Java Embedded Server 2.0 Tutorial™

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

Part No. 8xx-xxxx-xx
August 2000, Revision 01

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Adobe PostScript

Contents

Preface vii

1

1. Java Embedded Server Tutorial 1

Concepts 1

The Service Gateway Framework 1

Services 2

Bundles 2

Bundle Contexts 2

ServiceRegistrations and ServiceReferences 2

OSGi Service Gateway Architecture 3

Manifest 4

Bundle Activator 4

Import-Package and Export-Package 4

A Tour Behind the Scenes 5

The Component-Based Programming Model 6

Separation of Interface and Implementation 7

Creating Services for the JES 8

Steps to Develop a Bundle 8

Getting Started 8

Locate the Tutorial Files 8

▼ Start the JES Framework 9

Creating a Serviceless Bundle 9

BundleActivator Implementation for greeting1 Bundle 10

Manifest File for the greeting1 Bundle 10

▼ Build greeting1 Bundle 10

▼ Run greeting1 Bundle 11

Creating a Service Interface and Implementation 11

GreetingService Interface 12

Casual GreetingService Implementation 12

BundleActivator Implementation for greeting2 Bundle 13

Manifest File for greeting2 Bundle 14

▼ Build greeting2 Bundle 14

▼ Run greeting2 Bundle 14

An Alternative Service Implementation 15

BundleActivator Implementation for greeting3 Bundle 16

▼ Build greeting3 Bundle 16

▼ Run the greeting3 Bundle 17

Using Another Service 17

Manifest File for the club Bundle 19

▼ Build the club Bundle 19

▼ Run the club Bundle 19

Try This 19

Using a Core JES Service: HttpService 20

BundleActivator for the greeting4 Bundle 21

Manifest File for the greeting4 Bundle 22

GreetingServlet Example 22

Try This 23

Where to Go from Here 23

25

A. How to Modify JES Makefiles 25

Preface

This is a test document.

Before You Read This Book

Include this section only if this book requires that the reader must have read other documents first. This isn't the spot for recommended reading—that comes later.

If readers must know how to do something or must have completed something before using this book, tell them here. For example:

In order to fully use the information in this document, you must have thorough knowledge of the topics discussed in these books:

- *Editorial Style Guide*
- *Sun Microsystems New Look Book*
- *Frame of Reference*

How This Book Is Organized

If you want to describe the contents of your document, use a live cross-reference to list each chapter and give a brief description. Use the ChapterNumber and AppendixNumber cross-reference formats to create the cross-reference. Chapter and appendix titles are no longer included.

Chapter 1 describes entrance requirements for 14 trade schools. The chapter includes an application form.

Appendix A should be used only by experienced technical writers in panic situations.

Glossary is a list of words and phrases and their definitions.

Using UNIX Commands

Use this section to alert readers that not all UNIX commands are provided.
For example:

This document may not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

- *Solaris Handbook for Sun Peripherals* (*If you are incorporating Solaris software commands in your document, delete this sentence.*)
- AnswerBook2™ online documentation for the Solaris™ operating environment
- Other software documentation that you received with your system

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide.* These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename.*

Shell Prompts

Shell	Prompt
C shell	*machine_name*%
C shell superuser	*machine_name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Related Documentation

Application	Title	PartNumber
Installation	*ABC Release Notes*	801-xxxx
Service	*ABC System Service Manual*	801-xxxx
Options	*DEF SBus Card Manual*	800-xxxx

Accessing Sun Documentation Online

The `docs.sun.com`[sm] web site enables you to access Sun technical documentation on the Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject at:

```
http://docs.sun.com
```

Ordering Sun Documentation

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at:

```
http://www1.fatbrain.com/documentation/sun
```

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

```
docfeedback@sun.com
```

Please include the part number (8xx-xxxx-xx) of your document in the subject line of your email.

CHAPTER 1

Java Embedded Server Tutorial

This tutorial explains basic concepts of the Open Source Gateway Initiative (OSGi) service gateway architecture and the Java Embedded Server (JES), and teaches you how to write and deploy your own services.

The code examples presented here can all be compiled, installed, and activated on a running instance of the JES. You are encouraged to modify them and inspect how their running results change. You can also use them as templates as you develop your own services.

This tutorial does not cover every feature of the JES. It is an aid to get you started and ease your learning curve. See the recommended readings at the end of this tutorial to learn more on how to unleash the full power of the JES.

Concepts

In this section, we introduce the foundation concepts of the Java Embedded Server. We revisit them when we work with concrete examples, which may help you put the pieces together.

