How to Write Your First JES Service

Java Embedded Server™
Version 2.0

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

October 2000, Revision 03

Send comments about this document to: jes-comments@sun.com

Contents

Preface

Welcome to the Java Embedded Server™ technology, version 2.0. This tutorial is for complete beginners who have never developed services or bundles for the Java Embedded Server framework before.

Congratulations. You have made a good decision to read this tutorial and work through the examples. When you finish this tutorial, you may enjoy reading the *JES Developer Guide* for more in-depth and detailed information.

Before You Do the Examples

Before you can work through the examples in this tutorial, you must have a make utility such as gnumake or nmake installed. To check that you have a make utility, move to the directory jes2.0/docs/tutorial/greeting1 and type gnumake (on Solaris) or nmake (on Windows).

If your computer creates a JAR file named greeting1.jar and stores it in jes2.0/docs/tutorial/build/bundles, you have the appropriate make utility. If your system displays an error message saying that it can't find the command, you need to install gnumake or nmake.

Installing gnumake

To install a prebuilt version of gnumake on Solaris, follow these steps:

1. **Go to the website** http://www.sunfreeware.com**.**
2. **Choose your version of Solaris in the upper right frame.**
3. **Choose the** gnumake **utility in the lower right frame, for example,** make-3.78.1**.**
4. **Click the link to the downloadable binary for** make-3.78.1**.**

5. **Enter a pathname in the file download box, but make sure the pathname ends with a filename ending with the** .gz **suffix.**

6. **After the download is finished, unzip the file:**

   gunzip *filename*.gz

7. **Log in as** root**:**

   su -l root
   enter password

8. **To install** gnumake **in** /usr/local/**, use this** pkgadd **command and answer the questions it asks:**

   pkgadd -d *filename*

9. **To install** gnumake **in** /usr/local**, use this** pkgadd **command:**

   pkgadd -a none -d *filename*

10. **Enter a base directory name for installing the** gnumake **utility when** pkgadd **prompts you:**

    Enter path to package base directory [?,q]

    The gnumake utility will be stored in *basedirectory*/bin/make. The directories *basedirectory*/doc/make and *basedirectory*/info contain documentation.

Installing nmake

To install and build nmake on Windows NT, follow these steps:

1. **Go to the FTP site** ftp://ftp.microsoft.com/Softlib/MSLFILES/nmake15.exe**.**

2. **In the** Save As ... **dialog box, save** nmake15.exe **in a folder of your choice.**

3. **Open that folder and double-click** nmake15.exe **to install it.**

   The files it unpacks are nmake.exe, nmake.err, and Readme.txt.

CHAPTER 1

Writing Your First JES Service

So you want to write a service and create a bundle for the Java Embedded Server™ (JES) home gateway. This tutorial, complete with code examples, teaches you how to write your first simple services, build them into bundles, and run them on the JES framework.

This chapter continues an illustrious computer science tradition by presenting the *Hello, World* of services and bundles—although we say *Hello, World* with several different variations. Use the examples here with the JES framework—compile them, install them, and run them. Then, modify them and rerun them to see the new results.

The JES Framework

Before you begin, you need to understand the foundation of the Java Embedded Server, the JES framework. The JES framework is written entirely in Java™ and so must run on a Java virtual machine.

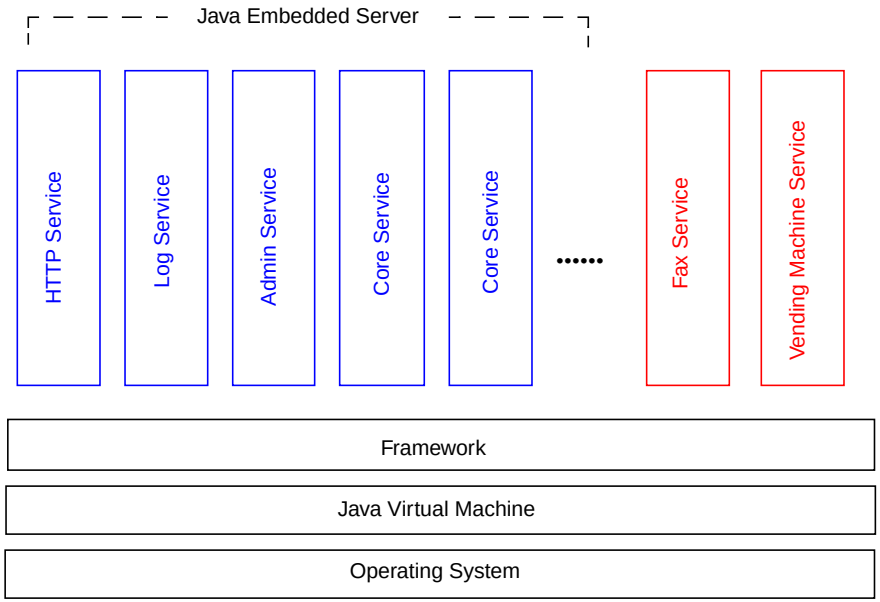
Java Embedded Server is a set of core services, such as LogService and HttpService, running on top of the framework. You can develop any service you need (for example, a FaxService and a VendingMachineService), and they can work with the existing JES core services.

