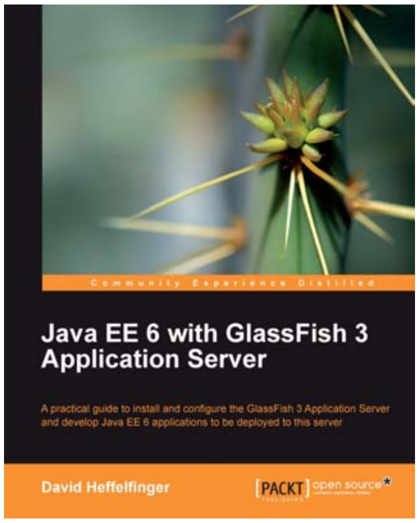
Java EE 6 with GlassFish 3 Application Server

David Heffelfi nger



Chapter No.7

"Java Messaging Service"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.7 "Java Messaging Service"

A synopsis of the book’s content

Information on where to buy this book

About the Author

David Heffelfinger is the Chief Technology Officer of Ensode Technology, LLC—a software consulting firm based in the greater Washington DC area. He has been architecting, designing, and developing software professionally since 1995, and has been using Java as his primary programming language since 1996. He has worked on many large scale projects for several clients, including the US Department of Homeland Security, Freddie Mac, Fannie Mae, and the US Department of Defense. He has a Masters degree in Software Engineering from Southern Methodist University. David is the Editor-in-Chief of Ensode.net (`http://www.ensode.net`), a website about Java, Linux, and other technology topics.

I would like to thank everyone who helped in making this book a reality. I would like to thank the Development Editors, Mehul Shetty and Dhiraj Chandiramani, and the Project Coordinators, Shubhanjan Chatterjee and Pallabi Chatterjee.

I would also like to thank the Technical Reviewers, Allan Bond and Arun Gupta for their insightful comments and suggestions.

Additionally, I would like to thank the GlassFish team at Oracle (formerly Sun Microsystems) for developing such an outstanding application server.

Finally, I would like to thank my wife and daughter for putting up with the long hours of work that kept me away from the family.

Java EE 6 with GlassFish 3 Application Server

This book begins with the installation of Glassfish 3 and deploying Java applications. It also explains how to develop, configure, package, and deploy servlets. Additionally, we will learn the processing of HTML forms. As we move on, we will develop Java Server Pages and get to know about implicit JSP objects. We will also get to know about all the JSTL (JSP Standard Tag Library) tag libraries. This book gives us a better understanding on how to manage data from a database through the Java Database Connectivity (JDBC) API and the Java Persistence API (JPA). We will also learn more about the newly introduced features of JPA 2.0 and develop JSF 2.0 applications to learn how to customize them. We will then set up Glassfish for the Java Messaging (JMS) API and understand the working of message queues and message topics. Later, we will use the Context and Dependency Injection (CDI) API to integrate application layers and study the SOAP-based web service development using the JAX-WS specification. Finally, we will learn more about the RESTful web service development using the JAX-RS specification.

The book covers the various Java EE 6 conventions and annotations that can simplify enterprise Java application development. The latest versions of the Servlet, JSF, JPA, EJB, and JAX-WS specifications are covered, as well as new additions to the specification, such as JAX-RS and CDI.

What This Book Covers

Chapter 1, Getting Started with GlassFish will discuss how to download and install GlassFish. We will look at several methods of deploying a Java EE application through the GlassFish web console, through the `asadmin` command, and by copying the fi le to the `autodeploy` directory. We will cover basic GlassFish administration tasks such as setting up domains and setting up database connectivity by adding connection pools and data sources.

Chapter 2, Servlet Development and Deployment will cover how to develop, configure, package, and deploy servlets. We will also cover how to process HTML form information by accessing the HTTP request object. Additionally, forwarding HTTP requests from one servlet to another will be explained, as well as redirecting the HTTP response to a different server. We will discuss how to persist objects in memory across requests by attaching them to the servlet context and the HTTP session. Finally, we will look at all the major new features of Servlet 3.0, including configuring web applications via annotations, pluggability through `web-fragment. xml` , programmatic servlet configuration, and asynchronous processing.

Chapter 3, JavaServer Pages will talk about how to develop and deploy simple JSPs. We will cover how to access implicit objects such as `request`, `session`, and so on, from JSPs. Additionally, we will look at how to set and get the values of JavaBean properties via the `<jsp:useBean>` tag. In addition to that, we will fi nd out how to include a JSP into another JSP at runtime via the `<jsp:include>` tag, and at compilation time via the JSP `include` directive. We will discuss how to write custom JSP tags by extending `javax.servlet.jsp.tagext.SimpleTagSupport` or by writing TAG fi les. We will also discuss how to access JavaBeans and their properties via the Unified Expression Language. Finally, we will cover the JSP XML syntax that allows us to develop XML-compliant JavaServer Pages.

Chapter 4, JSP Standard Tag Library will cover all JSP Standard Tag Library tags, including the core, formatting, SQL, and XML tags. Additionally, JSTL functions will be explained. Examples illustrating the most common JSTL tags and functions will be provided; additional JSTL tags and functions will be mentioned and described.

Chapter 5, Database Connectivity will talk about how to access data in a database via both the Java Database Connectivity (JDBC) and through the Java Persistence API (JPA). Defining both unidirectional and bidirectional one-to-one, one-to-many, and many-to-many relationships between JPA entities will be covered. Additionally, we will discuss how to use JPA composite primary keys by developing custom primary key classes. We will also discuss how to retrieve entities from a database by using the Java Persistence Query Language (JPQL). We will look at how to build queries programmatically through the JPA 2.0 Criteria API and automating data validation through JPA 2.0's Bean Validation support

Chapter 6, JavaServer Faces will cover how to develop web-based applications using JavaServer Faces—the standard component framework for the Java EE 5 platform. We will talk about how to write a simple application by creating JSPs containing JSF tags and managed beans. We will discuss how to validate user input by using JSF's standard validators and by creating our own custom validators, or by writing validator methods. Additionally, we will look at how to customize standard JSF error messages; both the message text and the message style (font, color, and so on). Finally, we will discuss how to write applications by integrating JSF and the Java Persistence API (JPA).

Chapter 7, Java Messaging Service will talk about how to set up JMS connection factories, JMS message queues, and JMS message topics in GlassFish using the GlassFish web console. We will cover how to send and receive messages to and from a message queue. We will discuss how to send and receive messages to and from a JMS message topic. We will find out how to browse messages in a message queue without removing the messages from the queue. Finally, we will look at how to set up and interact with durable subscriptions to JMS topics.