The Service Gateway Framework

The service gateway framework is the hosting platform from which services are deployed and run. It provides the following functionality to all services it hosts:

- Service registry
- Bundle life cycle management
- Tracking dependencies among services
- Event notifications

How these work will become evident presently when we describe the interplay of various entities within the framework.

Services

A service consists of Java classes that provide certain functionality. For example, an HTTP service implements the HTTP protocol and can respond to requests from HTTP clients; a vending machine service, on the other hand, examines the machine's internal temperature, sets prices for merchandise, and dispenses soda cans.

Programmatically, a service can consist of any Java objects, usually designed with interface and implementation separated. There are no other restrictions on what a service can do and how it is implemented.

Bundles

A bundle is a JAR file that contains the services, Java classes, and resources of any software component that runs on framework. It can contain one or more services, which usually make up an integrated functional unit.

Bundle Contexts

The framework creates a bundle context when a bundle is activated. A bundle context is the interface between the framework and a bundle. In other words, it represents the execution environment for the bundle.

ServiceRegistrations and ServiceReferences

When you successfully register a service with the framework using `BundleContext.registerService` a ServiceRegistration object is returned. You can use a ServiceRegistration object to change the properties of the service or to unregister the service.

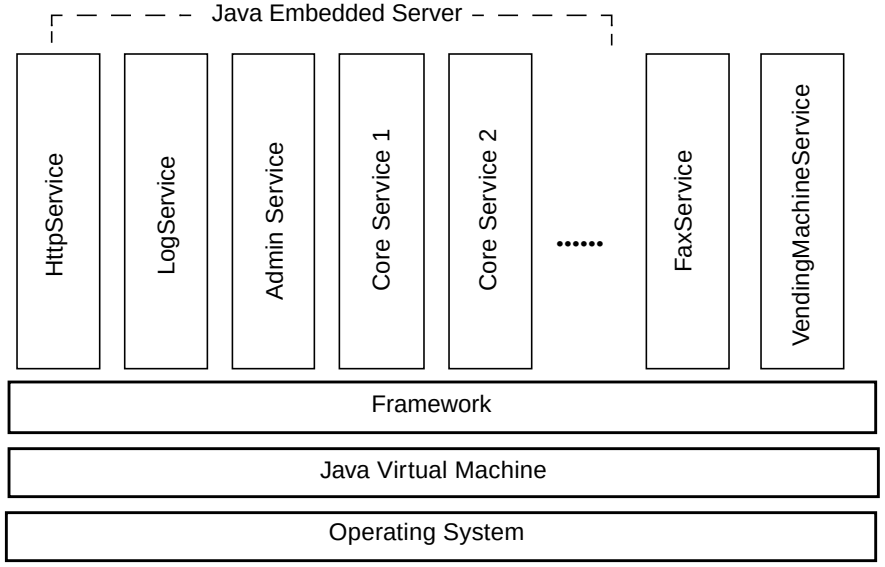
A ServiceReference is a reference to a service. It can be used to examine the properties of a service or to get the service object itself. Getting a service object is a two-step process.

You obtain a ServiceReference using the String name of the interface or class that is the type of the service, then use that ServiceReference to get service object. The indirection provided by ServiceReferences allows a caller to examine various properties of a service before committing to use it.

OSGi Service Gateway Architecture

FIGURE 1-1 shows the architecture of the OSGi (Open Service Gateway initiative) service gateway. The Java Embedded Server is designed to comply with the OSGi Specification Release 1.0. The JES software is written entirely in the Java Programming Language, so must therefore run on top of a Java Virtual Machine (VM). Above the Java VM layer comes a thin layer of the framework. The framework is a platform from which bundles can be deployed.

FIGURE 1-1 OSGi Service Gateway Architecture



Each bundle contains some service, and can be installed, activated, updated, deactivated, and uninstalled while it is hosted by the framework. This is the bundle life cycle.

When a bundle is first installed, a dedicated bundle classloader is created to access classes and resources provided by the bundle.

When a bundle is activated, it usually registers the service it provides with the framework under the names of the service types. Henceforth, the service begins to function, until the containing bundle is deactivated.

At the bundle activation time, the framework also tries to resolve dependencies among bundles. When bundle A needs to use classes or services provided by bundle B, A is said to depend on B. The framework ensures that A is present or B is not started.

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

Having discussed the concepts, let's see what means are available to us in Java Embedded Server in order to realize these ideas.

Manifest

A manifest file is a standard text file in a JAR archive; it contains information about the contents of the archive in headers. The OSGi has defined additional headers that can be included in the manifest file for a bundle. These headers provide the framework with “hooks" to the resources in the bundle, so that the framework knows what to do with the bundle as it goes through its life cycle. Some of the most essential headers are briefly explained below.