Looking at FIGURE 1-1, you can see that the JES framework is the hosting platform on which services are run. A **service** is a group of Java classes and interfaces that implement a certain feature. For example, the core HTTP service that is shipped with JES creates a tiny Web server that can respond to requests from HTTP clients. A vending machine service, on the other hand, might examine the machine's internal temperature, set prices for merchandise, and dispense soda cans.

While you are developing services, you run the JES framework on your development platform. Later, you deploy your services on the JES framework inside a home gateway server.

Once you write your service, you package it into a **bundle** before deploying it on the framework. A bundle is a JAR file containing the service and any other files, images, or resources it might need. A bundle can also contain several services that work together as an integrated unit. Like any other JAR file, a bundle contains a Manifest object (based on the java.util.jar.Manifest class) that describes the contents of the JAR file.

FIGURE 1-1 A Look into the JES Architecture



The JES framework manages the services it hosts. Specifically, it registers services, handles the bundle life cycle, tracks dependencies among services, and sends event notifications. In order to manage bundles and services, the JES framework creates a BundleContext object, which allows the framework and the bundle to interact. We'll talk about the BundleContext object in just a moment, when we can see it in some actual code.

The JES framework complies with and extends the *OSGi Service Gateway 1.0 Framework Specification* developed by the Open Services Gateway Initiative. If you haven't already, you can get a copy of the OSGi specification, including its Javadoc API reference, from http://www.osgi.org. You'll see references to the specification throughout this tutorial.

The Simplest Bundle You Can Create

Every bundle needs at least a bundle activator class and a manifest file. A bundle activator class is a Java class that implements org.osgi.framework.BundleActivator and defines logic for the start and stop methods. The JES framework uses start and stop to start and stop the bundle.

In this example, we create a simple bundle that does not have any services, but simply displays *Hello* when started and *Bye* when stopped. You'll follow these steps:

1. **Write a bundle activator class in Java.**
2. **Create a** Manifest **file as a text file.**
3. **Build a JAR file that contains the compiled activator class and the** Manifest **file.**
4. **Start the JES framework, and install, run, and stop the bundle.**

Later, to create more complex bundles, you will write services that handle framework events and use the core JES services or you will add functionality to the bundle activator class.

The BundleContext Object

Now it's time to talk about the BundleContext object, which allows a bundle to interact with the JES framework.

The framework creates a BundleContext object when it starts a bundle and passes the object to the bundle activator's start method. The methods in the BundleContext object allow the bundle to install itself, register services, get information about other installed bundles, retrieve references to services, get and release service objects, get and release Bundle and File objects, and subscribe to events published by the framework. The BundleContext object is described in more detail later in this guide.

For now, you need to know these facts about the BundleContext object:

- Only the JES framework, never bundles, can create it.
- It's passed to the start method when the bundle is activated.
- It's also passed to the stop method when the bundle is stopped.

Let's now look at how the bundle activator class is written.

The Bundle Life Cycle

A bundle has a life cycle, during which it can be installed, activated, updated, deactivated, and uninstalled. It's time that we examine this chain of events at a high level (if you're interested in more detail, you can look up the *OSGi Service Gateway Specification, Release 1.0*).

The steps of the life cycle run like this:

1. **The bundle is installed.**

   The JES framework reads the contents of the bundle, assigns it a bundle ID, and caches its location and state persistently. A dedicated class loader is created to access the bundle's resources.

2. **The bundle is activated.**

   The framework checks whether the Java classes the bundle requires have been exported by other bundles. If so, the framework calls the `start` method in the bundle's activator class and registers the bundle's services. At this point, the bundle's services start running.

   The JES framework also resolves dependencies among bundles at this step, ensuring that bundle A is present before it starts bundle B, if B depends on A.

3. **The bundle is deactivated.**

   The framework calls the bundle's `stop` method, which unregisters the service. The service stops running.

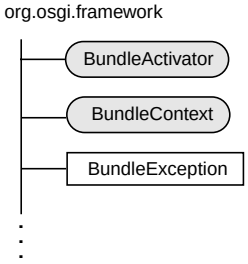
4. **The bundle is uninstalled.**

   The bundle is removed from the computer.

If an error occurs during the bundle life cycle, the JES framework throws a `BundleException`, which is a special type of exception that occurs only during the bundle life cycle. `BundleException` is a standard exception defined by the OSGi specification (in `org.osgi.framework`) that JES implements.

Step 1: Write the Bundle Activator Class

The greeting1 bundle has a very simple bundle activator class, Activator.java, that only defines start and stop methods.



CODE EXAMPLE 1-1 A Simple Bundle Activator Class (greeting1/Activator.java)

```
❶  package greeting1;

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

❸  public class Activator implements BundleActivator {

❹    public void start(BundleContext context) throws BundleException {
       System.out.println( "Hello" );
     }

❺    public void stop(BundleContext context) throws BundleException {
       System.out.println( "Bye" );
     }

   }
```

The Activator class is stored in the package named greeting1 ❶. It imports the BundleActivator, BundleContext, and BundleException classes from org.osgi.framework individually ❷, rather than importing the entire org.osgi.framework package, reducing the amount of memory used when the JES framework loads the bundle activator.

