "Hello World!!" followed by the OSGi console prompt.

The OSGi console

The OSGi console is a command-line interface to various OSGi container functions. Here are a few commands that you may find useful:

- `ss` - displays a list of installed bundles with their statuses.
- `start <bundleId>` - starts a bundle
- `stop <bundleId>` - stops a bundle

Go ahead and try them out. Stop and restart the bundle you just created a time or two, you should see the relevant messages being printed to the console.

Bundles as a service

The star of the show that is OSGi is the ability to break your code into a set of modules and then manage their dependencies. As mentioned previously, all of the code of a bundle is hidden from other bundles unless you explicitly allow it to be exported. So lets make a simple service to showcase that!

Exposing code

1. Create the service `edu.rosehulman.GreetingService` in a similar way to how you created `edu.rosehulman.HelloOSGi`. Except this time, we won't be creating a default Activator and we won't be selecting a template.

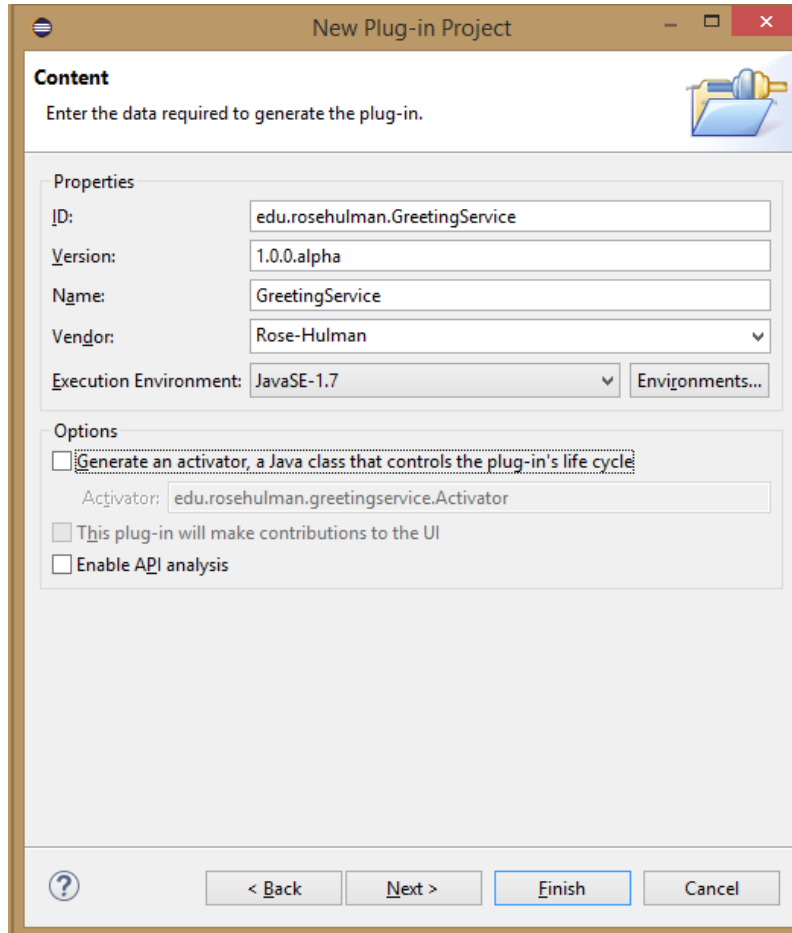


Figure 9: Go ahead and click **Finish** when you're done here.

2. Within the `src` folder, create the package `edu.rosehulman.GreetingService` (right click the folder then **New** → **package**)
3. Within that package, create `GreetingService.java` as shown in Figure 10


```

package edu.rosehulman.GreetingService;

public interface GreetingService {

    public String sayHello();

}

```

Figure 10: GreetingService.java

4. Now create the package `edu.rosehulman.GreetingService.impl`
5. Within `edu.rosehulman.GreetingService.impl`, create `GreetingServiceImpl.java` as shown in Figure 11

```

package edu.rosehulman.GreetingService.impl;

import edu.rosehulman.GreetingService.GreetingService;

public class GreetingServiceImpl implements
    GreetingService {

    @Override
    public String sayHello() {
        System.out.println("Help, I'm trapped inside
            GreetingServiceImpl!");
        return "Hello World!";
    }

}

```

Figure 11: GreetingServiceImpl.java

6. Open `MANIFEST.MF` and go to the **Runtime** tab. In **Exported Packages**, click **Add** and select `edu.rosehulman.GreetingService`. This tells OSGi that all the files directly within the package may be exported. Note that sub-packages are NOT exported.

