Oracle® GlassFish Server 3.1 Application Development Guide

Part No: 821–2418
November 2010

110127@25097

List of Remarks

Contents

Tables

Preface

This *Application Development Guide* describes how to create and run Java Platform, Enterprise Edition (Java EE platform) applications that follow the open Java standards model for Java EE components and APIs in the Oracle GlassFish Server environment. Topics include developer tools, security, and debugging. This book is intended for use by software developers who create, assemble, and deploy Java EE applications using Oracle servers and software.

This preface contains information about and conventions for the entire Oracle GlassFish Server (GlassFish Server) documentation set.

GlassFish Server 3.1 is developed through the GlassFish project open-source community at `https://glassfish.dev.java.net/`. The GlassFish project provides a structured process for developing the GlassFish Server platform that makes the new features of the Java EE platform available faster, while maintaining the most important feature of Java EE: compatibility. It enables Java developers to access the GlassFish Server source code and to contribute to the development of the GlassFish Server. The GlassFish project is designed to encourage communication between Oracle engineers and the community.

The following topics are addressed here:

- "GlassFish Server Documentation Set" on page 9
- "Related Documentation" on page 11
- "Typographic Conventions" on page 12
- "Symbol Conventions" on page 12
- "Default Paths and File Names" on page 13
- "Documentation, Support, and Training" on page 14
- "Searching Oracle Product Documentation" on page 14
- "Third-Party Web Site References" on page 14

GlassFish Server Documentation Set

The GlassFish Server documentation set describes deployment planning and system installation. The Uniform Resource Locator (URL) for GlassFish Server documentation is `http://download.oracle.com/docs/cd/E18930_01/index.htm`. For an introduction to GlassFish Server, refer to the books in the order in which they are listed in the following table.

TABLE P–1 Books in the GlassFish Server Documentation Set

Book Title	Description
Release Notes	Provides late-breaking information about the software and the documentation and includes a comprehensive, table-based summary of the supported hardware, operating system, Java Development Kit (JDK), and database drivers.
Quick Start Guide	Explains how to get started with the GlassFish Server product.
Installation Guide	Explains how to install the software and its components.
Upgrade Guide	Explains how to upgrade to the latest version of GlassFish Server. This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications.
Deployment Planning Guide	Explains how to build a production deployment of GlassFish Server that meets the requirements of your system and enterprise.
Administration Guide	Explains how to configure, monitor, and manage GlassFish Server subsystems and components from the command line by using the `asadmin`(1M) utility. Instructions for performing these tasks from the Administration Console are provided in the Administration Console online help.
Application Deployment Guide	Explains how to assemble and deploy applications to the GlassFish Server and provides information about deployment descriptors.
Application Development Guide	Explains how to create and implement Java Platform, Enterprise Edition (Java EE platform) applications that are intended to run on the GlassFish Server. These applications follow the open Java standards model for Java EE components and application programmer interfaces (APIs). This guide provides information about developer tools, security, and debugging.
Add-On Component Development Guide	Explains how to use published interfaces of GlassFish Server to develop add-on components for GlassFish Server. This document explains how to perform *only* those tasks that ensure that the add-on component is suitable for GlassFish Server.
Embedded Server Guide	Explains how to run applications in embedded GlassFish Server and to develop applications in which GlassFish Server is embedded.
High Availability Administration Guide	Explains how to configure GlassFish Server to provide higher availability and scalability through failover and load balancing.
Performance Tuning Guide	Explains how to optimize the performance of GlassFish Server.
Troubleshooting Guide	Describes common problems that you might encounter when using GlassFish Server and explains how to solve them.

TABLE P–1 Books in the GlassFish Server Documentation Set *(Continued)*

Book Title	Description
Error Message Reference	Describes error messages that you might encounter when using GlassFish Server.
Reference Manual	Provides reference information in man page format for GlassFish Server administration commands, utility commands, and related concepts.
Message Queue Release Notes	Describes new features, compatibility issues, and existing bugs for GlassFish Message Queue.
Message Queue Administration Guide	Explains how to set up and manage a Message Queue messaging system.
Message Queue Developer's Guide for JMX Clients	Describes the application programming interface in Message Queue for programmatically configuring and monitoring Message Queue resources in conformance with the Java Management Extensions (JMX).