Activator implements the BundleActivator interface ❸ and defines its start and stop methods, as all bundle activator classes should. The start method ❹ is implemented to throw a BundleException, rather than a java.lang.Exception as the method is declared in

BundleActivator, because BundleException (a subclass of java.lang.Exception) is the exception type the JES framework throws for a bundle life cycle problem. This implementation of start displays *Hello* when the bundle is started.

The stop method ❺ also throws a BundleException and displays *Bye* when the bundle is stopped.

Step 2: Create the Manifest File

Every bundle that you install in the JES framework contains a Manifest file, a standard text file that describes the contents of the JAR file. The Manifest file is structured in headers, and each header has an attribute. For an example, look at the file greeting1/Manifest.

```
BundleActivator: greeting1.Activator
```

This Manifest file has only one header. It tells the JES framework which Java class to use as the bundle activator class.

The headers provide the JES framework with "hooks" to the files and resources the bundle contains. This way, the framework knows where to find resources (such as the bundle activator) within the bundle as the bundle moves through its life cycle.

The OSGi specification defines headers (shown in TABLE 1-1) that can be used in the Manifest file for a JES bundle. But TABLE 1-1 just shows the names of the headers. Each header has a defined syntax that you must use to specify its attribute. You don't need the syntax details to complete this tutorial, but if you're interested, you can get them from the *Java Embedded Server 2.0 Developer Guide*.

TABLE 1-1 The Headers You Can Use in a Bundle Manifest File

Bundle-Activator	**Bundle-DocURL**	**Bundle-Version**
Bundle-ClassPath	**Bundle-Name**	**Export-Package**
Bundle-ContactAddress	**Bundle-NativeCode**	**Export-Service**
Bundle-Description	**Bundle-UpdateLocation**	**Import-Package**
	Bundle-Vendor	**Import-Service**

Step 3: Create the Bundle JAR File

Now that you have a bundle activator class and a Manifest file, you are ready to create a bundle. You create the bundle using a makefile written in gnumake as shown below.

1. **Move to the** greeting1 **directory:**

   ```
   % cd jes2.0/docs/tutorial/greeting1
   ```

2. **Build the JAR file:**

   ```
   % gnumake
   ```

3. **Check for the JAR file:**

   ```
   % cd ../build/bundles
   % ls greeting1.jar
   ```

Step 4: Install and Run the Bundle

1. **Open a terminal window and move to your JES 2.0 directory:**

   ```
   % cd
   % cd jes2.0
   ```

2. **Start the JES framework:**

   ```
   % runjes
   ```

   When the JES framework starts, you'll see its command prompt:

   ```
   Java Embedded Server 2.0
   Copyright 1998, 1999 and 2000 Sun Microsystems, Inc.
   All rights reserved. Use is subject to license terms.

   Type 'h[elp]' for a list of commands.
   >
   ```

3. **With a Web browser, open the JES Tools Portal at** jes2.0/index.html**.**

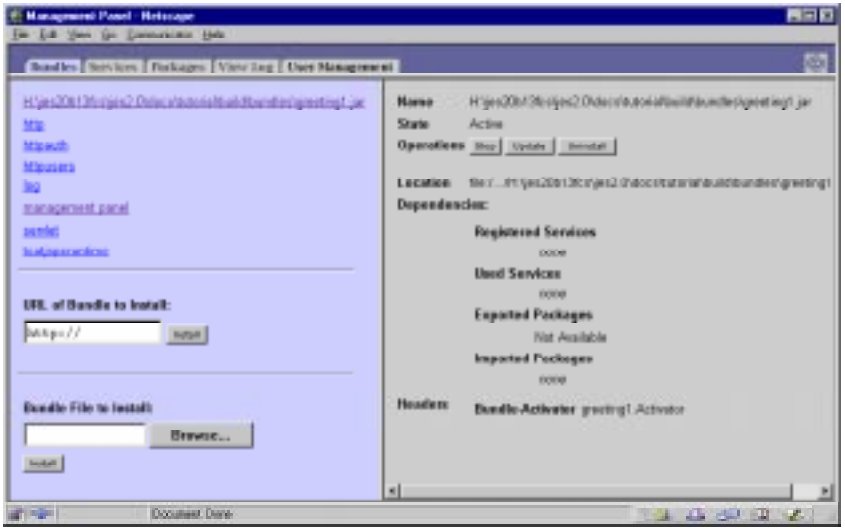
4. **In the Tools Portal, click** Management Panel **to open the JES Management Panel.**

5. **In the JES Management Panel, click the** Bundles **tab.**

6. **In** Bundle File to Install**, browse to the** greeting1 **bundle (it's at** jes2.0/docs/tutorial/build/bundles/greeting1.jar**), then click** Install**.**

7. When you see greeting1 **listed in the right pane, click** Start**.**

FIGURE 1-2 The JES Management Panel with greeting1 Installed



Adding an Interface and Service

Services are designed to be written with interface and implementation separated. When interface and implementation are separated, the interface is exposed to outside callers, such as the JES framework or other services. This means that the methods that are available, their arguments, return values, and exceptions are clearly documented. The interface acts as a contract between the callers and the service.

