

CHAPTER 6



Configuring Java Message Service

In the fast-moving world of technology, in which technologies come in and out of fashion, enterprise messaging has enjoyed a long and successful life. Enterprise messaging simplifies the process of inter-application communication, especially across platform boundaries, and it provides an excellent mechanism for systems integration, legacy or otherwise. Another common use of enterprise messaging is to provide asynchronous execution of code, something that's especially useful for executing long-running processes without the need to keep the user waiting around. Finally JMS is also quite useful for replicating data in a distributed environment, however, in most cases your relational database management system (RDBMS) will have much better support for this than JMS and will require less effort to set up.

JMS provides a standard API to access enterprise messaging resources, allowing your application to leverage the power of messaging without having to worry about vendor-specific implementations.

In this chapter we take a detailed look at JMS support in Oracle 10g AS. We'll show you how to configure the server, build a sample application to test your configuration, and use the JMS management tools for practical purposes. We won't cover every single parameter and property that can be set, because this is provided in excellent detail in the documentation. However, we'll point out those properties that sometimes cause confusion, or those places where you may want to consider changing the default setting.

This chapter is split into two main parts. The first part covers Oracle Application Server JMS, the default JMS implementation used by Oracle 10g AS. The second part of the chapter looks at the use of additional JMS providers via the Resource Provider extension mechanism. Specifically, this section will cover Oracle JMS and SonicMQ.

You should note that this chapter assumes an understanding of the JMS API and basic concepts of messaging. Wherever possible we'll explain JMS-specific topics as best as we can, but if there's something you don't understand then we recommend Richard Monson-Haefel and Dave Chappell's superb *Java Message Service* (O'Reilly & Associates, 2000).

JMS in Oracle 10g Application Server

Out of the box, Oracle 10g AS comes with a single, built-in provider for JMS, Oracle 10g AS JMS. Oracle 10g AS JMS complies with version 1.0.2b of the JMS specification, the latest version of JMS being 1.1. On top of the standard provider, you can configure additional providers for Oracle

JMS (OJMS) and other, third-party, messaging solutions. OJMS is the JMS-based interface to the Oracle Streams Advanced Queuing (AQ) feature of the Oracle database. It's important not to confuse Oracle 10g AS JMS with Oracle JMS. Oracle 10g AS JMS is an entirely in-container solution, whereas OJMS is based on the AQ technology and relies on the Oracle database. From a high-level the basic structure of a JMS-based application running on Oracle 10g looks like Figure 6-1.

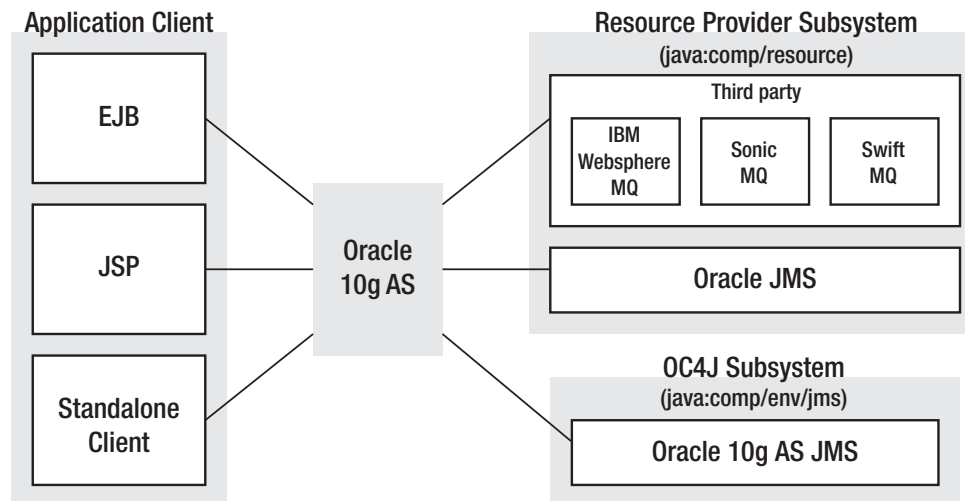


Figure 6-1. High-level overview of JMS in Oracle 10g AS

As you can see from the diagram, whether they're Enterprise JavaBeans (EJB), JSP, Servlet, or Standalone, clients interact with the messaging providers through Oracle 10g AS via the JMS API. The default provider, Oracle 10g AS JMS, is mapped to the `java:comp/env/jms` namespace within the Java Naming and Directory Interface (JNDI), while any external providers, linked in through the Resource Provider subsystem are mapped to the `java:comp/resource` namespace.