Related Documentation

The following tutorials explain how to develop Java EE applications:

- *Your First Cup: An Introduction to the Java EE Platform*. For beginning Java EE programmers, this short tutorial explains the entire process for developing a simple enterprise application. The sample application is a web application that consists of a component that is based on the Enterprise JavaBeans specification, a JAX-RS web service, and a JavaServer Faces component for the web front end.
- *Java EE 6 Tutorial*. This comprehensive tutorial explains how to use Java EE 6 platform technologies and APIs to develop Java EE applications.

Javadoc tool reference documentation for packages that are provided with GlassFish Server is available as follows.

- The API specification for version 6 of Java EE is located at `http://download.oracle.com/javaee/6/api/`.
- The API specification for GlassFish Server 3.1, including Java EE 6 platform packages and nonplatform packages that are specific to the GlassFish Server product, is located at `http://glassfish.java.net/nonav/docs/v3/api/`.

Additionally, the Java EE Specifications (`http://www.oracle.com/technetwork/java/javaee/tech/index.html`) might be useful.

For information about creating enterprise applications in the NetBeans Integrated Development Environment (IDE), see the NetBeans Documentation, Training & Support page (`http://www.netbeans.org/kb/`).

For information about the Java DB database for use with the GlassFish Server, see the Java DB product page (`http://www.oracle.com/technetwork/java/javadb/overview/index.html`).

The Java EE Samples project is a collection of sample applications that demonstrate a broad range of Java EE technologies. The Java EE Samples are bundled with the Java EE Software Development Kit (SDK) and are also available from the Java EE Samples project page (`http://java.net/projects/glassfish-samples`).

Typographic Conventions

The following table describes the typographic changes that are used in this book.

TABLE P–2 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	A placeholder to be replaced with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file.

Symbol Conventions

The following table explains symbols that might be used in this book.

TABLE P–3 Symbol Conventions

Symbol	Description	Example	Meaning
`[ ]`	Contains optional arguments and command options.	`ls [-l]`	The `-l` option is not required.
`{ | }`	Contains a set of choices for a required command option.	`-d {y|n}`	The `-d` option requires that you use either the `y` argument or the `n` argument.

TABLE P–3 Symbol Conventions *(Continued)*

Symbol	Description	Example	Meaning
`${ }`	Indicates a variable reference.	`${com.sun.javaRoot}`	References the value of the `com.sun.javaRoot` variable.
-	Joins simultaneous multiple keystrokes.	Control-A	Press the Control key while you press the A key.
+	Joins consecutive multiple keystrokes.	Ctrl+A+N	Press the Control key, release it, and then press the subsequent keys.
→	Indicates menu item selection in a graphical user interface.	File → New → Templates	From the File menu, choose New. From the New submenu, choose Templates.

Default Paths and File Names

The following table describes the default paths and file names that are used in this book.

TABLE P–4 Default Paths and File Names

Placeholder	Description	Default Value
as-install	Represents the base installation directory for GlassFish Server. In configuration files, *as-install* is represented as follows: `${com.sun.aas.installRoot}`	Installations on the Oracle Solaris operating system, Linux operating system, and Mac operating system: *user's-home-directory*`/glassfish3/glassfish` Windows, all installations: *SystemDrive*`:\glassfish3\glassfish`
as-install-parent	Represents the parent of the base installation directory for GlassFish Server.	Installations on the Oracle Solaris operating system, Linux operating system, and Mac operating system: *user's-home-directory*`/glassfish3` Windows, all installations: *SystemDrive*`:\glassfish3`
domain-root-dir	Represents the directory in which a domain is created by default.	*as-install*`/domains/`
domain-dir	Represents the directory in which a domain's configuration is stored. In configuration files, *domain-dir* is represented as follows: `${com.sun.aas.instanceRoot}`	*domain-root-dir*`/`*domain-name*

Documentation, Support, and Training

The Oracle web site provides information about the following additional resources:

- Documentation (`http://docs.sun.com/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://education.oracle.com/`)

Searching Oracle Product Documentation

Besides searching Oracle product documentation from the `http://docs.sun.com` web site, you can use a search engine by typing the following syntax in the search field:

search-term **`site:docs.sun.com`**

For example, to search for "broker," type the following:

`broker site:docs.sun.com`

To include other Oracle web sites in your search (for example, the Java Developer site on the Oracle Technology Network at `http://www.oracle.com/technetwork/java/index.html`), use `oracle.com` in place of `docs.sun.com` in the search field.

Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Oracle is not responsible for the availability of third-party web sites mentioned in this document. Oracle does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Oracle will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

P A R T I

Development Tasks and Tools

P A R T I I

Developing Applications and Application Components

CHAPTER 13

Developing OSGi-enabled Java EE Applications

This chapter describes how to configure OSGi services and OSGi client bundles for deployment on GlassFish Server, where they can then interoperate with Java EE applications. This chapter includes the following sections:

The OSGi module management subsystem that is provided with GlassFish Server is the Apache Felix OSGi framework. To enable you to administer this framework, the Apache Felix Remote Shell is enabled by default in GlassFish Server. For instructions on accessing and using the Apache Felix Remote Shell, see *Oracle GlassFish Server 3.1 Administration Guide*.

Overview of OSGi and Java EE Applications

GlassFish Serverenables interaction between OSGi components and Java EE components. Interaction automatically means bi-directional communication. For example, you can export EJBs as OSGi services without having to write any OSGi code. That allows any pure OSGi component, which is running without EE context, to discover the EJB and call it. That in turn allows you to write business components, such as EJBs, so that they can take advantage of things like declarative security, transaction, context dependency and injection, and yet allow them to be accessible to non-EE components.

You can perform distributed transactions as well since you can also configure various EE infrastructure services, such as TransactionManager, Data Sources, as OSGi services. Then you can start a transaction in your pure OSGi bundle, invoke an EJB as an OSGi service, and the transaction context will propagate. The same holds true for security or persistence context propagation as well.

In fact, you can also mix and match in the same application. If you do not want to use the EE component model, GlassFish Server is extensible enough to be augmented with blueprint containers or something else to support their model. By default, GlassFish Server ships with a declarative services bundle. GlassFish Server support for such hybrid applications is not just limited to EJB applications, there is support for hybrid web applications as well.

Create a Simple OSGi Service and Client

This section explains how to create a simple OSGi service that is invoked by an OSGi client. The same OSGi service is then invoked by a Web Application Bundle (WAB) client, which is actually a web application plus an OSGi bundle (also known as a hybrid application). The OSGi service is then replaced by an EJB-based service using Java Persistence API. This demonstrates how OSGi client and service can interoperate with Java EE counterparts.

[Remark 13–1 Writer: Still need to determine the main steps for GlassFish User's-only workflow]

Deploying an OSGi-enabled Web Application (WAR) as an OSGi Bundle

GlassFish Server allows web applications (WAR files) to be deployed as OSGi bundles, and thereby taking advantage of OSGi platform, as well as the Java EE platform. There are basically two starting points:

- You have a war file that has OSGi metadata in it.
- You have a vanilla war file.

In the latter case, you have to instruct GlassFish Server that it has to add necessary OSGi metadata to the WAR file. You can also customize the transformation step. It is achieved by using a special URl protocol called a *webbundle* together with the use of URL query parameters. GlassFish Server has a custom URL handler for this protocol and it does an in-place manifest rewrite when it encounters this scheme.

To use it, you would run a command similar to this in GlassFish Server:

```
telnet localhost 6666
install webbundle:file:///tmp/mybundle.war
start #bundle_id
```

Note – Telnet support in is provided by use of the Apache Felix Remote Shell, which is enabled by default in GlassFish Server. This shell uses the Felix shell service to interact with the OSGi module management subsystem, and enables you to perform administrative tasks, as described in the *Oracle GlassFish Server 3.1 Administration Guide*.

These commands make your web application available in the `localhost:8080/mybundle/` directory. A this point, you can control the life cycle of the web application using the OSGi bundle. For example, if you stop the bundle by issuing the `stop #bundle_id` command, the web application is undeployed. To deploy it again, issue the `start #bundle_id` command.