The implementation of the interface, on the other hand, is private to the service. You have complete flexibility in how to code your service, provided that it meets the specification the interface provides. You can create more than one implementation for the same interface.

Separating interface and implementation makes you think harder during the design stage. The service interface must have the minimum number of methods, no more, no less. Because the interface decides how services are coupled together, once it is defined, it should remain unchanged. It would be very costly to change a service interface when many other services have come to rely on it. This is one of the major challenges you face in developing for the JES framework.

Step 1: Write an Interface

When you write a service for the JES framework, you can implement an interface that is provided by the JES or OSGi APIs. You can also write your own interface, as shown in CODE EXAMPLE 1-2.

CODE EXAMPLE 1-2 The GreetingService Interface (greeting2/service/GreetingService.java)

```
❶ package greeting2.service;

❷ public interface GreetingService {


❸ /**
  * Prints a form of greeting to the stdout for the named
  * individual.
  * @param firstName First name of the person
  * @param lastName Last name of the person
  * @param title Title of the person, e.g., Mr./Ms./Sir
  */


❹     public void greet(String firstName, String lastName, String title);
  }
```

The interface in CODE EXAMPLE 1-2 is named GreetingService ❷. It belongs to the greeting2.service package ❶ and is placed in the docs/tutorial/greeting2/service directory. The greet method ❹ is declared to return a greeting that you define when you implement it. It requires three parameters: the individual's first name, last name, and title.

In this example, greet is documented with Javadoc comments ❸. It's a good idea to add Javadoc comments to any interface you write, so that a fellow developer implementing it knows exactly what values the method accepts and returns.

Step 2: Write a Casual Implementation

As you know, cultural and social circumstances usually dictate how people greet each other. Here we write an implementation of GreetingService for a casual setting.

CODE EXAMPLE 1-3 Implementing GreetingService (greeting2/impl/CasualGreetingImpl.java)

```
❶  package greeting2.impl;

❷  import greeting2.service.GreetingService;

❸  class CasualGreetingImpl implements GreetingService {

❹       public void greet( String firstName, String lastName, String title ) {
            System.out.println ("Hey, I'm Scott. Nice to meet you, " + firstName);
        }
}
```

Notice that the implementation class is stored in a different package and directory than the interface classes. In CODE EXAMPLE 1-3, the implementation class is placed in the package greeting2.impl ❶ and the directory jes2.0/docs/tutorial/greeting2/impl. This is so that you can expose the interface classes (in greeting2.service) to other bundles without exposing the implementation.

After defining the package, the next steps are to import the GreetingService interface ❷ and declare the class ❸. Notice that the implementation class has default package access, while the GreetingService interface has public access. This means that other bundles can call the interface, but not the implementation.

The next step is to implement the greet method ❹ to display a casual greeting. Notice that the declaration of greet matches its method signature in the interface.

Step 3: Write the Bundle Activator Class

You now need to write a bundle activator class. The bundle activator class is essential to any bundle, but in this bundle it must register our new service, CasualGreetingImpl, as well as start and stop the bundle. The bundle activator class uses two objects the JES framework creates—the BundleContext object that you are already familiar with and a ServiceRegistration object.

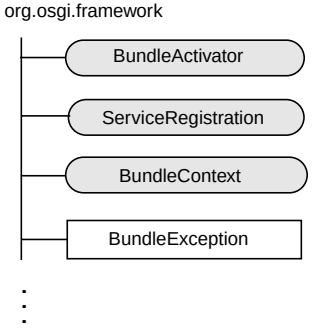
The ServiceRegistration Object

In your bundle activator class, you will register services using the registerService method that the BundleContext object defines. If the service registration is successful, the framework returns a unique ServiceRegistration object to the bundle. The ServiceRegistration object is always unique, even if the bundle registers the same service more than once.

The ServiceRegistration object lets other bundles get a reference to the service or update a service's properties (which you set with registerService when you register the service). You must use the ServiceRegistration object to unregister, so only the bundle that holds the ServiceRegistration object can unregister the service.

Registering a Service in a Bundle Activator Class

Now it's time to see how to use the BundleContext and ServiceRegistration objects in a bundle activator class.



CODE EXAMPLE 1-4 The Activator Class for greeting2 (greeting2/impl/Activator.java)

```
❶ package greeting2.impl;

❷ import java.util.Properties;
  import org.osgi.framework.*;
  import greeting2.service.GreetingService;
```

```
❸  public class Activator implements BundleActivator {

❹       private ServiceRegistration reg = null;

❺       public void start(BundleContext ctxt) throws BundleException {

❻           GreetingService greetingSvc = new CasualGreetingImpl();

            Properties props = new Properties();
❼           props.put("description", "casual");

❽           reg = ctxt.registerService("greeting2.service.GreetingService", greetingSvc, props);
        }

        public void stop(BundleContext ctxt) throws BundleException {
            if (reg != null)
❾               reg.unregister();
        }
    }
```

The bundle activator class is stored in greeting2.impl, the same package as the implementation classes ❶. You need to import java.util.Properties ❷, because you will create a Properties object in order to register the service. You also need to import the GreetingService interface ❷, because you will register your service under its interface name. Then, unlike the implementation classes, the BundleActivator class is declared public, so that the JES framework can call it ❸.