The Resource Provider Model

Oracle 10g AS provides a flexible way to plug additional resources into the container. In this context, resources aren't just limited to JMS connection factories, but also JDBC data sources, JavaMail sessions, and URL connection factories. The basis of the Resource Provider is the `ResourceProvider` interface. There's no need for every single JMS provider to create its own implementation of the `ResourceProvider` interface. Instead, you can use the `ContextScanningResourceProvider` class shipped with 10g to integrate any JMS provider that maintains its own JNDI tree. The `ContextScanningResourceProvider` class provides a useful mechanism for binding resources that exist in an external naming context into the local context. Using this mechanism, you can bind the JNDI resources stored in any message provider's context into your application's context. Which is to say that if you're running a third-party JMS provider such as SwiftMQ, you can bind the JMS resources from the SwiftMQ JNDI tree directly into the Oracle 10g AS tree using `ContextScanningResourceProvider`. In this way you simply configure the `ResourceProvider` and then configure SwiftMQ as normal—the `ContextScanningResourceProvider` takes care of the JNDI bindings in Oracle 10g AS for you. The usage of `ContextScanningResourceProvider` is covered in more detail later in the "Configuring Third-Party JMS" section.

Configuring and Using Oracle 10g AS JMS

As discussed earlier, the default JMS provider, Oracle 10g AS JMS, is enabled out of the box. To disable JMS in your environment simply comment out the following line in the server.xml file:

```
<jms-config path="./jms.xml"/>
```

From the preceding code you can probably tell that the path to the Oracle 10g AS JMS configuration file is `jms.xml` and that the file is stored in the same directory as `server.xml`. You can change the location of the configuration file by changing the path attribute to the path you want. However, you should think twice about doing this since you'll be moving the file out of its standard location, which may confuse others who have to manage your server. Also remember that the default configuration keeps the JMS configuration in the same place as all the other configuration files—so why change this?

Standard Configuration

The standard `jms.xml` file looks like this:

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE jms-server PUBLIC "OC4J JMS server"
    "http://xmlns.oracle.com/ias/dtds/jms-server-9_04.dtd">

<jms-server port="9127">

  <!-- Queue bindings, these queues will be bound to their respective
        JNDI path for later retrieval -->
  <queue name="Demo Queue" location="jms/demoQueue">
    <description>A dummy queue</description>
  </queue>

  <!-- Topic bindings, these topics will be bound to their respective
        JNDI path for later retrieval -->
  <topic name="Demo Topic" location="jms/demoTopic">
    <description>A dummy topic</description>
  </topic>

  <!-- path to the log file where JMS-events/errors are stored -->
  <log>
    <file path="../log/jms.log"/>
    <!-- Uncomment this if you want to use ODL logging capabilities
    <odl path="../log/jms/" max-file-size="1000" max-directory-size="10000"/>
    -->
  </log>

  <queue name="jms/OracleSyndicateQueue" location="jms/OracleSyndicateQueue">
    <description>Oracle Syndication Services Queue</description>
  </queue>
```

```

<!--
<queue-connection-factory name="jms/OracleSyndicateQueueConnectionFactory"
  location="jms/OracleSyndicateQueueConnectionFactory"/>
-->
<queue-connection-factory location="jms/OracleSyndicateQueueConnectionFactory"/>

<queue name="jms/OracleUddiReplicationQueue"
  location="jms/OracleUddiReplicationQueue">
  <description>Queue for replication scheduler</description>
</queue>

<!--
<queue-connection-factory
  name="jms/OracleUddiReplicationQueueConnectionFactory"
  location="jms/OracleUddiReplicationQueueConnectionFactory"/>
-->
<queue-connection-factory
  location="jms/OracleUddiReplicationQueueConnectionFactory"/>

<queue name="jms/OracleWebClippingQueue"
  location="jms/OracleWebClippingQueue">
  <description>Queue for Web Clipping</description>
</queue>