Bundle Activator

This header points to the bundle activator within the bundle. The bundle activator is a class implementing the `org.osgi.framework.BundleActivator` interface. It defines the customized start and stop logic when the framework activates and deactivates the bundle. We'll see how this is done in our first example.

Import-Package and Export-Package

Bundles that offer classes for others to use declare the packages with the Export-Package header; usually this involves the public service interfaces. Bundles that need to use classes from other bundles declare the needed packages with the Import-Package header. The framework is responsible for the matchmaking. We'll see an example shortly.

A Tour Behind the Scenes

It's high time that we examined the chain of events that occur when a bundle is installed, activated, stopped, and uninstalled from the framework. Note that some "unimportant" actions are omitted for sake of clarity; see the OSGi Framework Specification for complete descriptions.

1. **A bundle is installed.**

   The framework reads the contents of the bundle and establishes its presence by assigning it an ID and caching its location and state persistently.

2. **The bundle is activated.**

   The framework first checks if the Java classes required by this bundle have been exported by other bundles. If so, the `start()` method of the bundle's `BundleActivator` is called; any service provided by the bundle is registered with the framework. When this step is completed, the service starts running.

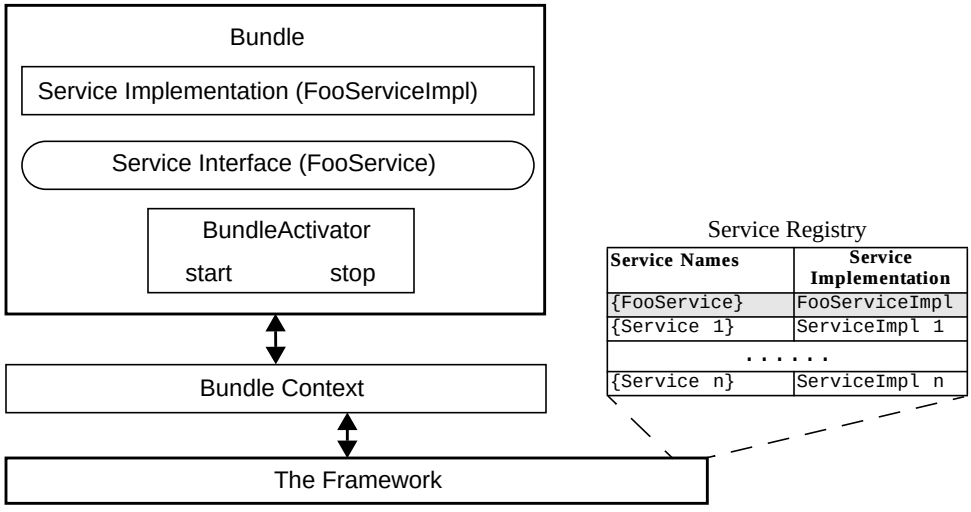
3. **The bundle is deactivated.**

   The bundle's `BundleActivator.stop()` method is called; usually the service provided by the bundle is unregistered from the framework. When this step is completed, the service stops running

4. **The bundle is uninstalled.**

   The bundle is removed from the computer.

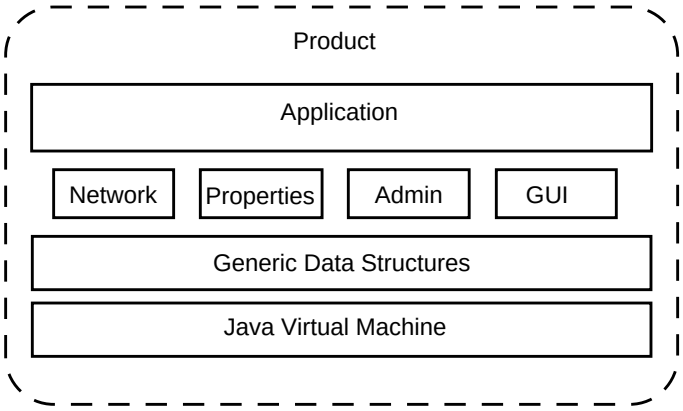
FIGURE 1-2 Anatomy of a bundle, its interaction with the framework after activation, and its service registration in the framework..



The Component-Based Programming Model

Java Embedded Server follows a component-based programming model, as contrast to the traditional library- or toolkit-based approach. Programming JES services is not merely Object-Oriented Programming in Java, because JES entities such as services and bundles have richer semantics than those of objects and classes in the Java Programming Language. The following figure depicts a library-based architecture.

FIGURE 1-3 Architecture of a hypothetical library-based software product.