Next, you should prepare to register the service by creating a reference to a ServiceRegistration object ❹, in this case named reg. Once the service is registered, the framework will return a ServiceRegistration object that you can store in that object reference variable.

Now it's time to implement the start method and register the service ❺. Notice that the framework passes a BundleContext object to the start method. Registering the service takes a number of steps:

- Create an instance of your service and cast it to its interface type ❻, because you will register the service under its interface name.
- Create a Properties object and give the Properties object a key and a value ❼.
- Register the service ❽, using the fully qualified name for the interface GreetingService, the instance of the service you just created, and the Properties object you just created.
- Store the ServiceRegistration object the JES framework returns in reg, the object reference created in ❽.

You must also implement the stop method ❾, checking for a valid ServiceRegistration object, then using the ServiceRegistration object to unregister the service with the unregister method.

Step 4: Write the Manifest File

Let's say that in designing your bundle, you decide to offer its services for other bundles to use. To do this, you need to add the Export-Package header to the Manifest file (remember that the headers are listed in TABLE 1-1). Export-Package names the Java packages that the bundle offers to share with other bundles.

The advantage of storing implementation classes and interfaces in separate Java packages now becomes clear. When you export only the package that contains the interfaces, other bundles do not have access to the service's implementation.

Because a bundle always contains a bundle activator class, the Manifest file must also have a Bundle-Activator header.

CODE EXAMPLE 1-5 The Manifest File for greeting2 (greeting2/Manifest)

```
❶  Bundle-Activator: greeting2.impl.Activator
❷  Export-Package: greeting2.service
```

You can see in CODE EXAMPLE 1-5 that the Bundle-Activator header ❶ names your bundle activator using its full package name.

The Export-Package header ❷ names the interface package as the one to export and make available to other bundles. Now other bundles can call the GreetingService interface without seeing details of the implementation.

Step 5: Build and Run the Bundle

1. **Move to the right tutorial directory:**

   ```
   % cd
   % cd jes2.0/docs/tutorial/greeting2
   ```

2. **Build the JAR file:**

   ```
   % gnumake
   ```

The build command generates the JAR file jes2.0/docs/tutorial/build/bundles/greeting2.jar.

3. Start the JES framework:

```
% cd
% cd jes2.0
% runjes
```

4. In the framework window, install the bundle you just generated:

```
> install file:/jes2.0/docs/tutorial/build/bundles/greeting2.jar
```

5. Get the bundle ID for the bundle you have just installed, and start the bundle as in the previous example:

```
> bundles
> start bundleID
```

When you start the greeting2 bundle, no messages appear.

6. Examine the registered services.

```
> services
[greeting2.service.GreetingService]
    description=casual
```

The property description=casual was set with the Properties.put method in CODE EXAMPLE 1-4.

7. Examine the exported packages.

```
> exportedpackages

Package: greeting2.service (0.0.0)
    Exported by: 10 (file:/jes2.0/docs/tutorial/build/bundles/greeting2.jar)
```

This shows you that the Java package greeting2.service, version 0.0.0, has been exported from the bundle greeting2.

8. Stop the bundle:

```
> bundles
> stop bundleID
```

Writing a Formal Implementation

In a formal setting, we would want to write a different implementation of GreetingService that gives a more formal greeting. The formal implementation is easy to write. It's similar to CasualGreetingImpl.java, except that the implementation of the greet method is different.

Step 1: Write the Implementation

Notice that this class, FormalGreetingImpl, implements GreetingService but belongs to a different package.

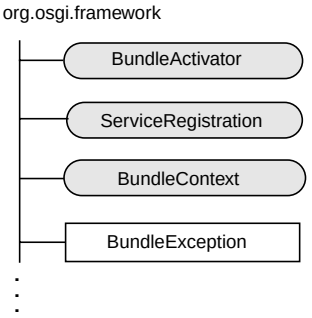
CODE EXAMPLE 1-6 A More Formal Greeting (greeting3/impl/FormalGreetingImpl.java)

```
❶ package greeting3.impl;

❷ import greeting2.service.GreetingService;


public class FormalGreetingImpl implements GreetingService {


    public void greet(String firstName, String lastName, String title) {
❸       System.out.println(
            "My name is Scott McNealy. How do you do, " + title + " " + lastName + "? "
                +   "It's a pleasure to make your acquaintance.");
    }
}
```

Just as CasualGreetingImpl.java did, FormalGreetingImpl.java belongs to a package of implementation classes ❶ that is separate from the interface classes ❷. FormalGreetingImpl.java also imports and implements the GreetingService interface ❷. The greet method is implemented as before, but this time it displays a more formal greeting ❸.

Step 2: Write the Activator Class

The new implementation must also have a new bundle activator class that is similar to the one shown in CODE EXAMPLE 1-4 but that registers our new service, FormalGreetingImpl.