Chapter 8, Security will talk about how to use GlassFish's default realms to authenticate our web applications. We will cover the fi le realm, which stores user information in a fl at fi le, and the certificate realm, which requires client-side certificates for user authentication. Additionally, we will discuss how to create additional realms that behave just like the default realms, by using the realm classes included with GlassFish.

Chapter 9, Enterprise JavaBeans will cover how to implement business logic via stateless and stateful session beans. Additionally, we will explain the concept of container-managed transactions and bean-managed transactions. We will look at the life cycles for the different types of Enterprise Java Beans. We will talk about how to have EJB methods invoked periodically by the EJB container, by taking advantage of the EJB timer service. Finally, we will explain how to make sure that EJB methods are only invoked by authorized users.

Chapter 10, Contexts and Dependency Injection will talk about how JSF pages can access CDI named beans as if they were JSF managed beans. We will explain how CDI makes it easy to inject dependencies into our code. We will discuss how we can use qualifiers to determine what specigic implementation of dependency to inject into our code. Finally, we will look at all the scopes that a CDI bean can be placed into.

Chapter 11, Web Services with JAX-WS will cover how to develop web services and web service clients via the JAX-WS API. We will discuss how to send attachments to a web service. We will explain how to expose an EJB's methods as web services. Finally, we will look at how to secure web services so that they are not accessible to unauthorized clients.

Chapter 12, RESTful Web Services with Jersey and JAX-RS will discuss how to easily develop RESTful web services using JAX-RS—a new addition to the Java EE specification. We will explain how to automatically convert data between Java and XML by taking advantage of the Java API for XML Binding (JAXB). Finally, we will cover how to pass parameters to our RESTful web services via the `@PathParam` and `@QueryParam` annotations.

7

Java Messaging Service

The **Java Messaging API** (**JMS**) provides a mechanism for Java EE applications to send messages to each other. JMS applications do not communicate directly, instead message producers send messages to a destination and message consumers receive the message from the destination.

The message destination is a message queue when the point-to-point (PTP) messaging domain is used, or a message topic when the publish/subscribe (pub/sub) messaging domain is used.

In this chapter, we will cover the following topics:

- Setting up GlassFish for JMS
- Working with message queues
- Working with message topics

Setting up GlassFish for JMS

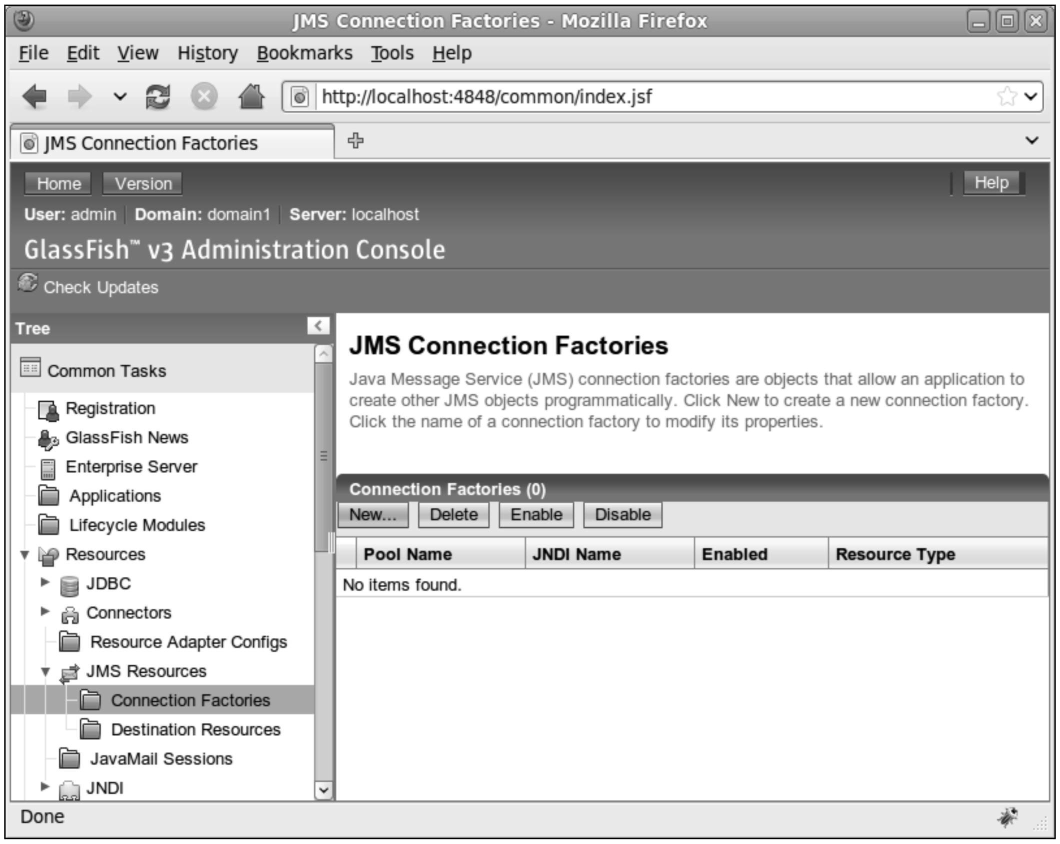
Before we start writing code to take advantage of the JMS API, we need to configure some GlassFish resources. Specifically, we need to set up a **JMS connection factory**, a **message queue**, and a **message topic**.

Setting up a JMS connection factory

The easiest way to set up a JMS connection factory is via GlassFish's web console. Recall from Chapter 1 that the web console can be accessed by starting our domain, by entering the following command in the command line:

```
asadmin start-domain domain1
```

Then point the browser to `http://localhost:4848` and log in:

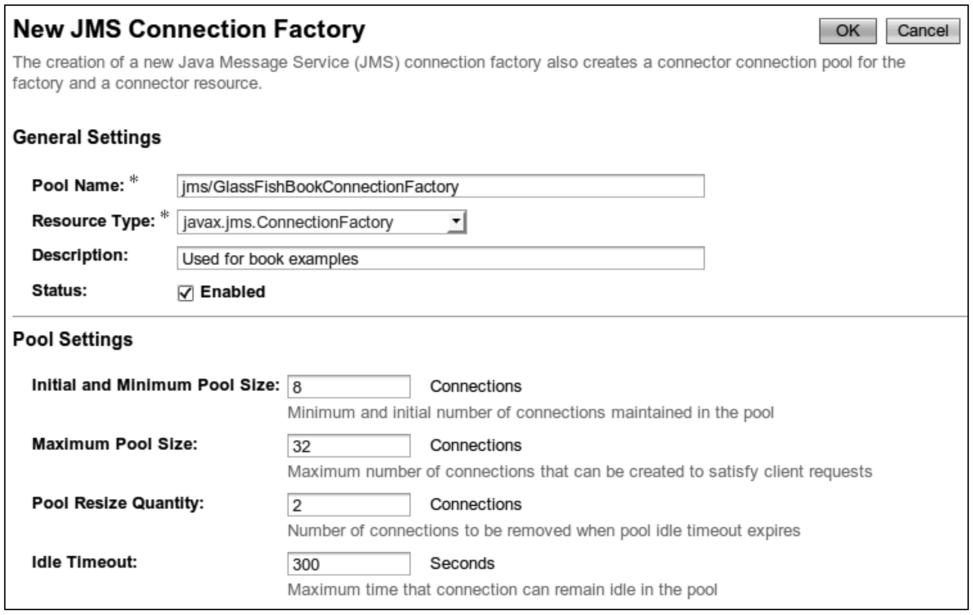


JMS Connection Factories

Java Message Service (JMS) connection factories are objects that allow an application to create other JMS objects programmatically. Click New to create a new connection factory. Click the name of a connection factory to modify its properties.

Pool Name	JNDI Name	Enabled	Resource Type
No items found.			

A connection factory can be added by expanding the **Resources** node in the tree at the left-hand side of the web console, expanding the **JMS Resources** node and clicking on the **Connection Factories** node, then clicking on the **New...** button in the main area of the web console.



New JMS Connection Factory

The creation of a new Java Message Service (JMS) connection factory also creates a connector connection pool for the factory and a connector resource.

General Settings

Pool Name: *	jms/GlassFishBookConnectionFactory
Resource Type: *	javax.jms.ConnectionFactory
Description:	Used for book examples
Status:	☑ **Enabled**

Pool Settings

Initial and Minimum Pool Size:	8	Connections
	Minimum and initial number of connections maintained in the pool	
Maximum Pool Size:	32	Connections
	Maximum number of connections that can be created to satisfy client requests	
Pool Resize Quantity:	2	Connections
	Number of connections to be removed when pool idle timeout expires	
Idle Timeout:	300	Seconds
	Maximum time that connection can remain idle in the pool	

For our purposes, we can take most of the defaults. The only thing we need to do is enter a **Pool Name** and pick a **Resource Type** for our connection factory.

It is always a good idea to use a **Pool Name** starting with "jms/" when picking a name for JMS resources. This way JMS resources can be easily identified when browsing a JNDI tree.

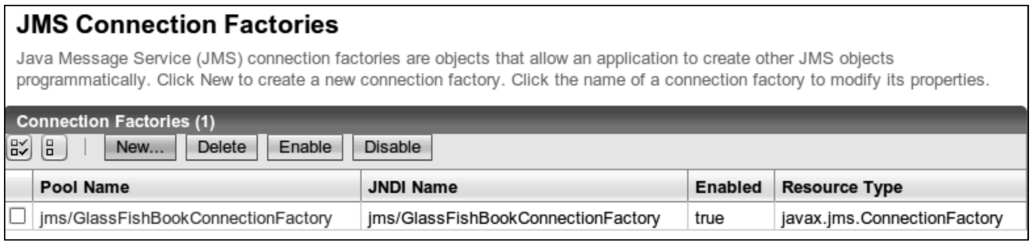
In the text field labeled **Pool Name**, enter **jms/GlassFishBookConnectionFactory**. Our code examples later in this chapter will use this JNDI name to obtain a reference to this connection factory.

The **Resource Type** drop-down menu has three options:

- **javax.jms.TopicConnectionFactory** - used to create a connection factory that creates JMS topics for JMS clients using the pub/sub messaging domain
- **javax.jms.QueueConnectionFactory** - used to create a connection factory that creates JMS queues for JMS clients using the PTP messaging domain
- **javax.jms.ConnectionFactory** - used to create a connection factory that creates either JMS topics or JMS queues

For our example, we will select **javax.jms.ConnectionFactory**. This way we can use the same connection factory for all our examples, those using the PTP messaging domain and those using the pub/sub messaging domain.

After entering the **Pool Name** for our connection factory, selecting a connection factory type, and optionally entering a description for our connection factory, we must click on the **OK** button for the changes to take effect.



JMS Connection Factories

Java Message Service (JMS) connection factories are objects that allow an application to create other JMS objects programmatically. Click New to create a new connection factory. Click the name of a connection factory to modify its properties.

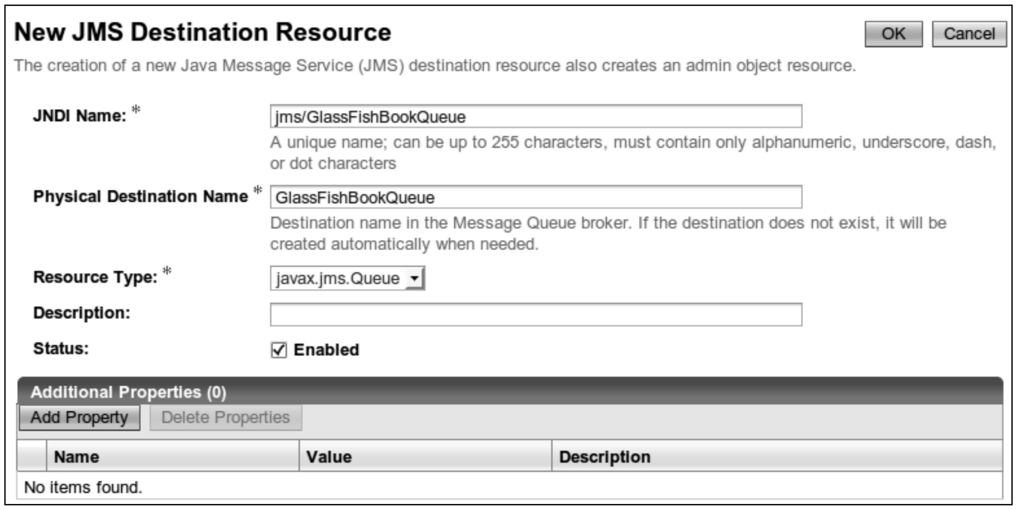
Connection Factories (1)

	Pool Name	JNDI Name	Enabled	Resource Type
☐	jms/GlassFishBookConnectionFactory	jms/GlassFishBookConnectionFactory	true	javax.jms.ConnectionFactory

We should then see our newly created connection factory listed in the main area of the GlassFish web console.