For vanilla web applications, you need to add a specific metadata called *Web-ContextPath* in the *manifest.mf* to mark the OSGi bundle as a Web Application Bundle (WAB).

Once you have done that, you can either install and start by running the shell commands without using the `webbundle`protocol or simply copy the bundle to the `glassfish/domains/domain1/autodeploy-bundles/` directory.

Remark 13–2 Writer FROM WIKI - "How this directory works is already described in a previous blog — http://weblogs.java.net/blog/ss141213/archive/2009/05/using_filesyste.html" -- NEED NEW SECTION IN ADMIN GUIDE CHAP FOR THIS

Using Typesafe Injection of Dynamic OSGi Services in Hybrid Java EE Applications

With GlassFish Server, application components can express their dependency on an OSGi Service, and have the container handle the discovery and binding of OSGi Services and inject them, by providing an additional qualifier, *@OSGiService*, in the injection point. So instead of all the verbose service discovery and binding code, you can state the requirement for an OSGi Service as follows:

@Inject @OSGiService

StockQuoteService sqs;

Note that the specification of the OSGi service type in the injection point is type-safe. The developer specifies that the injected service must implement the StockQuoteService interface using the field's type. Type-safety usually implies lesser runtime errors and easier debugging and, refactoring.

Since the injection is specified through standard *@Inject* coupled with a custom *OSGiService* qualifier, all standard CDI injection capabilities are available (constructor, field, setter method injection, programmatic lookup, etc). The container automatically manages service references and releases them when the component scope is completed.

A standard CDI portable extension (`org.glassfish.osgi-cdi`) comes pre-installed with GlassFish Server, that intercepts deployment of hybrid applications that has components who have expressed dependencies on OSGi services, as shown above. The portable extension takes care of discovering the service from the service registry using the criteria specified in the injection point, to bind and track the service and inject the service. Additional service discovery and injection related metadata could also be specified through annotation elements in the *OSGiService* qualifier.

For example, these are the current metadata attributes that could be specified:

- *Service Discovery Criteria:* The standard Filter syntax specified in the OSGi Core Specification can be used to narrow down choices for the Service type in the Service registry.
- *Wait Timeouts:* Waits for the specified amount of time for at least one service that matches the criteria specified to be available in the OSGi Service registry.
- *Dynamic Binding*: Used to handle service-dynamism. Since OSGi services are dynamic, they may not match the life cycle of the application component that has injected a reference to the service. Through this attribute, you could indicate that a service reference can be obtained dynamically or not. For stateless or idempotent services, a dynamic reference to a service implementation would be useful. The container then injects a proxy to the service and dynamically switches to an available implementation when the current service reference is invalid.

EXAMPLE 13–1 Example of an OSGi-enabled Stock Quote Service Interface

```
The following example demonstates how easy it to consume OSGi services in a hybrid Java EE application in a dynamic, t
```

For example, you could use your preferred IDE to create an OSGi bundle that registers a StockQuoteService implementation when the bundle is started. You could then create another web application bundle (WAB) that uses the StockQuoteService by having the container inject the service implementation using the *@OSGiService* qualifier. Then the servlet could then find all the symbols for which stock quotes are available and print their current quotes.

1. Your StockQuoteService API service interface could be something like this:

   ```
   org/acme/stockquoteservice/api/StockQuoteService.java
   public interface StockQuoteService {
      public Double getQuote(String symbol);
   }
   ```
2. And the service implementation could be at `org/acme/stockquoteservice/impl/SimpleStockQuoteServiceImpl.java` and could have a fixed list of symbols and quotes. The service implementation is registered in the `start()` method in the `BundleActivator`

   ```
   public class SimpleServiceActivator implements BundleActivator {
      public void start(BundleContext context) throws Exception {
         context.registerService(StockQuoteService.class.getName(),
         new SimpleStockQuoteServiceImpl(), null);
        }
   }
   ```
3. To deploy the bundle, use the Apache Felix Gogo shell . For example, if the Stock Quote service resides in the `/tmp` directory, install the bundle as follows:

EXAMPLE 13–1 Example of an OSGi-enabled Stock Quote Service Interface *(Continued)*

```
telnet localhost 6666
install webbundle:file:///tmp/stockquote_service/target/stockquote_service.jar
```

The shell will provide a Bundle ID for the installed bundle, as follows:

```
Bundle ID: 275
```

4. Start the bundle as follows:

```
start #bundle275
```

Tip – (Remember to replace ”275” in this command with the bundle ID provided by your Gogo shell.)