CODE EXAMPLE 1-7 Registering FormalGreetingImpl (greeting3/impl/Activator.java)

```
❶  package greeting3.impl;

   import java.util.Properties;
❷  import org.osgi.framework.*;
   import greeting2.service.*;

   public class Activator implements BundleActivator {

       private ServiceRegistration reg = null;

       public void start( BundleContext ctxt ) throws BundleException {
❸
             GreetingService greetingSvc = new FormalGreetingImpl();


❹            Properties props = new Properties();
             props.put("description", "formal");

❺          reg = ctxt.registerService( "greeting2.service.GreetingService", greetingSvc, props);
       }

       public void stop(BundleContext ctxt) throws BundleException {
❻          if (reg != null)
               reg.unregister();
       }
   }
```

Just as with greeting2, the Activator class is stored in the implementation package ❶. The class imports the greeting2.service package ❷, because that is the package that contains the GreetingService interface.

The start method creates an instance of the new service, FormalGreetingImpl ❸, casting it to its interface type, GreetingService. The method then creates a new Properties object ❹, giving it a key and value (description and formal, which you can check from the JES framework later). The new service is then registered under the interface name GreetingService ❺, storing the ServiceRegistration object the framework returns in the object named reg.

The Activator class implements the stop method ❻ as well, using reg to call the unregister method.

Step 3: Write the Manifest File

Now let's look at the Manifest file this bundle needs.

CODE EXAMPLE 1-8 The Manifest File for greeting3 (greeting3/Manifest)

```
❶ Bundle-Activator: greeting3.impl.Activator
❷ Export-Package: greeting2.service
```

Notice that the greeting3 bundle exports the package greeting2.service, which includes GreetingService.java. The makefiles included in the examples (GNUmakefile for Solaris and makefile for Windows NT) package the same GreetingService interface in both the greeting2.jar and greeting3.jar bundles.

Since both bundles contain the package, they both export it. So if another bundle wants to share greeting2.service, does it share with bundle greeting2 or bundle greeting3? The JES framework is designed so that the bundle that is started first will export the package.

The best way to check which bundle has exported a package is with the exportedpackages command from the JES framework command line. For example, if you have both greeting2 and greeting3 registered, you might see this output:

```
> exportedpackages
Package: greeting2.service (0.0.0)
Exported by: 10 (file:/.../greeting3.jar)
Imported by: 11 (file:/.../greeting2.jar)
```

This output shows that greeting3 was registered first, exporting greeting2.service. The greeting2 bundle was registered later. It also exports greeting2.service, but its export is not available to other bundles, so the JES framework shows you that it imports the service.

Step 4: Build and Run greeting3

You must compile greeting2 before greeting3, because the FormalGreetingImpl class in greeting3 requires the GreetingService interface in greeting2. This logic is already coded into the makefiles that are included with the tutorial examples.

Here's what you do:

1. **Move to the** greeting3 **example directory:**

   ```
   % cd
   % cd jes2.0/docs/tutorial/greeting3
   ```

2. **Build the JAR file:**

   ```
   % gnumake
   ```

   The build command generates the JAR file jes2.0/docs/tutorial/build/bundles/greeting3.jar.

3. **Start the JES framework:**

   ```
   % cd
   % cd jes2.0
   % runjes
   ```

4. **In the JES framework window, start the services needed for the JES Management Panel:**

   ```
   > start log, servlet, http, tcatjspcruntime, httpauth, httpusers, jesmp
   ```

5. **From a Web browser, open the JES Tools Portal at** jes2.0/index.html**.**

6. **In the Tools Portal, click** Management Panel **to open the JES Management Panel.**

7. **In the login dialog box, enter** admin **for the user name and** admin **for the password, then click OK.**

8. **Next to Bundle File to Install, click Browse, locate the** greeting3 **bundle you just built at** jes2.0/docs/tutorial/build/bundles/greeting3.jar**, then click** Install**.**

9. **In the right pane, click Start.**

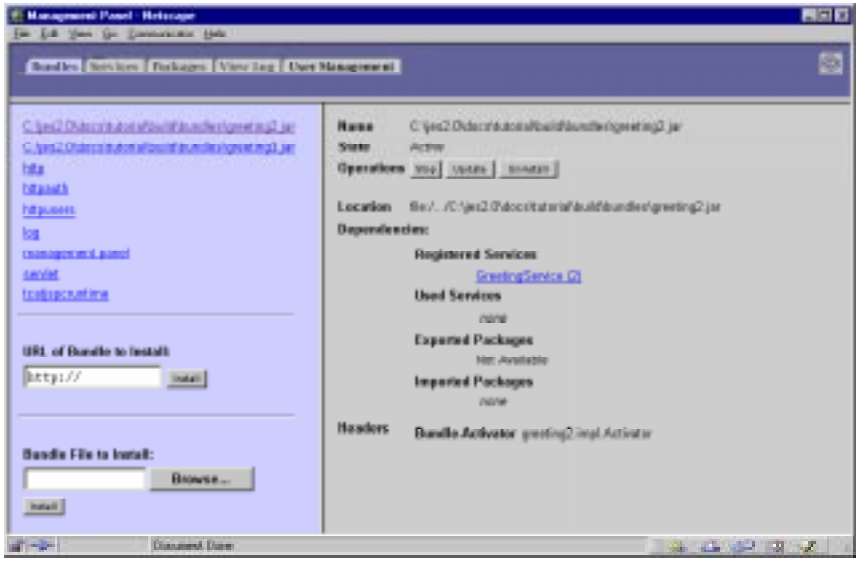
   The right pane shows the full pathname to the bundle, any services it depends on, and the name of its bundle activator class. Note that the Exported Packages feature is not available in this release of JES MP.