Setting up a JMS message queue

A JMS message queue can be added by expanding the **Resources** node in the tree at the left-hand side of the web console, expanding the **JMS Resources** node and clicking on the **Destination Resources** node, then clicking on the **New...** button in the main area of the web console.



New JMS Destination Resource

The creation of a new Java Message Service (JMS) destination resource also creates an admin object resource.

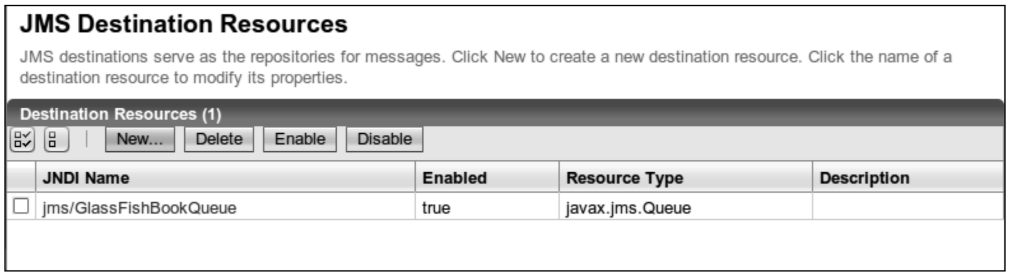
JNDI Name: *	jms/GlassFishBookQueue A unique name; can be up to 255 characters, must contain only alphanumeric, underscore, dash, or dot characters
Physical Destination Name *	GlassFishBookQueue Destination name in the Message Queue broker. If the destination does not exist, it will be created automatically when needed.
Resource Type: *	javax.jms.Queue
Description:	
Status:	✓ Enabled

Additional Properties (0)

	Name	Value	Description
No items found.			

In our example, the JNDI name of the message queue is **jms/GlassFishBookQueue**. The resource type for message queues must be **javax.jms.Queue**. Additionally, a **Physical Destination Name** must be entered. In this example, we use **GlassFishBookQueue** as the value for this field.

After clicking on the **New...** button, entering the appropriate information for our message queue, and clicking on the **OK** button, we should see the newly created queue:



JMS Destination Resources

JMS destinations serve as the repositories for messages. Click New to create a new destination resource. Click the name of a destination resource to modify its properties.

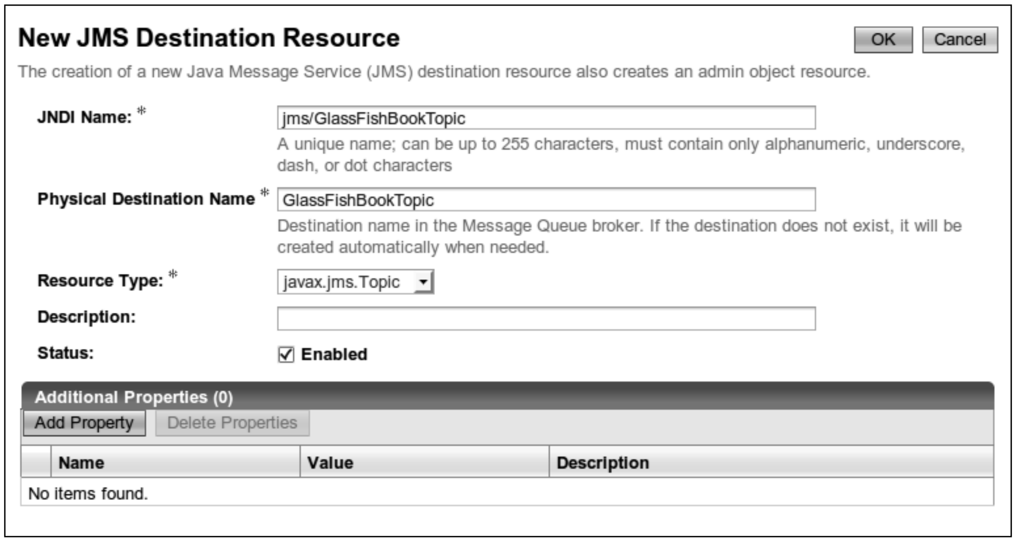
Destination Resources (1)

	JNDI Name	Enabled	Resource Type	Description
☐	jms/GlassFishBookQueue	true	javax.jms.Queue	

Setting up a JMS message topic

Setting up a JMS message topic in GlassFish is very similar to setting up a message queue.

In the GlassFish web console, expand the **Resources** node in the tree at the left hand side, then expand the **JMS Resouces** node and click on the **Destination** Resources node, then click on the **New...** button in the main area of the web console.



New JMS Destination Resource

The creation of a new Java Message Service (JMS) destination resource also creates an admin object resource.

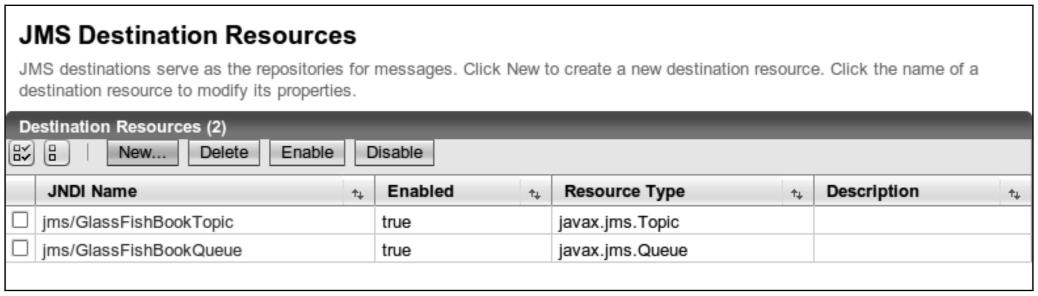
JNDI Name: *	jms/GlassFishBookTopic A unique name; can be up to 255 characters, must contain only alphanumeric, underscore, dash, or dot characters
Physical Destination Name *	GlassFishBookTopic Destination name in the Message Queue broker. If the destination does not exist, it will be created automatically when needed.
Resource Type: *	javax.jms.Topic
Description:	
Status:	☑ Enabled

Additional Properties (0)

	Name	Value	Description
No items found.			

Our examples will use a **JNDI Name** of **jms/GlassFishBookTopic**. As this is a message topic, **Resource Type** must be **javax.jms.Topic**. The **Description** field is optional. The **Physical Destination Name** property is required. For our example, we will use **GlassFishBookTopic** as the value for this property.