<!--
<queue-connection-factory
  name="jms/OracleWebClippingQueueConnectionFactory"
  location="jms/OracleWebClippingQueueConnectionFactory"/>
-->
<queue-connection-factory
  location="jms/OracleWebClippingQueueConnectionFactory"/>
</jms-server>

```

Due to the good use of XML, the configuration file is fairly self-explanatory. The `<queue>` elements are used to configure queues for point-to-point messaging, and the `<topic>` elements are used to configure topics for publish/subscribe messaging. Queues and topics are collectively known as *destinations*. Connection factories are configured using the `<queue-connection-factory>` and `<topic-configuration-factory>` tags.

For those of you who aren't totally familiar with the nuances of JMS, connection factories are used to obtain a connection to particular resource, be it a queue or a topic. The `Queue` and `Topic` interfaces defined in the JMS specification serve as an abstraction of the provider-specific destination name, not of the resource itself. The `QueueConnection` and `TopicConnection` interfaces encapsulate access to the resources identified by `Queue` and `Topic` objects. The `QueueConnectionFactory` and `TopicConnectionFactory` interfaces are designed to encapsulate provider-specific logic for obtaining a connection to the destination and to present a unified API to Java applications. Without these interfaces your application would need to understand all the different, provider-specific, mechanisms for obtaining a connection to a destination.

Unless you want to download and try out the examples from the Oracle website then you can safely remove the `<queue>` and `<topic>` tags for `demoQueue` and `demoTopic`. However, you should leave the other `<queue>` and `<queue-connection-factory>` tags (for the Web Clipping, Syndication Service and the UDDI replication service queues) in place, or at the most, comment them out so you can easily put them back in, because these features use them internally.

To define transactional connection factories you can use the `<xa-connection-factory>`, `<xa-queue-connection>`, and `<xa-topic-connection-factory>` elements.

Use these factories if you want your queues to participate in Java Transaction Service (JTS) managed transactions. This can be useful in applications in which messaging is part of the critical application logic and must be performed as part of larger, atomic blocks, perhaps in conjunction with some kind of database management system (DBMS). Configuring the XA factories is no more complicated than configuring their nontransactional counterparts, but you should only use the XA versions if your messaging application *definitely* requires transaction support. Don't be tempted to use the XA versions "just in case" you might need transaction support in future. Transactions add a performance overhead, which is completely unnecessary if your application doesn't need them. As you'll see, by coding to interfaces (`Connection` and `ConnectionFactory`) and using appropriate configuration, you can completely change the destination configuration without affecting the code.

In addition to whatever you configure in `jms.xml`, you'll find six built-in connection factories that your applications can use. These are available at the following JNDI locations:

- `jms/ConnectionFactory`
- `jms/QueueConnectionFactory`
- `jms/TopicConnectionFactory`
- `jms/XAConnectionFactory`
- `jms/XAQueueConnectionFactory`
- `jms/XATopicConnectionFactory`

Each one of these is bound to an implementation of the corresponding interface. For the most part these predefined connection factories will serve the needs of your application. However, if you need settings other than the defaults, then just configure your own connection factories in `jms.xml`.

In addition to these connection factories, there's also a default queue where all undelivered messages get placed. If you find that some of your messages just aren't getting through and you can't see why, check the `jms/0c4jJmsExceptionQueue` queue. If your messages are in there, then it's likely that your code is correct but that you have a configuration error that's causing the messages to be undeliverable. If the message isn't in this queue, then the likelihood is that the message never reached Oracle 10g AS, which points to a coding error.

Building and Configuring an Application

In the previous section you looked at the basics of configuration. In this section you'll put that knowledge to use and build and configure a full point-to-point JMS-based application. This application will implement two command-line tools: one that sends a message containing

“Hello World” to a queue and another that receives and displays a message from the same queue. Along the way, we’ll show you how you can use the JMS command-line management tools to check that your application is functioning correctly.

Creating a Queue

First, you need to configure the queue for the “Hello World” messages. Do this by adding a `<queue>` tag to the `jms.xml` configuration file, as shown here:

```
<queue name="Hello World Queue" location="jms/helloWorldQueue">
  <description>A queue for the hello world "application"</description>
</queue>
```

You can check that the queue is configured correctly by firing up OC4J and running the following command from the command line:

```
java -cp "$J2EE_HOME/oc4j.jar" com.evermind.server.jms.JMSUtils /
  -username admin -password welcome destinations
```

You should see the `helloWorldQueue` entry appear in the list. As far as server configuration goes, that’s all that’s required. The bulk of the configuration is done in the application configuration files.

NOTE The `JMSUtils` tool, used here to check the existence of the queue, performs lots of useful operations on the Oracle 10g AS JMS server. We’ve described a few of the more useful ones in this chapter, but you’ll find a full list in the documentation.

Creating a Base Class

Both of the command-line tools you’ll be implementing (the sender and the receiver) will need to look up the queue and get `QueueConnection` objects, so it’s easiest to put this functionality into a base class, as follows:

```
package com.apress.oracle10g.jms;

import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

```
public abstract class AbstractJMSClient {

    private static final String FACTORY_NAME =
        "java:comp/env/jms/QueueConnectionFactory";

    private static final String QUEUE_NAME = "java:comp/env/jms/helloWorldQueue";

    private QueueConnectionFactory factory = null;

    protected Queue getQueue() throws NamingException {

        Context ctx = new InitialContext();

        //get the connection factory
        QueueConnectionFactory factory = getQueueConnectionFactory();

        // get the queue
        Queue q = (Queue) ctx.lookup(QUEUE_NAME);

        return q;
    }

    protected QueueConnection getQueueConnection() throws NamingException,
        JMSEException {
        return getQueueConnectionFactory().createQueueConnection();
    }

    private QueueConnectionFactory getQueueConnectionFactory()
        throws NamingException {
        Context ctx = new InitialContext();
        factory = (QueueConnectionFactory) ctx.lookup(FACTORY_NAME);
        return factory;
    }
}
```

This code simply wraps the JNDI lookups for the connection factory and for the queue. Notice that it uses local JNDI names (which start with `java:comp/env`) rather than referring to the actual JNDI locations specified in the configuration file. Later, you'll map these local names to real locations in your deployment descriptors, using `<resource-ref>` and `<resource-ref-mapping>` elements. This indirection gives you the flexibility to change what these names map to without changing any code. It also makes your code more portable between application servers. Of course, if you're just whipping up a quick test client and know that you won't need to change the JNDI locations, then you can just hard-code the actual JNDI locations instead. Check out Chapter 5 for more tips on using JNDI.