Compare this picture with FIGURE 1-1, the JES architecture, we can list the following differences:

- In library-based model, multiple layers of software abstractions stack up one on top of another, while in component-based model, multiple software components plug in side-by-side.
- In library-based model, the sum of all libraries must be packaged together for the product to work, while in component-based model, a subset of components can be packaged together and serve useful functionality as a product.
- In library-based model, one must rebuild and repackage the entire set of libraries to fix bugs and add features during compile time, while in component-based model, new components can be added or existing components updated in an incremental way during runtime.
- In library-based model, problems in lower layers can propagate up and affect the stability of the entire software, while in component-based model, they are insulated from one another.

- In library-based model, changes made to public interface have less impact because one has to rebuild the software in its entirety anyway, while in component-based model, the cost of redoing public interface can be prohibitive because other components may have relied upon it at runtime.
- In library-based model, it is easy to follow the flow of control because applications usually have one entry point (e.g., `public static main(String[] args)`), while in component-based model, components are controlled by a hosting environment and they interact with one another to function.

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

Separation of Interface and Implementation

Separation of interface and implementation encourages software reuse, improves flexibility, and reduces maintenance cost. To the external caller of the service, its interface is the only thing that is exposed. The methods that are available, what they do, what arguments they take, what return value they are expected to produce, and what exceptions they may potentially throw are all clearly documented in the interface. The interface acts as a contract between the clients (callers) and the service. The implementation of this interface, on the other hand, is private to the service; the implementation is obliged to fulfill the specification of the interface, but it has much flexibility in how the specification is met. As a result, you can have multiple implementations for the same interface, which enables you to

- take advantage of different environments; for example, a scientific computation service can provide a compute method in its interface, but one implementation can optimize for multiprocessor system while another can utilize multiple computers over a network.
- have useful abstractions; for example, a filesystem service can offer its callers simple abstraction of reading, writing, and listing files, but it may have one implementation over local hard disks and another over networked filesystem such as NFS.
- minimize impact as a result of bugfix or feature enhancement; software always evolve; because of the separation of interface and implementation, the callers of the service are insulated from changes made to the implementation. This benefit can significantly contribute to the overall stability of the entire software system built with this model.

Separation of interface and implementation also forces the developer to think harder during the design stage: it is desirable to have the minimal set of methods in the service interface, 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 go back and change a service interface when many other services have already come to rely on that piece of interface. This is one of the major challenges developing for Java Embedded Server.

Creating Services for the JES

Enough theory. Let's work with some code examples to see things in action.

Steps to Develop a Bundle

In most cases, you follow these five steps to create a bundle that can be deployed on the Java Embedded Server:

1. Design and write an interface for your service.
2. Implement the interface for your service.
3. Implement the `org.osgi.framework.BundleActivator` interface to handle the start and stop logic of the bundle.
4. Write the manifest file for the bundle.
5. Package the manifest, bundle activator, and service files into a a single JAR file, ready for deployment.

Getting Started

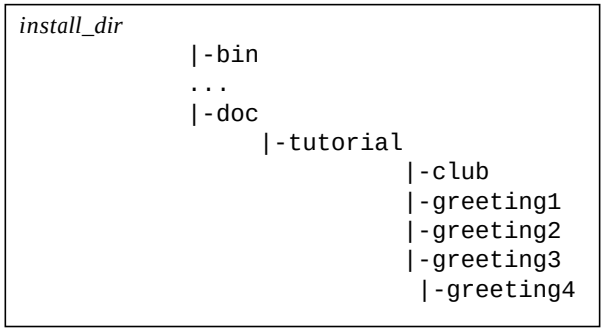
First, locate the tutorial example files, then start the JES fraemwork.

Locate the Tutorial Files

The complete set of examples are included with the distribution of the Java Embedded Server, and can be found at *install_dir*`/doc/tutorial` directory, where *install_dir* is the path to the where you have installed the JES. Each example goes into a subdirectory; they are `club`, `greeting1` , `greeting2` , `greeting3` , `greeting4`. Each of the following sections is based on one example, whose location

will be given. The examples are ready to be compiled and run. Appendix A shows you how to modify the makefiles if you want to move the examples out of the JES distribution; this is desirable if you want to develop your own bundles independent of the requirement of JES directory hierarchy.

FIGURE 1-4 Locating the Examples in the JES Directory Structure



▼ Start the JES Framework

While you work on the examples that follow, run the JES framework from one terminal and run builds in another terminal.