As of now, any bundle that imports our new bundle will be able to see the `GreetingService` interface and use it freely. They will not, however, be able to use our implementation. Therefore we need to export a service on top of our interface.

Exposing services

In OSGi, a bundle registers a service for a particular interface. The source bundle can then ask OSGi for any services that are registered under that interface. When one is found, it gets bound and the source bundle can start using the functions.

1. Open MANIFEST.MF, under Dependencies (Imported Packages), add an import of
org.osgi.framework
2. In the package edu.rosehulman.GreetingService.impl create the class GreetingActivator.java as follows:

```
package edu.rosehulman.GreetingService.impl;

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

import edu.rosehulman.GreetingService.GreetingService;

public class GreetingActivator implements BundleActivator
{
    private ServiceRegistration greetingServiceRegistration;

    @Override
    public void start(BundleContext context) throws
        Exception {
        GreetingService greetingService = new
            GreetingServiceImpl();
        greetingServiceRegistration = context.registerService(
            GreetingService.class.getName(), greetingService,
            null);
    }

    @Override
    public void stop(BundleContext context) throws Exception
    {
        greetingServiceRegistration.unregister();
    }
}
```

Figure 12: GreetingActivator.java

BundleContext.registerService() takes three parameters:

- The name the service being registered.
 - The java object that implements the service.
 - Service properties. Can be used to decide between multiple implementations of a service.
3. Open MANIFEST.MF and register GreetingActivator.java as the bundle's activator. This is done in the **Overview** tab.

By now, the manifest of Greeting Service should look like Figure 13.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: GreetingService
Bundle-SymbolicName: edu.rosehulman.GreetingService
Bundle-Version: 1.0.0.alpha
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: org.osgi.framework;version="1.8.0"
Bundle-Vendor: Rose-Hulman
Export-Package: edu.rosehulman.GreetingService
```

Figure 13: Greting Service's MANIFEST.MF

Importing services

Now that we've had fun creating our service, it's time to use it! All it takes is a little fiddling around with the Activator and Manifest in HelloOSGi. First, modify MANIFEST.MF. Go to the **Dependencies** tab and add `edu.rosehulman.greetingservice` under **Imported Packages**. The manifest should now look like Figure 14. Now modify `Activator.java` to look like Figure 15.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: HelloOSGi
Bundle-SymbolicName: edu.rosehulman.HelloOSGi
Bundle-Version: 1.0.0.alpha
Bundle-Activator: edu.rosehulman.helloosgi.Activator
Bundle-Vendor: Rose-Hulman
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: org.osgi.framework;version="1.8.0"
Require-Bundle: edu.rosehulman.GreetingService;bundle-
    version="1.0.0"
```

Figure 14: Updated Manifest of HelloOSGi

```

package edu.rosehulman.helloosgi;

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

import edu.rosehulman.GreetingService.GreetingService;

public class Activator implements BundleActivator {
    private ServiceReference greetingServiceReference;

    public void start(BundleContext context) throws
        Exception {
        System.out.println("Hello World!!");
        greetingServiceReference = context.getServiceReference
            (GreetingService.class.getName());
        GreetingService greetingService = (GreetingService)
            context.getService(greetingServiceReference);
        System.out.println(greetingService.sayHello());
    }

    public void stop(BundleContext context) throws Exception
    {
        System.out.println("Goodbye World!!");
        context.ungetService(greetingServiceReference);
    }
}

```

Figure 15: Activator.java

`BundleContext.getServiceReference()` returns a `ServiceReference` object for some service registered as the `GreetingService` interface. If there were multiple services registered, it would return the service with the highest ranking.

You're all ready to run! Go ahead and give 'er a spin!

Credits

This tutorial is a product of collaboration in an independent study between an alumnus, Jordon Phillips and Dr. Chandan Rupakheti. It was adapted from the [Hello OSGi tutorial on JavaWorld](#). Information has been pared down to be more digestible and further detail was added in areas where ambiguity resulted in issues progressing.

Now it's your turn!

Now it's time for you to build your own OSGi application. Create a bundle that lists all of the files in the current working directory, use `“.”` for the directory location. The Hel-

loOSGi bundle should use the new bundle when it starts along with the GreetingService bundle.

Going Farther

OSGi is useful for more than just plugin development. The concept of a containerized solution is increasingly popular as distributed computing becomes more prevalent. You can, for instance, use OSGi to expose web services by embedding a small server, such as Jetty, inside. JavaWorld has a [good tutorial](#) on just that. And if you just want to do some distributed computing, OSGi can handle that too. In fact, distributed computing support is built right into the standard (version 4.2+). The reference implementation for distributed OSGi is [Apache CXF](#), which contains several examples if you're interested.