The Stock Quote service implementation is initialized during the bundle startup and registered in the OSGi Service Registry. An entry similar to the following must appear in the Apache Felix Gogo shell.

```
Registered:[IBM, MSFT, HPQ, ORCL]
```

5. The `stockquoteweb` application bundle references and uses the Stock Quote service, as follows:

```
  public class StockQuoteServlet extends HttpServlet {
     @Inject
     @OSGiService(/* wait for 1 min */ waitTimeout=60*1000)
     StockQuoteService sqs;
     ...
  }
```

Note that this WAR is a normal web application bundle, with an empty `beans.xml` descriptor to indicate that it is a CDI bean archive. The context root is specified as `stock_quote`, using the Web-ContextPath manifest header:

```
  Web-ContextPath                         /stockquote
```

For simplicity, the service API and one implementation of that service can be bundled in the `stockquote_service` bundle.

6. Install the stock quote WAB bundle using the Gogo shell. For example, if the WAR file is in the `/tmp` directory, use the shell to install WAB bundle, and then start the WAB bundle using the identifier provided by the shell, as follows:

```
  install file:///tmp/stockquote_cdi_wab/target/stockquote_cdi_wab.war
  Bundle ID: 276
  start 276
```

7. To see the stock quotes provided by the stock quote service, go to the following URL: `http://localhost:8080/stockquote/list`The web application uses the Stock Quote service implementation to get the quotes for a set of stock symbols.

8. To see how service dynamism is handled, stop the service bundle by executing `stop 275` in the Gogo shell. This stops the service bundle and the registered service implementation is removed from the service registry and is now unavailable for use. Refresh the `http://localhost:8080/stockquote/list` URL. Since the there is a wait

EXAMPLE 13–1 Example of an OSGi-enabled Stock Quote Service Interface *(Continued)*

timeout of 30 seconds, the OSGi CDI extension waits for 30 seconds before it quits and the web application prints `service unavailable`. However within the 30 seconds, if you execute `start 275` to start the service bundle, the service bundle would register the service implementation again and the container would get the latest service implementation and provide it to the servlet.

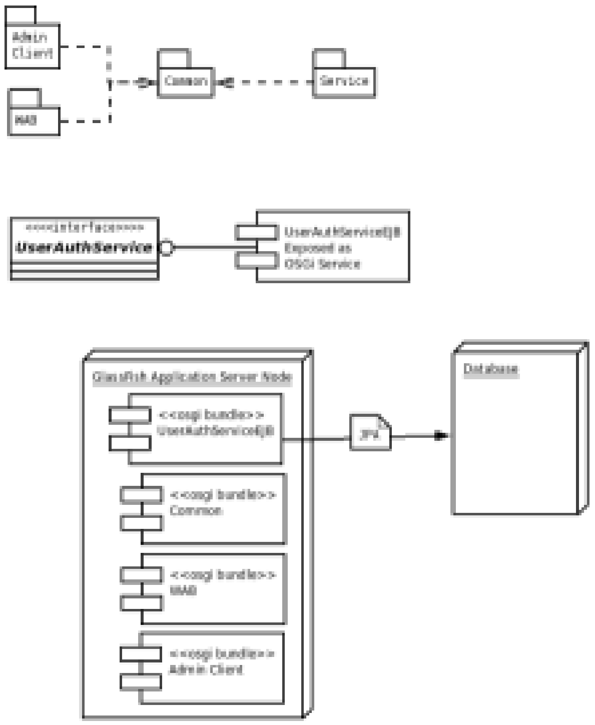
```
stop 275
SimpleServiceActivator stopped
start 275
SimpleServiceActivator::start
SimpleStockQuoteServiceImpl::Initializing quotes
SimpleStockQuoteServiceImpl::getSymbols
Registered:[IBM, MSFT, HPQ, ORCL]
SimpleServiceActivator::registration of Stock quote service successful
```