1. **Set the the CLASSPATH variable.**

   `setenv CLASSPATH` *install_dir*`/lib/framework.jar`

2. **Start the JES framework.**

   `java com.sun.jes.impl.framework.Main`

3. Include remaining start up steps--install & start core bundles.

Creating a Serviceless Bundle

In this first example, we create a bundle that does not supply any services, but simply prints “Hello” when started and “Bye” when stopped.

Location: *install_dir*`/doc/tutorial/greeting1`

Input files: Activator.java, greeting1.mf

Output files: greeting1.jar

BundleActivator Implementation for greeting1 Bundle

Since this bundle does not have any services, all the work is done inside its bundle activator. The `Activator` class implements the `org.osgi.framework.BundleActivator` interface, therefore it must define `start` and `stop` methods with the logic particular to this bundle.

CODE EXAMPLE 1-1 `BundleActivator` implementation for `greeting1`

```
package greeting1;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.BundleException;

public class Activator implements BundleActivator {
    public void start(BundleContext ctxt) throws BundleException {
        System.out.println("Hello");
    }

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

Manifest File for the greeting1 Bundle

The manifest file for greeting1, greeting1.mf, contains only one header. It tells the framework to look for `greeting1/Activator.class` in the bundle and use it as the activator.

CODE EXAMPLE 1-2 greeting1.mf

```
Bundle-Activator: greeting1.Activator
```

▼ Build greeting1 Bundle

Here are the steps to build an example. We will describe how to do this with greeting1 example; the rest are similar.

1. **Change to the example directory.**

   `cd` *install_dir*`/doc/tutorial/greeting1`

2. **Invoke build command.**

   Type one of the following build commands on the command line.

TABLE 1-1 Build Commands for UNIX and Windows

UNIX	Windows NT
`gnumake`	`nmake`

The resultant bundle JAR file will be generated at *install_dir*`/doc/tutorial/greeting1.jar`.

▼ Run greeting1 Bundle

3. **Install the bundle just generated.**

```
> install file:/doc/tutorial/greeting1.jar
```

4. Get the bundle ID for the bundle you have just installed.

```
> bundles
ID   STATE       LOCATION
--   ---------   ------------------------
...
3    INSTALLED   file:/.../doc/tutorial/greeting1.jar
```

5. **Start the bundle using the bundle ID.**

```
> start 3
Hello
```

6. **Stop the bundle.**

```
> stop 3
Bye
```

Creating a Service Interface and Implementation

In this section, we create a service that does something once it is started. Furthermore, we want to write this service in a manner that its interface and implementation are separated.

Location: *install_dir*`/doc/tutorial/greeting2`

Input files: Activator.java, greeting2.mf, GreetingService.java, CasualGreetingImpl.java

Output files: greeting2.jar

GreetingService Interface

The `GreetingService.greet` method requires three strings. Any implementation of the `GreetingService` interface must supply a definition of the method..

CODE EXAMPLE 1-3 GreetingService Interface

```
package greeting2.service;

public interface GreetingService {

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

Notice the detailed comments preceding the only method in the interface. The `javadoc` tool generates API documentation from code comments. Familiarize yourself with the tags and formats recognized by `javadoc` and use them in your code, as your fellow developers will need this information as they use your service.

Casual GreetingService Implementation

As we know, cultural and social circumstances usually dictate how people greet each other. This illustrates the need for separation of interface and implementation well. Here we are to show an implementation of `GreetingService` in casual settings.