Implementing the Sender

Now you can create a simple sender client, as shown here:

```
package com.apress.oracle10g.jms;

import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.NamingException;

public class SendHelloWorld extends AbstractJMSClient {

    public static void main(String[] args) throws Exception {
        SendHelloWorld sender = new SendHelloWorld();
        sender.send();
    }

    public void send() throws JMSException, NamingException {
        // get queue and connection
        Queue q = getQueue();
        QueueConnection connection = getQueueConnection();

        // start...
        connection.start();
        QueueSession session = connection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);

        // send
        QueueSender sender = session.createSender(q);
        TextMessage msg = session.createTextMessage();
        msg.setText("Hello World!");
        sender.send(msg);

        // close
        sender.close();
        session.close();
        connection.close();

        System.out.println("Message Sent");
    }
}
```

The preceding code simply uses the queue and queue connection provided by the base class to create and send a textual message containing the words “Hello World.”

Configuring an Application Client

At this point you’re almost ready to try out the code; all that’s left to do is to create the application configuration files. The first one to create is `jndi.properties`, which contains the JNDI connection details, as follows:

```
java.naming.factory.initial=com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=ormi://localhost/
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

The `jndi.properties` specifies the name of the JNDI context factory class, along with the URL, username, and password required to connect to the JNDI server. If you have OC4J running on a different machine or if you need to use a different username or password, then you should make the appropriate changes in your file.

Next up, as shown in the following code sample, is the standard J2EE `application-client.xml` file, which is required by all standalone J2EE client applications:

```
<?xml version="1.0"?>
<!DOCTYPE application-client PUBLIC
    "-//Sun Microsystems, Inc.//DTD J2EE Application Client 1.2//EN"
    "http://java.sun.com/dtd/application-client_1_3.dtd">
<application-client>
  <display-name>Hello World</display-name>
  <resource-ref>
    <res-ref-name>jms/QueueConnectionFactory</res-ref-name>
    <res-type>javax.jms.QueueConnectionFactory</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>
  <resource-env-ref>
    <resource-env-ref-name>jms/helloWorldQueue</resource-env-ref-name>
    <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
  </resource-env-ref>
</application-client>
```

In this file you’re telling OC4J that this application client will need to be able to look up a connection factory at `jms/QueueConnectionFactory` and a queue at `jms/helloWorldQueue`. Note that the names used here are the local JNDI names (minus the `java:comp/env` prefix) that your application uses, not the real locations specified in `jms.xml`.

Next, create an `orion-application.xml` file. Inside it, map these local JNDI names to real locations, as follows:

```
<?xml version="1.0"?>
<!DOCTYPE orion-application-client PUBLIC
"-//Evermind//DTD J2EE Application-client runtime 1.2//EN"
"http://xmlns.oracle.com/ias/dtds/orion-application-client.dtd">

<orion-application-client>
  <resource-ref-mapping name="jms/QueueConnectionFactory"
    location="jms/QueueConnectionFactory"/>
  <resource-env-ref-mapping name="jms/helloWorldQueue"
    location="jms/helloWorldQueue"/>
</orion-application-client>
```

Running the Client

Once the code is compiled you can run it with the following:

```
java -cp "classes:src/META-INF:$J2EE_HOME/oc4j.jar" \
    com.apress.oracle10g.jms.SendHelloWorld
```

This command assumes that the class files for your application are under the `classes` directory and that your `application-client.xml` and `orion-application.xml` configuration files are in the `src/META-INF` directory. If you've used a different directory structure, edit the `cp` (classpath) parameter accordingly.

If the application runs successfully you'll receive a "Message Sent" message. If you don't get this message, check that the classpath is specified correctly, that OC4J is running, and that the queue is configured correctly. To check that the message actually made it to your queue, you can use the `browse` command of `JMSUtils`, as shown here:

```
java -cp "$J2EE_HOME/oc4j.jar" com.evermind.server.jms.JMSUtils -username admin /
-passwd welcome browse "Hello World Queue"
```

The important thing to notice about the use of the `browse` command is that instead of the using the JNDI name for the queue, you have to use the "friendly" name that you specified in the `name` attribute of the `<queue>` tag. This means that if your name contains spaces, you need to enclose it in quotes. For this reason it's generally easier to use names that don't contain spaces. Running the `browse` command should yield output like this:

```
<textmessage>
  <header>
    <JMSCorrelationID value="" />
    <JMSDeliveryMode value="PERSISTENT" />
    <JMSDestination value="Queue[Hello World Queue]" />
    <JMSExpiration value="0" />
    <JMSMessageID
value="ID:0c4jJMS.Message.python.local.6b352c:fc0725fdff:-8000.4" />
    <JMSPriority value="4" />
```