Using an EJB as OSGi Service

Remark 13–3 Writer **There's not enough content in the related wiki page for me add as suitable documentation: http://weblogs.java.net/blog/ss141213/archive/2010/03/30/ejb-osgi-service-demo-eclipsecon**

The example is organized as shown in this diagram:

FIGURE 13–1 Example of an EJB as an OSGi Service


Admin Client
Common
Service
WAB
<<interface>>
UserAuthService
UserAuthServiceEJB Exposed as OSGi Service
GlassFish Application Server Node
<<osgi bundle>> UserAuthServiceEJB
<<osgi bundle>> Common
<<osgi bundle>> WAB
<<osgi bundle>> Admin Client
JPA
Database

All the components including the EJB are deployed as separate OSGi bundle.

EJB uses JPA in container-managed mode to communicate with the database.

Both the admin client and Web Application Bundle client use the EJB using OSGi service registry.

Deployment order of bundles is irrelevant.

Local EJBs being accessed from other bundles.

Using JMS Message Consumer and Producer Services In an OSGi Bundle

This section contains an example of a JMS hybrid (OSGi + Java EE) application. In fact, it is a complete JMS consumer and producer bundle using OSGi and GlassFish Server.

JMS Message Consumer Bundle

The JMS message consumer bundle is also an OSGi bundle that contains a single class, which is the message consumer or listener. It is implemented as a Message Driven Bean (MDB). There is nothing OSGi-specific in the bean. The JAR file contains OSGi metadata and an additional GlassFish Server specific header called Export-EJB to indicate to the server that the OSGi bundle contains EJBs that need to be processed.

The relevant metadata for the EJB OSGi bundle looks like this:

```
[MANIFEST osgijms1.consumer.jar]

Export-EJB                              NONE
Bundle-ManifestVersion                  2
Bundle-SymbolicName                     my.osgijms1.consumer
Bundle-Version                          1.0.0.SNAPSHOT
Import-Package                          javax.ejb,javax.jms
Manifest-Version                        1.0
```

The MDB looks like this:

```
@MessageDriven(mappedName = "jms/osgi.Topic1")

public class AnMDB implements MessageListener {
    public void onMessage(Message message) {
        String str = null;
        if (message instanceof TextMessage) {
            try {
                str = TextMessage.class.cast(message).getText();
            } catch (JMSException e) {
                // ignore
            }
        }
        if (str == null) str = message.toString();
        System.out.println("AnMDB Received: " + str);
    }
}
```

JMS Message Producer Bundle

The JMS message producer bundle contains a single `BundleActivator` class. The bundle activator is configured about JMS destination using the OSGi Configuration Administration service. Upon configuration, it sends messages to the JMS destination. The complete source code for the message producer is shown below:

```
public class Activator1 implements BundleActivator {

  public void start(BundleContext context) throws Exception {
    System.out.println("Message producer started -
      waiting to be configured with topic name");
    Properties props = new Properties();
    props.put(Constants.SERVICE_PID, "osgijms1.producer");
    context.registerService(ManagedService.class.getName(), new ManagedService() {
        public void updated(Dictionary properties) throws ConfigurationException {
           if (properties != null) {
              String destinationName = (String) properties.get("osgijms1.Destination");
```

```
                String connectionFactoryName = (String) properties.get
                  ("osgijms1.ConnectionFactory");
                int noOfMsgs = Integer.valueOf((String) properties.get
                  ("osgijms1.NoOfMsgs"));
                sendMessage(connectionFactoryName, destinationName, noOfMsgs);
              }
          }
      }, props);
   }

    private void sendMessage(String connectionFactoryName, String destinationName,
    int noOfMsgs) {

        Connection connection = null;
        try {
            InitialContext ctx = new InitialContext();

            ConnectionFactory connectionFactory = (ConnectionFactory)
            ctx.lookup(connectionFactoryName);

            connection = connectionFactory.createConnection();

            Session session = connection.createSession(
                    false,
                    Session.AUTO_ACKNOWLEDGE);

            Destination dest = (Destination) ctx.lookup(destinationName);
            MessageProducer producer = session.createProducer(dest);
            TextMessage message = session.createTextMessage();

            for (int i = 0; i < noOfMsgs; i++) {
                message.setText("This is message " + (i + 1));
                System.out.println("Sending message: " + message.getText());
                producer.send(message);
            }

            /*
             * Send a non-text control message indicating end of
             * messages.
             */
            producer.send(session.createMessage());
        } catch (JMSException e) {
            System.err.println("Exception occurred: " + e.toString());
        } catch (NamingException e) {
            System.err.println("Exception occurred: " + e.toString());
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (JMSException e) {
                }
            }
        }
    }

    public void stop(BundleContext context) throws Exception {
    }
  }
```