CODE EXAMPLE 1-4 Casual Greeting Implementation

```
package greeting2.impl;
import greeting2.service.GreetingService;

class CasualGreetingImpl implements GreetingService {
    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 of the `greet` method matches the signature stipulated in the interface exactly. Additionally, the implementation class resides in a different Java package than that of the service interface; unlike the service interface, which is public, the implementation class is package private.

BundleActivator Implementation for greeting2 Bundle

Because our bundle now provides services, a few more methods, such as registering and unregistering services, are necessary in the `BundleActivator` implementation.

CODE EXAMPLE 1-5

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

public class Activator implements BundleActivator {
    private ServiceRegistration reg = null;
    public void start(BundleContext ctxt) throws BundleException {
        GreetingService greetingSvc = new CasualGreetingImpl();
        Properties props = new Properties();
        props.put("description", "casual");
        reg = ctxt.registerService
               ("greeting2.service.GreetingService",
                   greetingSvc, props);
    }

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

The framwork passes the `BundleContext ctxt`, to the `start` and `stop` methods. The bundle context is provided to the bundle so that it can interact with the framework through the `BundleContext` interface.

When the `start` method is called, it creates an instance of the service implementation class (`greetingSvc`) and a simple property associated with the service. It then registers the service under its type along with the property.

The `stop` method unregisters the service.

Manifest File for greeting2 Bundle

Because we intend to offer services for other bundles to use, we must export the Java package in which the service interface is defined.

CODE EXAMPLE 1-6 greeting2.mf

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

The advantage of putting implementation classes and service interfaces into different Java packages becomes obvious here: by only exporting the interface package, callers from other bundles do not have any access to the implementation details of the service.

▼ Build greeting2 Bundle

1. **Change to the example directory.**

   `cd` *install_dir*`/doc/tutorial/greeting2`

2. **Invoke build command for your system.**

   ```
   gnumake # UNIX
   nmake # Windows
   ```

   The make command generates the JAR file *install_dir*`/doc/tutorial/greeting2.jar`.

▼ Run greeting2 Bundle

1. **Install the bundle just generated**

   `> install file:/doc/tutorial/greeting2.jar`

2. **Get the bundle ID for the bundle you have just installed.**

   `> bundles`

3. **Start the bundle using the bundle ID.**

   `> start` *greeting2_bundle_id*

   When you start the greeting2 bundle, no messages appear. This is to be expected, as the services will not exhibit what they do until they are called upon by some other services.

4. **Stop the bundle.**

An Alternative Service Implementation

Location: *install_dir*`/doc/tutorial/greeting3`

Input files: Activator.java, greeting3.mf, FormalGreetingImpl.java, GreetingService.java

Output files: greeting3.jar

In a formal setting, we would want to implement our `GreetingService` interface as follows.

CODE EXAMPLE 1-7 Formal Greeting Implementation

```
package greeting3.impl;
import greeting2.service.GreetingService;
public class FormalGreetingImpl implements GreetingService {
    public void greet(String firstName,
                      String lastName,
                      String title) {
        System.out.println
            ("My name is Scott McNealy. How do you do, "
               + title + " " + lastName + "? "
               +  "It's a pleasure to make your acquaintance.");
  }
}
```

BundleActivator Implementation for greeting3 Bundle

The `BundleActivator` must register the correct implementation with the framework.

CODE EXAMPLE 1-8 Modified BundleActivator Implementation

```
package greeting3.impl;
import java.util.Properties;
import org.osgi.framework.*;
import greeting2.service.*;
public class Activator implements BundleActivator {
    private ServiceRegistration reg = null;
    public void start(BundleContext ctxt) throws BundleException {
        GreetingService greetingSvc = new FormalGreetingImpl();
        Properties props = new Properties();
        props.put("description", "formal");
        reg = ctxt.registerService
              ("greeting2.service.GreetingService",
              greetingSvc, props);
    }

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

Here in the `start` method, an instance of `FormalGreetingImpl` class is instantiated, and `FormalGreetingImpl` implements the same `GreetingService` interface from `greeting2` package.

▼ Build greeting3 Bundle

The greeting2 example must be compiled before greeting3 is compiled, because the implementation in greeting3 needs the interface class to be compiled in the previous example.

1. **Change to the example directory.**

   `cd` *install_dir*`/doc/tutorial/greeting3`

2. **Invoke build command for your system.**

   ```
   gnumake # for UNIX or
   nmake # for Windows
   ```

   The make command generates the JAR file *install_dir*`/doc/tutorial/greeting3.jar`.

▼ Run the greeting3 Bundle

1. **Install the bundle just generated**

   `> install file:/doc/tutorial/greeting3.jar`

2. **Get the bundle ID for the bundle you have just installed.**

   `> bundles`

3. **Start the bundle using the bundle ID.**

   `> start` *greeting3_bundle_id*

   When you start the greeting3 bundle, no messages appear. This is to be expected, as the services will not exhibit what they do until they are called upon by some other services. We demonstrate this in the next example.

Using Another Service

Location: *install_dir*`/doc/tutorial/club`

Input files: Activator.java, club.mf

Output files:

One bundle's service can be used by another more elaborate service to achieve more complex overall functionality. Suppose we have a club service that enrolls new members and during the process, we want to use an implementation of `GreetingService` to show greetings to newcomers. In this example, the

ClubService bundle depends upon that of the GreetingService. This example does not show how to define the ClubService bundle; it focuses on obtaining the GreetingService and using it in its activator.