```

        <JMSTimestamp value="1082451689466" />
        <JMSType value="" />
    </header>
    <properties>
        <string key="JMSXConsumerTXID" value="" />
        <int key="JMSXDeliveryCount" value="1" />
        <string key="JMSXProducerTXID" value="" />
        <long key="JMSXRcvTimestamp" value="1082451693818" />
        <string key="JMSXUserID" value="" />
        <string key="JMS_OC4J_Type" value="textmessage" />
    </properties>
    <textbody>
        <string value="Hello World!" />
    </textbody>
</textmessage>

```

1 messages processed

Near the bottom of the message you should see the `<textbody>` element, which contains the value of the message in a `<string>` tag. This output contains a lot of information about the message that can be used for diagnostic purposes if you're experiencing problems with your JMS application.

Implementing the Receiver

Now that you have a message in the queue you can build a receiver to consume the message and display it in the console. The code for the `ReceiveHelloWorld` class is very similar to that of the `SendHelloWorld` class, as shown here:

```

package com.apress.oracle10g.jms;

import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueReceiver;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.NamingException;

public class ReceiveHelloWorld extends AbstractJMSClient{

    public static void main(String[] args) throws Exception {
        ReceiveHelloWorld receiver = new ReceiveHelloWorld();
        receiver.receive();
    }
}

```

```

public void receive() throws NamingException, JMSEException {
    // get queue and connection
    Queue q = getQueue();
    QueueConnection connection = getQueueConnection();

    // start...
    connection.start();
    QueueSession session = connection.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);

    // receive
    QueueReceiver receiver = session.createReceiver(q);
    TextMessage msg = (TextMessage)receiver.receiveNoWait();

    if(msg != null)
        System.out.println("Received: " + msg.getText());
    else
        System.out.println("No message received");

    // close
    receiver.close();
    session.close();
    connection.close();
}
}

```

The main difference, of course, is that this class receives a message from the queue instead of sending one to it. Run the receiver using the following command in order to receive the message you just sent:

```

java -cp "bin:src/META-INF:$J2EE_HOME/oc4j.jar" \
    com.apress.oracle10g.jms.ReceiveHelloWorld

```

You should receive a message informing you whether or not a message was received from the queue. Running the JMSutils browse command again should now show that no messages are left in the queue.

As you can see, getting a basic JMS-based application up and running is easy. In the rest of this section you'll look at additional configuration options.

Configuring File Persistence

The queue you created in the previous example was an in-memory queue. This means that if you send a message to the queue and then shut the server down, the message will be gone when you restart the server. In some cases this may be the desired effect, but in others you may wish to have some level of protection for failures in the server. Using file persistence, you can instruct Oracle 10g AS to persist messages in a particular queue or topic to the file system.

To see this in action, you should first try sending a message to `helloWorldQueue` using the sender, and then shut down and restart OC4J. When you restart, use the `JMSutils browse` command to check the contents of the queue. You should find nothing there.

To turn on file persistence for the `helloWorldQueue`, make the following modification to the `jms.xml` file:

```
<queue name="Hello World Queue" location="jms/helloWorldQueue"
      persistence-file="helloWorld.queue">
  <description>A queue for the hello world "application"</description>
</queue>
```

Notice that the `<queue>` element now contains a `persistence-file` attribute that contains the path to the file where messages should be stored. If the file doesn't exist, then Oracle 10g AS will create it for you. The path specified can be either an absolute path or a path relative to the default persistence location specified in `application.xml`. You can change the default location by modifying the following entry in `application.xml`:

```
<persistence path="../persistence"/>
```

Once you've updated `jms.xml` to specify a persistence file, restart the application server and send another message to the `helloWorldQueue`. Use the `JMSutils browse` command to verify that the message made it to the queue. Then restart the server and use the `browse` command to verify that unlike before, the message is still in the queue.

Persistence File Management

The file format used by Oracle 10g AS is a vaguely decipherable binary format. Open the file up in a text editor and you'll be able to make out the gist of each message in between a mass of undisplayable characters. Although you could probably edit these files there would be no guarantee that it would work all the time and we certainly wouldn't recommend it.

You're free to copy or delete persistence files when they aren't in use, but doing so while Oracle 10g AS is using them will result in an unrecoverable error.

Provided you haven't disabled locking, a lock file that's used to synchronize access across multiple instances of Oracle 10g AS will accompany each queue. Lock files have the same name as the queue persistence file; they just have `.lock` bolted on at the end. In general, you should avoid playing with these files while the server is running. A normal shutdown of Oracle 10g AS will clean up the lock files used by the queues. However if Oracle 10g AS terminates without a proper shutdown, either because of an error or because you've killed the process, then some lock files may be left lying around. You should delete these before you restart the Oracle 10g AS server, since the state they contain will be inconsistent with the runtime environment.

Be aware that there's a limit on the number of active file handles that Oracle 10g AS will maintain for persistent destinations. By default this is set to 64, meaning that Oracle will only maintain 64 active file handles for persistence. If you have more than this number of persistent queues then Oracle will open and close files as appropriate, which can slow down message delivery. You can change the maximum number of allowable file handles by setting the `oc4j.jms.maxOpenFiles` system property when starting Oracle 10g AS. This is useful if your operating system won't allow 64 open file handles for the process or if you wish to increase the number (within the bounds of the operating system) to increase performance.

If you have a large number of persistent queues, then you should consider using OJMS (covered later in this chapter) instead of the default Oracle 10g AS JMS provider. OJMS is built on top of the Oracle database and is more reliable than simple file-based persistence.

Configuring Hosts and Ports

By default Oracle 10g AS JMS will listen on port 9127 for all network interfaces in the machine. You can change the port number by editing the port attribute of the root `<jms>` tag in `jms.xml` as follows:

```
<jms-server port="9127">
```

If you want Oracle 10g AS JMS to listen on a single network interface, then you can also set the host attribute of the `<jms>` tag, as shown here:

```
<jms-server port="9127" host="192.168.0.254"/>
```

By default, connections obtained through connection factories will use the local JMS services configured in the `jms-server` element. To force all connections to go to a different server, adjust the host and port attributes on the `connection-factory` element. For example, to force all connections obtained through `myQueueConnectionFactory` to go to the server listening on `192.168.1.57:9127`, you might specify the following:

```
<queue-connection-factory location="jms/myQueueConnectionFactory"
                        host="192.168.1.57" port="9127"/>
```

This can be useful if you wish to redirect JMS operations to a different server, and is a valid reason for specifying a custom `connection-factory` rather than using the default factories provided by Oracle 10g AS JMS.

Configuring Logging

You can configure email and file-based logging using the `<log>` element within `jms.xml`. By default, OC4J JMS logs to `/j2ee/home/log/jms.log`. This file is specified relative to `jms.xml`, like this:

```
<log><file path="../log/jms.log" /></log>
```

If a mail session has been configured, events can also be logged to an email address, like this:

```
<log>
  <file path="../log/jms.log" />
  <mail address="admin@somewhere.com" />
</log>
```

By default the logging output includes WARNING, ERROR, and CRITICAL events, meaning that it only contains information when something goes wrong. You can increase the verbosity of the log by setting the `oc4j.jms.debug` system property to true. This will include all NORMAL level events—basically a trace of the actions taken by Oracle 10g AS JMS. When you set `oc4j.jms.debug` to true the log messages are written to `stderr` as well as to the log file.

None of the log levels will write the message contents to the log. If you want to check the contents of a queue, then you need to use the JMSutils command-line tool that you saw earlier.

Configuring Oracle JMS

In the previous section you saw how to create and configure an Oracle 10g AS JMS-based application. In this section you'll take that application and make it run against the Oracle JMS provider. As we mentioned earlier, Oracle 10g AS provides a standard mechanism with which you can plug in additional JMS providers: the `ResourceProvider` interface. Oracle 10g AS comes complete with an implementation of this interface, `oracle.jms.OracleJMSContext`, which enables Oracle JMS to be used as a JMS provider.

Oracle JMS is based on the Oracle Streams Advanced Queuing (AQ) feature of the Oracle database. AQ is a full messaging system built on top of the Oracle database system supporting both point-to-point and publish/subscribe messaging. AQ has a vast array of features that improve upon plain JMS, but one of the biggest benefits of AQ is the fact that it has many different client libraries that allow it to be accessed from Java, C, PL/SQL and many more. This makes AQ a great way to integrate applications that were written using different languages. A full discussion of AQ is outside the scope of this chapter and this book, but we'll demonstrate how to configure a queue on a database that's running AQ in the following section.

Setting Up the Queue

Configuring an OJMS queue is mostly a database process. The first step is to create a user with the appropriate permissions, and then, using the `DBMS_AQADM` package, create the tables and queues in that user's schema. The following SQL script will create the user `jmsuser` and give him the appropriate permissions. It will then create the queue and queue table within the `jmsuser` schema.

```
DROP USER jmsuser CASCADE ;
```

```
GRANT connect, resource,AQ_ADMINISTRATOR_ROLE TO jmsuser IDENTIFIED BY pwd ;
GRANT execute ON sys.dbms_aqadm TO jmsuser;
GRANT execute ON sys.dbms_aq TO jmsuser;
GRANT execute ON sys.dbms_aqin TO jmsuser;
GRANT execute ON sys.dbms_aqjms TO jmsuser;
```

```
connect jmsuser/jmsuser;
```

```
BEGIN
```

```
  DBMS_AQADM.CREATE_QUEUE_TABLE(
    Queue_table      => 'helloWorldQueueTbl',
    Queue_payload_type => 'SYS.AQ$_JMS_MESSAGE',
    sort_list => 'PRIORITY,ENQ_TIME',
    multiple_consumers => false,
    compatible       => '8.1.5');
```

```
  DBMS_AQADM.CREATE_QUEUE(
```

```

Queue_name      => 'helloWorldQueue',
Queue_table     => 'helloWorldQueueTbl');

DBMS_AQADM.START_QUEUE(
  queue_name     => 'helloWorldQueue');
END;
/
quit;
```

If you want to create a topic destination rather than a queue, just set the `multiple_consumers` parameter of the `CREATE_QUEUE_TABLE` procedure to true.

Configuring Oracle 10g AS

Configuring Oracle 10g AS for the Oracle JMS provider requires modifications to two configuration files: the global `application.xml` and `data-sources.xml`. The first step is to configure the resource provider in the `application.xml` file, as shown here:

```

<resource-provider class="oracle.jms.OjmsContext" name="oraclejms">
  <description> OJMS/AQ </description>
  <property name="datasource" value="jdbc/OracleDS"></property>
</resource-provider>
```

Though the description isn't important, the name you specify in the `<resource-provider>` tag is, because it will be used later as part of the JNDI name for resources managed by the Oracle JMS provider.

The `OjmsContext` resource provider also requires you to specify a `<property>` element with the name of the data source it should use. This should point to the database holding the Oracle AQ queue. Therefore, it's necessary to configure a data source in the `data-sources.xml` file that points at that Oracle database server, as follows:

```

<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/OracleCoreDS"
  xa-location="jdbc/xa/OracleXADS"
  ejb-location="jdbc/OracleDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="jmsuser"
  password="pwd"
  url="jdbc:oracle:thin:@meerkat:1521:orcl"
  inactivity-timeout="30"
/>
```

If you used the SQL script described earlier to create the queues in Oracle AQ, then the username and password will be the same—otherwise you should change them to match your environment. The URL for your Oracle database will undoubtedly be different than the one configured here, so make sure that you change that to match your environment as well.

At this point, the resource provider you've configured should be available globally to all applications. If you wish to set up a resource provider specific to an individual application, move the `<resource-provider>` declaration from the global `application.xml` file to that application's `orion-application.xml` file.

Configuring the Hello World Application

All that's left is to reconfigure the application so that it uses the Oracle JMS queues instead of the Oracle 10g AS JMS queues. Because we used local JNDI names in the Java code and in the `application-client.xml` file, all that's required is a change to the `orion-application-client.xml` file, as shown here:

```
<resource-ref-mapping name="jms/queueConnectionFactory"
  location="java:comp/resource/oraclejms/QueueConnectionFactories/aqQcf"/>
<resource-env-ref-mapping name="jms/helloWorldQueue"
  location="java:comp/resource/oraclejms/Queues/jmsuser.helloWorldQueue"/>
```

You'll notice that we've changed the location attributes to the values required by Oracle JMS. To build these location strings, start with the prefix, `java:comp/resource`, which is used by all resource providers. Follow that with the resource provider's name: in this case, `oraclejms`. After that the remainder of the location is specific to the resource provider. For Oracle JMS, queue connection factories are accessed under the `QueueConnectionFactories` name and queues are accessed under the `Queues` name. Likewise you can access topic connection factories and topics using `TopicConnectionFactories` and `Topics`.

The name used for the queue connection factory in the final part of the location attribute is unimportant, unless you require access to a specifically configured connection factory. The configuration of connection factories in AQ is outside the scope of this chapter, but you'll find more details in the online AQ reference at http://download-west.oracle.com/docs/cd/B13789_01/server.101/b10728/toc.htm.

TIP If you're having problems getting this example to work, start OC4J with the `datasource.verbose` system property set to `true`. This will output some fairly detailed information about the data sources and will help you to trace the behavior of the data source that's configured for the AQ server.

Before you run the sender to test the Oracle queue, run a `SELECT COUNT(*) FROM helloWorldQueueTbl` query on the `helloWorldQueueTbl` table in Oracle to make sure the current message count is zero. After running the sender, run the `SELECT` query again to check that the message is in the queue.

Configuring Third-Party JMS

If you want to access queues and topics that aren't in Oracle 10g AS JMS or Oracle JMS, you can configure the application server to use additional JMS providers. Oracle 10g AS provides a `ResourceProvider` interface as a standardized mechanism for linking in these third-party

providers. However, rather than relying on individual manufacturers to build adapters for their products, Oracle provides an out-of-the-box `ResourceProvider` implementation the `ContextScanningResourceProvider` class, which binds objects from an external JNDI tree into the local tree. This mechanism works well because most JMS providers tend to bind their objects into a JNDI tree.

Using `ContextScanningResourceProvider`, you can use a third-party provider for the Hello World application. We've chosen to use SwiftMQ because it's available as a small evaluation download for both Windows and UNIX platforms—it even runs on Mac OS X. Plus the installation is a breeze! Visit the SwiftMQ website at www.swiftmq.com and download the evaluation version of the SwiftMQ Router. The download is available in a TAR format for UNIX and a ZIP format for Windows. The scripts in each release are specific to the operating environment.

To install the SwiftMQ Router, simply extract the archive to a suitable location and you're done. SwiftMQ already comes with some preconfigured queues for testing, so rather than create a new one, just fire up the first router by running `${SWIFTMQ_HOME}/scripts/unix/smqr1.sh` script. For Windows users, the location for the start script is `${SWIFTMQ_HOME}/scripts/win32/smqr1.sh`.

The next step is to add an additional `<resource-provider>` entry to the `global application.xml` file, as shown here:

```
<resource-provider
  class="com.evermind.server.deployment.ContextScanningResourceProvider"
  display-name="SwiftMQ resource"
  name="swiftmq" >
  <description>
    SwiftMQ resource provider.
  </description>
  <property name="java.naming.factory.initial"
    value="com.swiftmq.jndi.InitialContextFactoryImpl" />
  <property name="java.naming.provider.url"
    value="smqp://localhost:4001" />
  <property name="resource.names"
    value="testqueue@router1,plaintext@router1" />
</resource-provider>
```

From the preceding code you can see that the `ContextScanningResourceProvider` is configured with three properties. First, it requires the `java.naming.factory.initial` and `java.naming.provider.url` properties—without them it won't know which JNDI tree to search for resources. The `resource.names` property specifies which resources in SwiftMQ's JNDI tree should be bound into the local JNDI context. Here, we've included the `testqueue` queue and the `plaintext` connection factory from router 1.

The next step is to point the local names used by the Hello World application to the SwiftMQ resources instead of the Oracle 10g AS JMS or Oracle JMS resources that they're currently bound to. As mentioned before, this is simply a matter of making the appropriate change in `orion-application.xml`:

```
<resource-ref-mapping name="jms/queueConnectionFactory"
  location="java:comp/resource/swiftmq/plaintext@router1"/>
<resource-env-ref-mapping name="jms/helloWorldQueue"
  location="java:comp/resource/swiftmq/testqueue"/>
```

Again, you'll notice that the JNDI names for the external resources have the prefix `java:comp/` resource followed by the resource provider name that's specified in `application.xml` and then the provider-specific name.

Now for the interesting part—the documentation states that you should be able to simply drop the `swiftmq.jar` file into the `$J2EE_HOME/lib` directory, start up the server, and everything will be OK. Unfortunately, that's not the case. After placing the `.jar` file in every `lib` directory in the OC4J directory, after attempting to configure it as an additional `<library>` within `application.xml`, and after starting up, OC4J still informed me that it was unable to find the `com.swiftmq.jndi.InitialContextFactoryImpl` class. We tried this on OC4J on Mac OS X, Linux and Windows as well as on Oracle 10g AS Standard for Windows, but it didn't work on any of those installations. In the end we decided to take a different approach and instructed Java to load the `swiftmq.jar` into the classloader hierarchy above OC4J—that way it would be available to OC4J at startup.

To do this, specify the `bootclasspath` option when starting OC4J, as follows:

```
java -Xbootclasspath/a:$SWIFT_MQ/jars/swiftmq.jar:$SWIFT_MQ/jars/jms.jar \  
-jar oc4j.jar
```

You'll notice that we also load the `jms.jar` file supplied with SwiftMQ. This is because SwiftMQ requires classes that aren't available in the OC4J JMS distribution.

You should now be able to run your Hello World sender and receiver applications in order to send and receive messages using the SwiftMQ message queue.

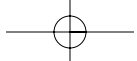
Picking Your Provider

Choosing which kind of provider to use can be a difficult decision, but in most cases following a few simple steps can make the decision easier.

First, it's perfectly acceptable to use a different provider for development and production—just be sure to test on the production provider during development. In most cases Oracle 10g AS JMS will prove the best choice for development. Oracle 10g AS JMS is fast and simple to configure, making it an ideal proposition for developers who would rather focus on coding than on wrestling with their development environment. Another reason for using Oracle 10g AS JMS for development is that it can be configured and monitored in isolation. Using Oracle JMS for development means that somebody has to install and administer the Oracle instance used for AQ.

Choosing which provider to use for production is more difficult, however, in some cases your environment may have already made the decision for you. For example, if you're using Oracle 10g AS on top of a database other than Oracle then you can automatically rule out the use of Oracle JMS. Furthermore, many systems are built on top of existing infrastructure that may include a substantial investment in messaging techniques. If this is the case then it's often wise, from a business point of view, to use the messaging infrastructure already in place, especially if you'll be integrating with existing applications.

When choosing between Oracle 10g AS JMS and Oracle JMS, look at your performance, reliability, management, and integration needs. If your application only uses messaging internally and persistence isn't important, then Oracle 10g AS JMS is your fastest option. However, if you're using persistence, this performance difference disappears and Oracle JMS provides far more in terms of reliability, ease of backup, and management.



Like other external providers, Oracle JMS is also very useful when you want to use messaging as a mechanism for interacting with other applications, especially those written in languages other than Java. Because Oracle JMS sits on top of AQ, you can use any AQ client library to access the messages sent to Oracle JMS from your application. This has many benefits. Let's say you have an application that you want to save data to the database and queue off a message to another source. You can either queue the message in your JDBC call or use a trigger or stored procedure. The big advantage to this over an external message provider is that you know for sure that you don't queue a message if it doesn't get in your database and vice versa. You also have the ability to make take a point-in-time backup and be guaranteed through Oracle's facilities to have perfect synchronization.

Summary

Support for JMS in the Oracle Application Server is excellent. This chapter has covered the practical aspects of configuring JMS using Oracle 10g AS JMS, Oracle JMS, and third-party providers. The decision of which provider to use can be a difficult one, but if you use the knowledge you gained from this chapter, you should be able to make it easily.