10. **Click the Services tab and examine both of the registered** GreetingService **instances and their descriptions.**

    One GreetingService has the property description **casual** and one has description **formal**.

11. **Click the View Log tab and notice what the framework has done with** greeting3.jar**.**

FIGURE 1-3 Installing the greeting3 Bundle Using JES MP



The Component Model

The Java Embedded Server architecture uses a component programming model, rather than the traditional library or toolkit model. Writing JES services is not simple object-oriented programming, because JES services and bundles have a richer semantics than classes written in the Java programming language.

FIGURE 1-4 A Typical Library-Based Architecture

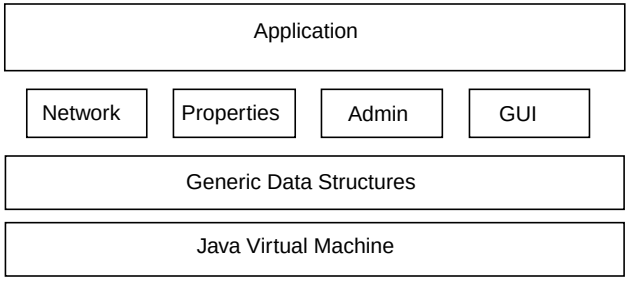
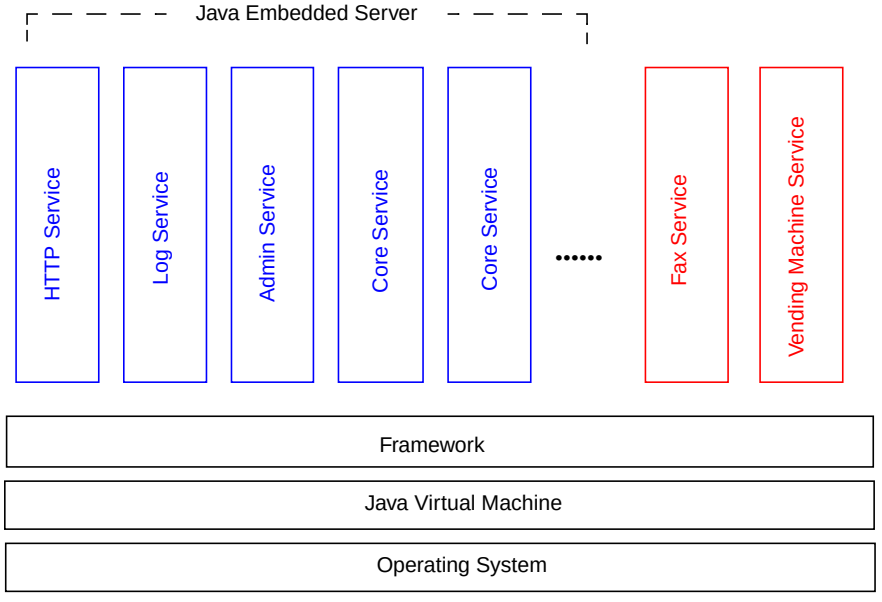


FIGURE 1-5 The JES Architecture *(repeated)*



Compare FIGURE 1-4, a software library architecture, with FIGURE 1-5, the JES architecture. You can see these differences:

- In a library-based model, multiple layers of software abstractions stack up one on top of another, while in a component-based model, multiple software components plug in side by side.
- In a library-based model, the sum of all libraries must be packaged together for the product to work, while in a component-based model, a subset of components can be packaged together and serve useful functionality as a product.
- In a library-based model, you must rebuild and repackage the entire set of libraries to fix bugs and add features during compile time, while in a component-based model, new components can be added or existing components updated in an incremental way during runtime.
- In a library-based model, problems in lower layers can propagate up and affect the stability of the entire software, while in a component-based model, the layers are insulated from one another.
- In a library-based model, changes made to public interfaces have less impact because you have to rebuild the entire software package anyway, while in a component-based model, the cost of rewriting a public interface can be prohibitive, because another component may have relied upon it at runtime.
- In a library-based model, it is easy to follow the flow of control because applications usually have one entry point (for example, `public static void main(String[ ] args)`), while in a component-based model, components are controlled by a hosting environment and interact with each other dynamically.

In conclusion, the component-based model requires more discipline on the part of developers during the design stage, but provides more reliability, extensibility, and flexibility over the library-based model during runtime.

Using One Bundle from Another Bundle

Because JES uses a component model in which components plug in to the framework side by side, one bundle can use a second bundle's services, provided the second bundle exports them. One bundle's service can be used by another more elaborate service so that you get richer functionality.

Suppose we have a bundle that enrolls new members in a club, maybe a chess club. In the process we want to use the casual implementation of GreetingService to show greetings to newcomers.