After clicking on the **OK** button, we can see our newly created message topic:



JMS Destination Resources

JMS destinations serve as the repositories for messages. Click New to create a new destination resource. Click the name of a destination resource to modify its properties.

Destination Resources (2)

	JNDI Name	Enabled	Resource Type	Description
☐	jms/GlassFishBookTopic	true	javax.jms.Topic	
☐	jms/GlassFishBookQueue	true	javax.jms.Queue	

Now that we have set up a connection factory, a message queue, and a message topic, we are ready to start writing code using the JMS API.

Message queues

Like we mentioned earlier, message queues are used when our JMS code uses the point-to-point (PTP) messaging domain. For the PTP messaging domain, there is usually one message producer and one message consumer. The message producer and the message consumer don't need to run concurrently in order to communicate. The messages placed in the message queue by the message producer will stay in the message queue until the message consumer executes and requests the messages from the queue.

Sending messages to a message queue

The following example illustrates how to add messages to a message queue:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;

public class MessageSender
{
  @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
  private static ConnectionFactory connectionFactory;
```

```
    @Resource(mappedName = "jms/GlassFishBookQueue")
    private static Queue queue;

    public void produceMessages()
    {
      MessageProducer messageProducer;
      TextMessage textMessage;
      try
      {
        Connection connection = connectionFactory.createConnection();
        Session session = connection.createSession(false,
          Session.AUTO_ACKNOWLEDGE);
        messageProducer = session.createProducer(queue);
        textMessage = session.createTextMessage();

        textMessage.setText("Testing, 1, 2, 3. Can you hear me?");
        System.out.println("Sending the following message: "
          + textMessage.getText());
        messageProducer.send(textMessage);

        textMessage.setText("Do you copy?");
        System.out.println("Sending the following message: "
          + textMessage.getText());
        messageProducer.send(textMessage);

        textMessage.setText("Good bye!");
        System.out.println("Sending the following message: "
          + textMessage.getText());
        messageProducer.send(textMessage);

        messageProducer.close();
        session.close();
        connection.close();
      }
      catch (JMSException e)
      {
        e.printStackTrace();
      }
    }
    public static void main(String[] args)
    {
      new MessageSender().produceMessages();
    }
  }
```

Before delving into the details of this code, alert readers might have noticed that this class is a standalone Java application as it contains a `main` method. As this class is standalone, it executes outside the application server. In spite of this, we can see that some resources are injected into it, specifically the connection factory and queue. The reason we can inject resources into this code, even though it runs outside the application server, is because GlassFish includes a utility called `appclient`.

This utility allows us to "wrap" an executable JAR file and allows it to have access to the application server resources. To execute the previous code, assuming it is packaged in an executable JAR file called `jmsptpproducer.jar`, we would type the following command in the command line:

```
appclient -client jmsptpproducer.jar
```

We would then see, after some GlassFish log entries, the following output on the console:

```
Sending the following message: Testing, 1, 2, 3. Can you hear me?
Sending the following message: Do you copy?
Sending the following message: Good bye!
```

The `appclient` executable can be found under `[GlassFish installation directory]/glassfish/bin`. The previous example assumes this directory is in your PATH variable. If it isn't the complete path to the `appclient` executable, it must be typed in the command line.

With that out of the way, we can now explain the code.

The `produceMessages()` method performs all the necessary steps to send messages to a message queue.

The first thing this method does is obtain a JMS connection by invoking the `createConnection()` method on the injected instance of `javax.jms.ConnectionFactory`. Notice that the `mappedName` attribute of the `@Resource` annotation decorating the connection factory object matches the JNDI name of the connection factory we set up in the GlassFish web console. Behind the scenes, a JNDI lookup is made using this name to obtain the connection factory object.

After obtaining a connection, the next step is to obtain a JMS session from said connection. This can be accomplished by calling the `createSession()` method on the `Connection` object. As can be seen in the previous code, the `createSession()` method takes two parameters.

The first parameter of the `createSession()` method is a Boolean indicating if the session is transacted. If this value is `true`, several messages can be sent as part of a transaction by invoking the `commit()` method in the session object. Similarly, they can be rolled back by invoking its `rollback()` method.

The second parameter of the `createSession()` method indicates how messages are acknowledged by the message receiver. Valid values for this parameter are defined as constants in the `javax.jms.Session` interface.

- `Session.AUTO_ACKNOWLEDGE`: indicates that the session will automatically acknowledge the receipt of a message.
- `Session.CLIENT_ACKNOWLEDGE`: indicates that the message receiver must explicitly call the `acknowledge()` method on the message.
- `Session.DUPS_OK_ACKNOWLEDGE`: indicates that the session will lazily acknowledge the receipt of messages. Using this value might result in some messages being delivered more than once.

After obtaining a JMS session, an instance of `javax.jms.MessageProducer` is obtained by invoking the `createProducer()` method on the session object. The `MessageProducer` object is the one that will actually send messages to the message queue. The injected `Queue` instance is passed as a parameter to the `createProducer()` method. Again, the value of the `mappedName` attribute for the `@Resource` annotation decorating this object must match the JNDI name we gave our message queue when setting it up in the GlassFish web console.

After obtaining an instance of `MessageProducer`, the code creates a series of text messages by invoking the `createTextMessage()` method on the session object. This method returns an instance of a class implementing the `javax.jms.TextMessage` interface. This interface defines a method called `setText()`, which is used to set the actual text in the message. After creating each text message and setting its text, they are sent to the queue by invoking the `send()` method on the `MessageProducer` object.

After sending the messages, the code disconnects from the JMS queue by invoking the `close()` method on the `MessageProducer` object, on the `Session` object, and on the `Connection` object.

Although the previous example sends only text messages to the queue, we are not limited to this type of message. The JMS API provides several types of messages that can be sent and received by JMS applications. All message types are defined as interfaces in the `javax.jms` package.

The following table lists all the available message types:

Message type	**Description**
`BytesMessage`	Allows sending an array of bytes as a message.
`MapMessage`	Allows sending an implementation of `java.util.`Map as a message.
`ObjectMessage`	Allows sending any Java object implementing `java.io.Serializable` as a message.
`StreamMessage`	Allows sending an array of bytes as a message. Differs from `BytesMessage` in that it stores the type of each primitive type added to the stream.
`TextMessage`	Allows sending a `java.lang.String` as a message.

For more information on all of these message types, consult their JavaDoc documentation at `http://java.sun.com/javaee/6/docs/api/`.

Retrieving messages from a message queue

There is no point in sending messages from a queue if nothing is going to receive them. The following example illustrates how to retrieve messages from a JMS message queue:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;

public class MessageReceiver
{
  @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
  private static ConnectionFactory connectionFactory;
  @Resource(mappedName = "jms/GlassFishBookQueue")
  private static Queue queue;

  public void getMessages()
  {
    Connection connection;
    MessageConsumer messageConsumer;
    TextMessage textMessage;
    boolean goodByeReceived = false;
    try
    {
      connection = connectionFactory.createConnection();
      Session session = connection.createSession(false,
        Session.AUTO_ACKNOWLEDGE);
      messageConsumer = session.createConsumer(queue);
      connection.start();
      while (!goodByeReceived)
      {
        System.out.println("Waiting for messages...");
        textMessage = (TextMessage) messageConsumer.receive();
        if (textMessage != null)
        {
          System.out.print("Received the following message: ");
          System.out.println(textMessage.getText());
          System.out.println();
        }
        if (textMessage.getText() != null
          && textMessage.getText().equals("Good bye!"))
        {
          goodByeReceived = true;
        }
      }
      messageConsumer.close();
      session.close();
      connection.close();
    }
    catch (JMSException e)
    {
      e.printStackTrace();
    }
  }
  public static void main(String[] args)
  {
    new MessageReceiver().getMessages();
  }
}
```