▼ Create and Deploy the JMS Topic and Connection Factory Resources

1. Create the JMS topic and connection factory resources by executing the following `asadmin` commands:

   ```
   asadmin create-jms-resource --restype javax.jms.Topic jms/osgi.Topic1
   ```

   ```
   asadmin create-jms-resource --restype javax.jms.ConnectionFactory
   jms/osgi.ConnectionFactory1
   ```

2. Copy the message consumer and producer bundles and the configuration file to the `domain1/autodeploy/bundles/` directory. You can copy them in any order you want, but it is recommended that you copy them in the following order and monitor the `domain1/logs/server.log`. (For example, you can use `tail -f`) to see the action.)

   ```
   cp ./message-consumer/target/osgijms1.consumer.jar
   $glassfish/domain1/autodeploy/bundles
   ```

   ```
   cp ./message-producer/target/osgijms1.producer.jar
   $glassfish/domain1/autodeploy/bundles/
   ```

   ```
   cp ./osgijms1.producer.cfg $glassfish/domain1/autodeploy/bundles/
   ```

3. We will make the JMS resources available as OSGi services just like we make JDBC resources available as OSGi services. Once we do that, our message producer can track the service and send message once the resource is deployed.

Remark 13–4 Writer I can't find and such information on the OSGi wiki -- http://wikis.sun.com/display/GlassFish/BlogsGfOsgi

Using JDBC Resources as OSGi Services

This section contains an example that demonstrates how to make JDBC resources available as OSGi services.

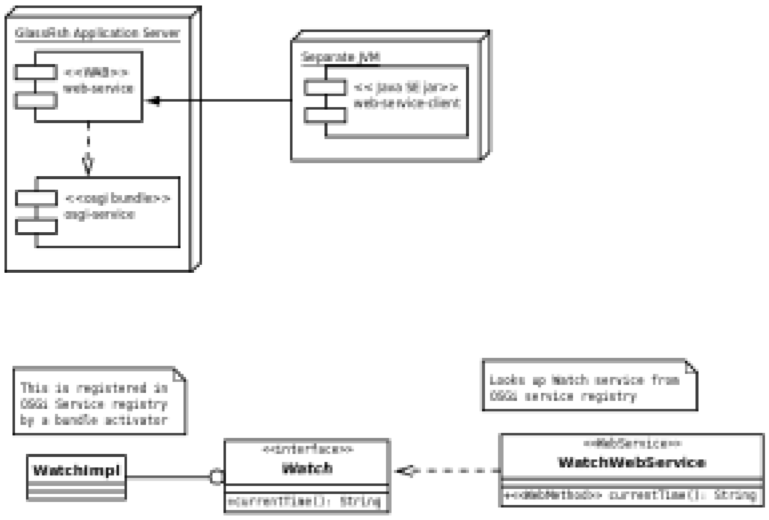
Remark 13–5 Writer There are no persistence topics on the OSGi wiki --

Using JAX-WS Web Services In an OSGi Bundle

This section contains an example that demonstrates a JAX-WS web service invoking an OSGi service using an OSGi service registry. The flow of the application modules is demonstrated in the following diagram:

Remark 13–6 Writer We need a better overivew of this section.

FIGURE 13–2 Example of a Web Service in an OSGi Bundle


GlassFish Application Server
<<WAB>> web-service
<<osgi bundle>> osgi-service
Separate JVM
<< Java SE jar>> web-service-client
This is registered in OSGi Service registry by a bundle activator
Looks up Watch service from OSGi service registry
WatchImpl
<<interface>> Watch
+currentTime(): String
<<WebService>> WatchWebService
+<<WebMethod>> currentTime(): String

- `web-service-client.jar` -- A plain JAR file that makes use of JAX-WS stack of Java SE environment to invoke the web service. It has a single class named `sahoo.hybridapp.jaxws1.webserviceclient.Main`. The rest of the classes that are part of the JAR are generated by the WSDL compiler as part of the build.
- `web-service.war` -- A Web Application Bundle (WAB). A WAB is a hybrid application -- it is both a Java EE archive as well as an OSGi bundle. In this case, it is a WAR file as well as an OSGi bundle. It is a WAR file, because it contains a servlet-based JAX-WS endpoint. It is an OSGi bundle, because we want to make use of OSGi service in the implementation of our web service. It contains a single class named `sahoo.hybridapp.jaxws1.webservice.WatchWebService`, which is defined as follows:

```
package sahoo.hybridapp.jaxws1.webservice;

import sahoo.hybridapp.jaxws1.service.Watch;
import org.osgi.framework.*;
import javax.jws.*;

@WebService
```

```
public class WatchWebService {
   @WebMethod public String currentTime() {
       Watch watch = getService(Watch.class);
       System.out.println("WatchService: OSGi service is: " + watch);
       if (watch == null) {
          return "I don’t have a watch";
       } else {
          return watch.currentTime();
       }
   }

   /**
     * This method looks up service of given type in OSGi service registry and returns if found.
     * Returns null if no such service is available,
     */
   private static <T> T getService(Class<T> type) {
       BundleContext ctx = BundleReference.class.cast(WatchWebService.class.getClassLoader())
          .getBundle().getBundleContext();
       ServiceReference ref = ctx.getServiceReference(type.getName());
       return ref != null ? type.cast(ctx.getService(ref)) : null;
   }
}
```

The MANIFEST.MF of `web-service.war` looks like this:

```
Bundle-ClassPath                   WEB-INF/classes/
Bundle-ManifestVersion             2
Bundle-SymbolicName                sahoo.hybridapp.jaxws1.web-service
Bundle-Version                     1.0.0.SNAPSHOT
Import-Package                     javax.jws;version="2.0",org.osgi.framework;version="1.5",sahoo.hybridapp.jaxws1.serv
Web-ContextPath                    /hybridapp.jaxws1.web-service
```

- `osgi-service.jar` -- An OSGi bundle that provides a service to other bundles. It contains two POJOs:
  - An interface named `sahoo.hybridapp.jaxws1.service.Watch`.
  - An implementation of the same interface called`sahoo.hybridapp.jaxws1.service.Activator`, which is responsible for registering an instance of `WatchImpl` in the OSGi service registry.

▼ Create and Deploy the OSGi Service and Web Service Bundles

1. **Use your preferred IDE to build the necessary OSGi service and WAB bundles. For the purposes of this example name them `osgi-service/target/osgi-service.jar` and `web-service/target/web-service.war`.**

2. **Deploy these OSGi service bundles to GlassFish by simply copying them to `domain1/autodeploy/bundles/` directory, as follows:**

   ```
   cp osgi-service/target/osgi-service.jar web-service/target/web-service.war
   $glassfish.home/domains/domain1/autodeploy/bundles/
   ```

GlassFish will automatically detect that `web-service.war` is a WAB and will perform necessary deployment of Java EE artifacts as a result of which a web service endpoint will be available. You would see something like this appearing in server.log:

```
WS00018: Webservice Endpoint deployed  WatchWebService  listening at address at
http://localhost:8080/hybridapp.jaxws1.web-service/WatchWebServiceService
```

3 **Once the web service is available, build the web-service-client.jar using a command such as this:**

`mvn -f web-service-client/pom.xml`

This is necessary because the WSDL URL, as specified in `web-service-client/pom.xml`, is not available until the web service is deployed.

4 **To test the web service, run:**

`java -jar web-service-client.jar`

The web service will print the current time as obtained from the web service, which in turn obtains it from the OSGi service.

P A R T I I I

Using Services and APIs