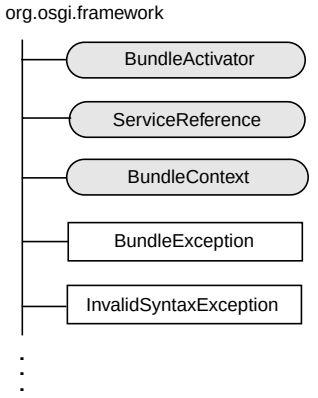
In this example, the club bundle depends upon the GreetingService. This example does not show the services of the club bundle. It only shows the Activator class that obtains the GreetingService by using a ServiceReference object.

The ServiceReference Object

A ServiceReference object is a reference to a service. If your bundle uses another service, it must first get a ServiceReference object and then use the ServiceReference object to get the service. Because you get the ServiceReference object first, you can examine the properties of the service before you use it. A registered service can have multiple, distinct ServiceReference objects referring to it. When a service has more than one reference, the references are considered to be equal.

Step 1: Write the Activator Class

You get the ServiceReference object by writing an Activator class, using the getServiceReference or getServiceReferences methods from the BundleContext object. The getServiceReference method returns one service reference, while getServiceReferences returns an array of references that match certain criteria. Because the framework is a dynamic environment with services constantly being registered, started, and stopped, you may want to use getServiceReferences, as shown in CODE EXAMPLE 1-9.



CODE EXAMPLE 1-9 Getting Service References in an Activator Class (club/Activator.java)

```
package club;

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

import greeting2.service.GreetingService;


public class Activator implements BundleActivator {

    public void start (BundleContext ctxt) throws BundleException {
        try {

❶           ServiceReference[ ] casualRefs = ctxt.getServiceReferences(
               "greeting2.service.GreetingService", "(description=casual)" );

❷           GreetingService greetingSvc = (GreetingService) ctxt.getService( casualRefs[0] );


❸           greetingSvc.greet ("Bill", "Gates", "Chairman");

❹       } catch (InvalidSyntaxException e) {
            throw new BundleException("Invalid LDAP filter", e);
        }
    }

    public void stop(BundleContext ctxt) throws BundleException {
    }
}
```

When the club bundle is started, it gets the references to all instances of the casual greeting service, CasualGreetingImpl, that are running on the JES framework ❶. The club bundle does this by using the getServiceReferences method with two parameters—the fully qualified name of the GreetingService interface, and an LDAP filter that specifies the properties used when CasualGreetingImpl was registered.

The method needs to use the LDAP filter, because we know that the JES framework has two services registered under the GreetingService interface. In this example, the LDAP filter parameter looks like "(description=casual)" ❶. The filter parameter to getServiceReferences has its own syntax rules that are described in the Javadoc API documentation for the BundleContext interface.

Once you have the array of ServiceReference objects, the bundle gets the casual greeting service ❷ by calling the getService method on the first element in the array and casting the service object to GreetingService, the interface type the service is registered with. Because GreetingService is registered with the framework and the club bundle declares that it imports GreetingService in its Manifest file, the JES framework is able to satisfy the request.

The start method then uses the service's greet method ❸ to display a casual greeting to a well-known club member when the bundle is started. The Activator class needs to handle syntax errors in the LDAP filter parameter, so the catch clause catches an InvalidSyntaxException and throws it with the message Invalid LDAP filter ❹.

Step 2: Write the Manifest File

Of course, this bundle also needs a Manifest file. The Manifest file needs to specify the name of the bundle activator class and any packages that the bundle depends on, in this case, greeting2.service.

CODE EXAMPLE 1-10 The Manifest File for club (club/Manifest)

```
❶  Bundle-Activator: club.Activator

❷  Import-Package: greeting2.service
```

The Import-Package header ❷ in the Manifest file specifies the Java package the club bundle requires. When the JES framework starts the bundle, the framework will ensure that the greeting2 bundle has exported greeting2.service. The framework will then activate the club bundle, using the specified Activator class ❶.

Example: Building and Running the club Bundle

1. **Move to the correct tutorial directory:**

```
% cd
% cd jes2.0/docs/tutorial/club
```

2. **Build the JAR file:**

```
% gnumake
```

The build command generates the JAR file jes2.0/docs/tutorial/build/bundles/club.jar.

3. **Start the JES framework:**

```
% cd
% cd jes2.0
% runjes
```

4. **In the framework window, install the bundle you just generated:**

```
> install file:/jes2.0/docs/tutorial/build/bundles/club.jar
```

5. **Get the bundle ID for the bundle you have just installed, and start the bundle as in earlier examples:**

```
> bundles
> start bundleID
```

You see the greeting:

```
Hey, I'm Scott. Nice to meet you, Bill
```

6. **Examine the exported packages:**

```
> exportedpackages
```

```
Package: greeting2.service (0.0.0)

  Exported by: 10 (file:/home/sahmed/kmakebuild/product/docs/tutorial/build/bundles/greeting2.jar)
  Imported by: 12 (file:/home/sahmed/kmakebuild/product/docs/tutorial/build/bundles/greeting3.jar)
               13 (file:/home/sahmed/kmakebuild/product/docs/tutorial/build/bundles/club.jar)
```

This shows that greeting2.service has been exported by greeting2.jar and imported by club.jar.

7. **Stop the bundle:**

```
> bundles
> stop bundleID
```