Just like in the previous example, an instance of `javax.jms.ConnectionFactory` and an instance of `javax.jms.Queue` are injected by using the `@Resource` annotation. Getting a connection and a JMS session is exactly the same as in the previous example.

In this example, we obtain an instance of `javax.jms.MessageConsumer` by calling the `createConsumer()` method on the JMS session object. When we are ready to start receiving messages from the message queue, we need to invoke the `start()` method on the JMS connection object.

Code not receiving messages?

A common mistake when writing JMS messages is to fail to call the `start()` method on the JMS connection object. If our code is not receiving messages it should be receiving, we need to make sure we didn't forget to call this method.

Messages are received by invoking the `receive()` method on the instance of `MessageConsumer` obtained from the JMS session. This method returns an instance of a class implementing the `javax.jms.Message` interface. It must be casted to the appropriate type in order to obtain the actual message.

In this particular example, we placed this method call in a `while` loop, as we are expecting a message that will let us know that no more messages are coming. Specifically, we are looking for a message containing the text `"Good bye!"`. Once we receive said message, we break out of the loop and continue processing. In this particular case, there is no more processing to do. Therefore, all we do is call the `close()` method on the message consumer object, on the session object, and on the connection object.

Just like in the previous example, using the `appclient` utility allows us to inject resources into the code and prevents us from having to add any libraries to the CLASSPATH. After executing the code through the `appclient` utility, we should see the following output in the command line:

```
appclient -client target/jmsptpconsumer.jar
Waiting for messages...
Received the following message: Testing, 1, 2, 3. Can you hear me?

Waiting for messages...
Received the following message: Do you copy?

Waiting for messages...
Received the following message: Good bye!
```

This of course assumes that the previous example was already executed and it placed the messages in the message queue.

Asynchronously receiving messages from a message queue

The `MessageConsumer.receive()` method has a disadvantage—it blocks execution until a message is received from the queue. We can avoid this disadvantage by receiving messages asynchronously via an implementation of the `javax.jms.MessageListener` interface.

The `javax.jms.MessageListener` interface contains a single method called `onMessage`. It takes an instance of a class implementing the `javax.jms.Message` interface as its sole parameter. The following example illustrates a typical implementation of this interface:

```
package net.ensode.glassfishbook;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class ExampleMessageListener implements MessageListener
{
  @Override
  public void onMessage(Message message)
  {
    TextMessage textMessage = (TextMessage) message;
    try
    {
      System.out.print("Received the following message: ");
      System.out.println(textMessage.getText());
      System.out.println();
    }
    catch (JMSException e)
    {
      e.printStackTrace();
    }
  }
}
```

In this case, the `onMessage()` method simply outputs the message text to the console.

Our main code can now delegate message retrieval to our custom `MessageListener` implementation:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.Session;

public class AsynchMessReceiver
{
  @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
  private static ConnectionFactory connectionFactory;
  @Resource(mappedName = "jms/GlassFishBookQueue")
  private static Queue queue;

  public void getMessages()
  {
    Connection connection;
    MessageConsumer messageConsumer;
    try
    {
      connection = connectionFactory.createConnection();
      Session session = connection.createSession(false,
        Session.AUTO_ACKNOWLEDGE);
      messageConsumer = session.createConsumer(queue);
      messageConsumer.setMessageListener(new
        ExampleMessageListener());
      connection.start();

      System.out.println("The above line will allow the "
          + "MessageListener implementation to "
          + "receiving and processing messages from the queue.");
      Thread.sleep(1000);
      System.out.println("Our code does not have to block "
          + "while messages are received.");
      Thread.sleep(1000);
      System.out.println("It can do other stuff "
          + "(hopefully something more useful than sending "
          + "silly output to the console. :)");
      Thread.sleep(1000);
```

```
          messageConsumer.close();
          session.close();
          connection.close();
        }
        catch (JMSException e)
        {
          e.printStackTrace();
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
      public static void main(String[] args)
      {
        new AsynchMessReceiver().getMessages();
      }
    }
```

The only relevant difference between this example and the one in the previous section is that in this case, we are calling the `setMessageListener()` method on the instance of `javax.jms.MessageConsumer` obtained from the JMS session. We pass an instance of our custom implementation of `javax.jms.MessageListener` to this method. Its `onMessage()` method is automatically called whenever there is a message waiting in the queue. By using this approach, the main code does not block while waiting to receive messages.

Executing the previous example (using of course GlassFish's `appclient` utility) results in the following output:

```
appclient -client target/jmsptpasynchconsumer.jar

The above line will allow the MessageListener implementation to receiving
and processing messages from the queue.

Received the following message: Testing, 1, 2, 3. Can you hear me?


Received the following message: Do you copy?


Received the following message: Good bye!


Our code does not have to block while messages are received.