CODE EXAMPLE 1-9 BundleActivator for the ClubService Bundle

```
package club;
import org.osgi.framework.*;
import greeting2.service.GreetingService;

public class ClubActivator implements BundleActivator {
    public void start(BundleContext ctxt) throws BundleException{
        ServiceReference[] ref = ctxt.getServiceReferences
               ("greeting2.service.GreetingService",
                  "(description=casual)");
        GreetingService greetingSvc
               = (GreetingService) ctxt.getService(ref[0]);
        greetSvc.greet("Bill", "Gates", "Chairman");
    }

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

When club bundle is started, it first gets the service reference to the service it is going to use; it does this by providing two parameters—the interface name of the service and an LDAP filter on the service properties—to the `getServiceReferences` method. We need the assistance of the LDAP filter to pinpoint the instance of the service we intend to use, as we know there may be two implementations of the `GreetingService` interface registered in the framework. Recall that the `description` property has been registered with the service.

Then the activator calls the `getService` method using the service reference to get an instance of the correct `GreetingService` object. Because the `greeting2.GreetingService` has registered with the framework, and the club bundle has declared its dependency on the package in its manifest (see below) , the framework is able to satisfy the request.

Manifest File for the club Bundle

The Import-Package header in the manifest file specifies out the Java package required by the club bundle. When the framework starts the bundle, the framework will try to ensure that the needed packages have been offered (exported) by the greeting2 bundle; if so, the club bundle will be activated; otherwise, it will not get started.

CODE EXAMPLE 1-10 club.mf

```
Bundle-Activator: club.ClubActivator
Import-Package: greeting2.service
```

▼ Build the club Bundle

1. **Change to the example directory.**

   `cd` *install_dir*`/doc/tutorial/club`

2. **Invoke build command for your system.**

   ```
   gnumake # UNIX
   nmake # Windows
   ```

   The make command generates the JAR file *install_dir*`/doc/tutorial/club.jar`.

▼ Run the club Bundle

Since the club bundle depends upon the service provided by greeting2 bundle, make sure the greeting2 bundle is installed and activated before you start the club bundle.

1. **Install the bundle just generated**

   `> install file:/doc/tutorial/club.jar`

2. **Get the bundle ID for the bundle you have just installed.**

   `> bundles`

3. **Start the bundle using the bundle ID.**

   `> start` *club_bundle_id*

******What happens?******

Try This

One way to examine the dependency relationships is from the JES console. At the prompt, type.

`> exportedpackages`

and you will see the following display:

```
Package: greeting2.service (0.0.0)
   Exported by: 1 (file:/.../greeting2.jar)
   Imported by: 2 (file:/.../club.jar)
```

This indicates that the Java package greeting2.service version 0.0.0 has been exported by bundle `greeting2.jar` and imported by bundle `club.jar`.

Using a Core JES Service: HttpService

Location: *install_dir*`/doc/jes/tutorial/greeting4`

We've seen how you can have one service use another service. The Java Embedded Server includes a set of core services that you may find useful. In this example, we'll see how to use one of these services, the HttpService.

The main functionality of JES HttpService is to map resources to URL namespace, so that when a client requests an URL, the corresponding resource is delivered using HTTP protocol. Resources can be some HTML files, classes from bundles, or servlets.

So far we have been sending output messages to stdout. In this example, the greeting message appears in a web browser. To do so, we are to write a servlet and ask the help of JES HttpService.

BundleActivator for the greeting4 Bundle

Much of the preparation is done in the activator.

CODE EXAMPLE 1-11 Bundle Activator for HttpService Example

```
package greeting4;
import com.sun.servicespace.wizard.ActivatorWizard;
import com.sun.servicespace.*;
import com.sun.jes.service.http.*;

public class BundleActivator implements BundleActivator {
    private HttpService  httpService = null;
    private GreetingServlet greetingServlet = null;
    private static String GREETING_ALIAS = "/jes/greetings.html";

    public void start(BundleContext ctxt) throws BundleException {
        ServiceReference ref = ctxt.getServiceReference
               ("org.osgi.service.http.HttpService");
        this.httpService = (HttpService) ctxt.getService(ref);
        this.greetingServlet = new GreetingServlet();
        try {
            servletReg =
httpService.registerServlet(GREETING_ALIAS,
            greetingServlet, null);
        }
        catch (Exception ex) {
            System.err.println(ex);
            throw new BundleException(ex.getMessage());
        }
    }

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

The framework passes the BundleContext to the `BundleActivator.start` method. It gets a handle to the JES HttpService by calling `getService` method of the BundleContext object, passing in HttpService' service reference; then it registers our servlet with HttpService under the specified URL alias (held in `GREETING_ALIAS`).

When the bundle is stopped, the servlet is unregistered from HttpService using the same alias it was registered under. What's missing is that we do not have a service, nor do we have service registration or unregistration in this example! This is completely permissible: a bundle can simply just have classes that can be invoked by BundleActivator , and does not contain any services.

See the HttpService section in the Developer Guide for complete coverage of the usage of HttpService. ****XREF TO DEV GUIDE******

Manifest File for the greeting4 Bundle

The manifest file defines the dependency relationship.

```
Bundle-Activator: greeting4.Activator
Import-Package: org.osgi.service.http
```

GreetingServlet Example

And here is the servlet, where all the magic happens.

CODE EXAMPLE 1-12 GreetingServlet Example

```
package greeting4;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
/**
* This servlet responds to HTTP GET requests for the URL under
which
* this servlet is registered with HttpService. It generates the
* greeting message in HTML and returns to the client.
*/
class GreetingServlet extends HttpServlet {
    public void doGet(HttpServletRequest req,
              HttpServletResponse resp)
              throws ServletException, IOException {
        resp.setContentType("text/html");
        ServletOutputStream out = resp.getOutputStream();
        out.println("<HTML><HEAD>");
        out.println("<TITLE>Greetings</TITLE></HEAD>");
        out.println("<BODY><H1>Hey, nice to meet you!</H1>");
        out.println("</BODY></HTML>");
        out.close();
    }
}
```

This servlet responds to `HTTP GET` requests, generates and returns the HTML page. You can go to a browser and type the URL `http://`*host*`:8080/jes/greetings.html`.

Note – The URL is programmed into our bundle; it does not map to a directory tree on the disk.

Try This

In our examples, when an error happens, we simply print its message (look for `System.err.println()` lines). Try to change the code so that errors are logged using JES core service LogService.

Where to Go from Here

The OSGi Service Gateway Specification specifies the complete features of the framework. It delves into details of the concepts described at the beginning of this tutorial, and discusses their APIs in full. This document is more targeted to a framework implementor, but as a bundle developer, you have much to gain in understanding what's under the hood in the hosting environment.

Java Embedded Server Developer Guide describes the functionality of the core services provided by JES.

APPENDIX **A**

How to Modify JES Makefiles

The following is the Makefile used to build the examples in this tutorial.

```
GNUmakefile for greeting1 example
SRCDIR = greeting1 FILES = \
GreetingService.java
include ../GNUcommon.make
-----------------------------------------------------------------
GNUcommon.make for all examples; on Windows NT platform, a
corresponding common.make is provided.
# The example GNUMakefile must define the following variables and then
# include this file.
# SRCDIR: the name of the example directory.
# JARNAME: the name of the jar file for the bundle generated.
# FILES: .java files to be compiled.
# EXTRA_CLASSFILES: Other .class files to include in the .jar file.
ifndef JARNAME
JARNAME = $(SRCDIR).jar
endif
# The path to where JES has been installed; if it is not an
# absolute path, it is relative to the directory where gnumake is
# invoked.
INSTALLDIR = ../../../..
# If JES_JAVA_HOME is defined, use javac and jar from it, else from
# the PATH.
ifdef JES_JAVA_HOME
JAVAC=$(JES_JAVA_HOME)/bin/javac
JAR=$(JES_JAVA_HOME)/bin/jar
```

```
else
JAVAC=javac
JAR=jar endif
LIBS = $(INSTALLDIR)/lib/manager.jar:$(INSTALLDIR)/lib/
jsdk.jar:$(INSTALLDIR)/lib/remote.jar:$(INSTALLDIR)/lib/
jesservices.jar
CLASSFILES := $(FILES:.java=.class)
xCLASSFILES := $(CLASSFILES:%=$(SRCDIR)/%)
all:: ../$(JARNAME)
../$(JARNAME): $(CLASSFILES) $(EXTRA_CLASSFILES) Manifest
cd .. ; \
$(JAR) cmf $(SRCDIR)/Manifest $(JARNAME) $(xCLASSFILES)
$(EXTRA_CLASSFILES)
$(CLASSFILES): $(FILES)
CLASSPATH=$(LIBS):..:. ; \
export CLASSPATH; \
$(JAVAC) $?
$(EXTRA_CLASSFILES)::
clean:
/bin/rm -f $(CLASSFILES) ../$(JARNAME)
```

By changing variables *SRCDIR* , *INSTALLDIR* , *JARNAME* , and *FILES* , you will have some flexibility in determining the outcome of the build. If you decide to move the tutorial out of JES distribution directory tree, for example, you must modify the *INSTALLDIR* (currently `../..`) to point to the path where JES is installed, as our examples needs JES classes in order to compile.

Also we assume commands such as `javac` and `jar` are correctly included in your *PATH* environment variable; if not, you may want to spell out the absolute path to these commands on your system. Additionally, you may find the Makefile inadequate if your package goes into multiple levels of directories. In any case, this Makefile is meant to get things going, and unlike other examples in this tutorial, does not try to be exemplary.