It can do other stuff (hopefully something more useful than sending silly
output to the console. :)
```

Notice how the messages were received and processed while the main thread was executing. We can tell this is the case because the output of the `onMessage()` method of our `MessageListener` can be seen between calls to `System.out.println()` in the primary class.

Browsing message queues

JMS provides a way to browse message queues without actually removing the messages from the queue. The following example illustrates how to do this:

```
package net.ensode.glassfishbook;

import java.util.Enumeration;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueBrowser;
import javax.jms.Session;
import javax.jms.TextMessage;

public class MessageQueueBrowser
{
  @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
  private static ConnectionFactory connectionFactory;
  @Resource(mappedName = "jms/GlassFishBookQueue")
  private static Queue queue;

  public void browseMessages()
  {
    try
    {
      Enumeration messageEnumeration;
      TextMessage textMessage;
      Connection connection = connectionFactory.createConnection();

      Session session = connection.createSession(false,
        Session.AUTO_ACKNOWLEDGE);
      QueueBrowser browser = session.createBrowser(queue);
      messageEnumeration = browser.getEnumeration();
```

```
        if (messageEnumeration != null)
        {
          if (!messageEnumeration.hasMoreElements())
          {
            System.out.println("There are no messages " + "in the
              queue.");
          }
          else
          {
            System.out.println("The following messages are in the
              queue:");
            while (messageEnumeration.hasMoreElements())
            {
              textMessage =
                (TextMessage) messageEnumeration.nextElement();
              System.out.println(textMessage.getText());
            }
          }
        }
        session.close();
        connection.close();
      }
      catch (JMSException e)
      {
        e.printStackTrace();
      }
    }
    public static void main(String[] args)
    {
      new MessageQueueBrowser().browseMessages();
    }
  }
```

As we can see, the procedure to browse messages in a message queue is straightforward. We obtain a JMS connection and a JMS session the usual way, then invoke the `createBrowser()` method on the JMS session object. This method returns an implementation of the `javax.jms.QueueBrowser` interface. This interface contains a `getEnumeration()` method that we can invoke to obtain an enumeration containing all messages in the queue. To examine the messages in the queue, we simply traverse this enumeration and obtain the messages one by one. In the previous example, we simply invoke the `getText()` method of each message in the queue.

Message topics

Message topics are used when our JMS code uses the publish/subscribe (pub/sub) messaging domain. When using this messaging domain, the same message can be sent to all subscribers of the topic.

Sending messages to a message topic

The following example illustrates how to send messages to a message topic:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;

public class MessageSender
{
  @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
  private static ConnectionFactory connectionFactory;
  @Resource(mappedName = "jms/GlassFishBookTopic")
  private static Topic topic;

  public void produceMessages()
  {
    MessageProducer messageProducer;
    TextMessage textMessage;
    try
    {
      Connection connection = connectionFactory.createConnection();
      Session session = connection.createSession(false,
        Session.AUTO_ACKNOWLEDGE);
      messageProducer = session.createProducer(topic);
      textMessage = session.createTextMessage();
```

```
        textMessage.setText("Testing, 1, 2, 3. Can you hear me?");
        System.out.println("Sending the following message: "
            + textMessage.getText());
        messageProducer.send(textMessage);

        textMessage.setText("Do you copy?");
        System.out.println("Sending the following message: "
            + textMessage.getText());
        messageProducer.send(textMessage);

        textMessage.setText("Good bye!");
        System.out.println("Sending the following message: "
            + textMessage.getText());
        messageProducer.send(textMessage);

        messageProducer.close();
        session.close();
        connection.close();
      }
      catch (JMSException e)
      {
        e.printStackTrace();
      }
    }
    public static void main(String[] args)
    {
      new MessageSender().produceMessages();
    }
  }
```

As we can see, this code is nearly identical to the `MessageSender` class we saw when we discussed point-to-point messaging. As a matter of fact, the only lines of code that are different are the ones that are highlighted. The JMS API was designed this way so that application developers do not have to learn two different APIs for the PTP and pub/sub domains.

As the code is nearly identical to the corresponding example in the *Message queues* section, we will only explain the differences between the two examples. In this example, instead of declaring an instance of a class implementing `javax.jms.Queue`, we declare an instance of a class implementing `javax.jms.Topic`. Just like in the previous examples, we use dependency injection to initialize the `Topic` object. After obtaining a JMS connection and a JMS session, we pass the `Topic` object to the `createProducer()` method in the `Session` object. This method returns an instance of `javax.jms.MessageProducer` that we can use to send messages to the JMS topic.

Receiving messages from a message topic

Just as sending messages to a message topic is nearly identical to sending messages to a message queue, receiving messages from a message topic is nearly identical to receiving messages from a message queue.

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;

public class MessageReceiver
{
  @Resource(mappedName = "jms/GlassFishBookConnectionFactory")
  private static ConnectionFactory connectionFactory;
  @Resource(mappedName = "jms/GlassFishBookTopic")
  private static Topic topic;

  public void getMessages()
  {
    Connection connection;
    MessageConsumer messageConsumer;
    TextMessage textMessage;
    boolean goodByeReceived = false;

    try
    {
      connection = connectionFactory.createConnection();
      Session session = connection.createSession(false,
        Session.AUTO_ACKNOWLEDGE);
      messageConsumer = session.createConsumer(topic);
      connection.start();

      while (!goodByeReceived)
      {
        System.out.println("Waiting for messages...");
        textMessage = (TextMessage) messageConsumer.receive();
        if (textMessage != null)
```

```
          {
            System.out.print("Received the following message: ");
            System.out.println(textMessage.getText());
            System.out.println();
          }
          if (textMessage.getText() != null
            && textMessage.getText().equals("Good bye!"))
          {
            goodByeReceived = true;
          }
        }

        messageConsumer.close();
        session.close();
        connection.close();
      }
      catch (JMSException e)
      {
        e.printStackTrace();
      }
    }
    public static void main(String[] args)
    {
      new MessageReceiver().getMessages();
    }
  }
```

Once again, the differences between this code and the corresponding code for PTP are trivial. Instead of declaring an instance of a class implementing `javax.jms.Queue`, we declare a class implementing `javax.jms.Topic`. We use the `@Resource` annotation to inject an instance of this class into our code using the JNDI name we used when creating it in the GlassFish web console. After obtaining a JMS connection and a JMS session, we pass the `Topic` object to the `createConsumer()` method in the `Session` object. This method returns an instance of `javax.jms.MessageConsumer` that we can use to receive messages from the JMS topic.

Using the pub/sub messaging domain as illustrated in this section has the advantage that messages can be sent to several message consumers. This can be easily tested by concurrently executing two instances of the `MessageReceiver` class we developed in this section, then executing the `MessageSender` class we developed in the previous section. We should see console output for each instance, indicating that both instances received all messages.

Just like with message queues, messages can be retrieved asynchronously from a message topic. The procedure to do this is so similar to the message queue version that we will not show an example. To convert the asynchronous example shown earlier in this chapter to use a message topic, simply replace the `javax.jms.Queue` variable with an instance of `javax.jms.Topic` and inject the appropriate instance by using `"jms/GlassFishBookTopic"` as the value of the `mappedName` attribute of the `@Resource` annotation decorating the instance of `javax.jms.Topic`.

Creating durable subscribers

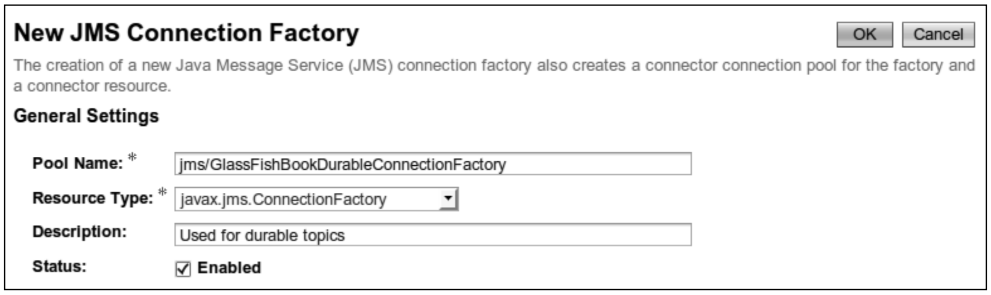
The disadvantage of using the pub/sub messaging domain is that message consumers must be executing when messages are sent to the topic. If the message consumer is not executing at the time, it will not receive the messages, whereas in PTP, messages are kept in a queue until the message consumer executes. Fortunately, the JMS API provides a way to use the pub/sub messaging domain and keep messages in the topic until all subscribed message consumers execute and receive the message. This can be accomplished by creating durable subscribers to a JMS topic.

In order to be able to service durable subscribers, we need to set the `ClientId` property of our JMS connection factory. Each durable subscriber must have a unique client id, therefore a unique connection factory must be declared for each potential durable subscriber.

InvalidClientIdException?

Only one JMS client can connect to a topic for a specific client id. If more than one JMS client attempts to obtain a JMS connection using the same connection factory, a `JMSException` stating that the client id is already in use will be thrown. The solution is to create a connection factory for each potential client that will be receiving messages from the durable topic.

Like we mentioned before, the easiest way to add a connection factory is through the GlassFish web console. Recall that to add a JMS connection factory through the GlassFish web console, we need to expand the **Resources** node on the left hand side, then expand the **JMS Resources** node, click on the **Connection Factories** node, and click on the **New...** button in the main area of the page. Our next example will use the settings displayed in the following screenshot:



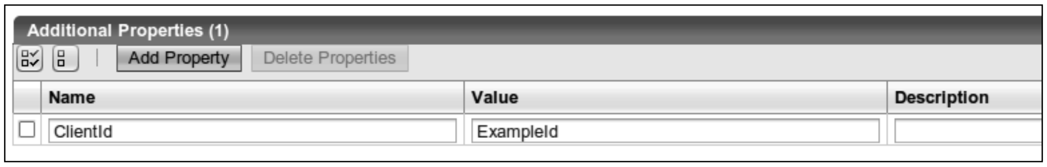
New JMS Connection Factory

The creation of a new Java Message Service (JMS) connection factory also creates a connector connection pool for the factory and a connector resource.

General Settings

Pool Name: *	jms/GlassFishBookDurableConnectionFactory
Resource Type: *	javax.jms.ConnectionFactory
Description:	Used for durable topics
Status:	✓ Enabled

Before clicking on the **OK** button, we need to scroll to the bottom of the page, click on the **Add Property** button, and enter a new property named `ClientId`. Our example will use `ExampleId` as the value for this property.



	Name	Value	Description
☐	ClientId	ExampleId	

Now that we have set up GlassFish to be able to provide durable subscriptions, we are ready to write some code to take advantage of them:

```
package net.ensode.glassfishbook;

import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;

public class MessageReceiver
{
  @Resource(mappedName = "jms/GlassFishBookDurableConnectionFactory")
  private static ConnectionFactory connectionFactory;
  @Resource(mappedName = "jms/GlassFishBookTopic")
  private static Topic topic;
```

```
    public void getMessages()
    {
      Connection connection;
      MessageConsumer messageConsumer;
      TextMessage textMessage;
      boolean goodByeReceived = false;

      try
      {
        connection = connectionFactory.createConnection();
        Session session = connection.createSession(false,
          Session.AUTO_ACKNOWLEDGE);
        messageConsumer = session.createDurableSubscriber(topic,
          "Subscriber1");
        connection.start();

        while (!goodByeReceived)
        {
          System.out.println("Waiting for messages...");
          textMessage = (TextMessage) messageConsumer.receive();
          if (textMessage != null)
          {
            System.out.print("Received the following message: ");
            System.out.println(textMessage.getText());
            System.out.println();
          }
          if (textMessage.getText() != null
            && textMessage.getText().equals("Good bye!"))
          {
            goodByeReceived = true;
          }
        }

        messageConsumer.close();
        session.close();
        connection.close();
      }
      catch (JMSException e)
      {
        e.printStackTrace();
      }
    }
    public static void main(String[] args)
    {
      new MessageReceiver().getMessages();
    }
  }
```

As we can see, this code is not much different from the previous examples whose purpose was to retrieve messages. There are only two differences from the previous examples: the instance of `ConnectionFactory` to which we are injecting is the one we set up earlier in this section to handle durable subscriptions, and instead of calling the `createSubscriber()` method on the JMS session object, we are calling `createDurableSubscriber()`. The `createDurableSubscriber()` method takes two arguments: a JMS `Topic` object to retrieve messages from and a string designating a name for this subscription. This second parameter must be unique between all subscribers to the durable topic.

Summary

In this chapter, we covered how to set up JMS connection factories, JMS message queues, and JMS message topics in GlassFish using the GlassFish web console.

We also covered how to send messages to a message queue via the `javax.jms.MessageProducer` interface.

Additionally, we covered how to receive messages from a message queue via the `javax.jms.MessageConsumer` interface. We also covered how to asynchronously receive messages from a message queue by implementing the `javax.jms.MessageListener` interface.

We also saw how to use these interfaces to send and receive messages to and from a JMS message topic.

We covered how to browse messages in a message queue without removing the messages from the queue via the `javax.jms.QueueBrowser` interface.

Finally, we saw how to set up and interact with durable subscriptions to JMS topics.

: