

Understanding and Using Java Messaging Service

By Henry Yamauchi, hyamauchi@skillbuilders.com

Messaging systems have long been recognized as being flexible and reliable. Thus they have found wide acceptance in many industries. Java Messaging Service (JMS) provides a standard, vendor independent API for accessing messaging systems using Java. This article will provide the reader with an understanding of the API, an example of how to use it, and pointers to additional information resources.

Application programmers are accustomed to passing data to functions by assigning values to its arguments. Subsequently the function is invoked and the application will wait until the function returns. This pattern was extended for distributed computing using RPCs (Remote Procedure Calls) so programmers are able to develop distributed systems using this same familiar function call paradigm. This mode of programming is called *synchronous* because the client making the function (or RPC) call must wait until the function finishes executing. Most popular distributed systems today such as DCE, CORBA and DCOM are based on such synchronous calls.

Although synchronous programming is easy to code and understand, in many situations it may not be the best way to build distributed applications. The major disadvantage of synchronous systems is that the server must be running at the time the client makes the call. This creates a dependency between clients and servers, which is further exacerbated when multiple systems are calling each other in a chain. In this last case, all the systems in the chain must be up in order for the initial request to succeed.

Fortunately, messaging provides another way of solving this problem. Messaging clients can send requests to an agent and continue working even if the destination server happens not to be running at the moment the requests are made. Once the server is brought up, it will process those requests and (possibly) send results back to the client. If, in its turn, the client is not up at the time the responses are sent, these will be stored until the time the client is ready to receive them. The messaging paradigm completely decouples clients from servers, breaking the chain of dependency mentioned previously. This mode of programming is called *asynchronous*.

Synchronous and asynchronous requests can be better understood if we compare them with the way people communicate in their day-to-day lives. The synchronous way of programming can be compared to a phone call. When someone calls another he/she will not be able to conduct his/her business unless the called party is there to take the call. If this were the case he/she would need to keep calling the other party (polling, in programming jargon) until he/she can get someone to take the call.

Of course, this was the state of affairs until the advent of the answering machine. With answering machines the caller is able to leave his/her request(s) in the machine and go about his/her business. The called person is also free to review the messages at his/her leisure rather than being interrupted in the middle of his/her work to answer calls. Using an answering machine is the equivalent of performing an asynchronous request.

Messaging systems however, usually go beyond the above-mentioned convenience and offer additional features that make them even more compelling. One such feature offered by most vendors is guaranteed delivery of messages. This means that all requests sent by a client are guaranteed to be delivered to the destination server, eventually, and that no messages will be lost. This feature allows one to use messaging for critical applications such as financial systems where the loss of even a single transaction may have grave consequences.

Messaging systems became very popular and several vendors compete in this area. The most popular products are MQSeries from IBM, TIBCO from TIBCO Software, Oracle's AQ, MSMQ from Microsoft. In addition there are several other less known products such as Sun JMQ, SonicMQ, FioranoMQ, SwiftMQ and others. The latter is unique in that it is a commercial and yet it offers a free implementation of JMS, which is the reason why we will be using it in our examples. SwiftMQ can be downloaded from the company's web site: www.swiftmq.com

In all of the above products functionality is similar but until the advent of JMS, they all had proprietary APIs, which made it hard to port applications written for one of these products to work with another. Skills transfer was also made difficult since their APIs differed substantially from each other. JMS (Java Messaging Service) was developed to create a single Java API that would allow programming of messaging applications in a consistent way regardless of the product being used.

Although JMS has been part of the J2EE framework since version 1.1, its recent inclusion in Message Driven Beans in J2EE 1.3 will likely spur a substantial increase in the use of messaging applications. This makes knowing this API indispensable for most developers. Messaging systems can also be written using JMS alone, independently from J2EE thus adding another reason to be aware of the capabilities of this technology.

Links to messaging software vendors:

www.swiftmq.com

www.tibco.com

<http://www-4.ibm.com/software/ts/mqseries/messaging/>

<http://www.fiorano.com/>

http://www.oracle.com/ip/dep/otn/database/oracle9i/index.html?aq_home.html

<http://www.microsoft.com/msmq/>

<http://www.sun.com/forte/iplanet/jmq.html>

<http://www.sonicsoftware.com/>

The JMS API

The JMS API is a relatively simple one when compared to others such as JDBC. Like JDBC it consists of mostly Java interfaces whose implementation is provided by a vendor.

Messaging systems can be classified broadly into two types: point-to-point and publish-subscribe. In point-to-point systems a client sends a message to a single application. In publish-subscribe systems a client (in this case called a publisher) sends a message that can be received by multiple applications (called subscribers). In this article we will be covering the point-to-point topology. In a follow up article we will cover publish-subscribe.

A central concept when dealing with point-to-point messaging systems is the concept of a queue. When clients send a message to be processed by another application it does so by an intermediate server (the messaging server). This server receives the message and places it in a repository known as a queue. The physical characteristics of this repository vary from vendor to vendor. Like any queue this is usually a FIFO (first in, first out) data structure. Another application can then connect to the server and get this message from the queue to process it. Therefore you can already guess that the API calls to put and get a message from a queue are the most important ones in point-to-point applications.

Links to useful information about JMS:

<http://java.sun.com/products/jms/>

http://searchmiddleware.techtarget.com/bestWebLinks/0,289521,sid26_tax285566,00.html

With the preceding concepts in mind let us see how we can write a simple client application that connects to a server and places a message in a queue using the JMS API.

We start by importing the JMS package in our source code with:

```
import javax.jms.*;
```

Next you should get hold of a connection factory object and use it to create a JMS connection. The standard way of getting a factory is through JNDI. The code would look something like:

```
InitialContext ic = new InitialContext();  
QueueConnectionFactory qcf = (QueueConnectionFactory)ic.lookup("QueueConnectionFactory");
```

In the above we got hold of a queue connection factory because we are coding a point-to-point application. Were we coding a publish-subscribe application, we would have gotten a TopicConnectionFactory instead.

The code shown above is not the only way of getting a connection factory although it is the preferred way. Often, however, we may not have access to a JNDI server. In that case you can always instantiate explicitly the connection factory but doing so will make your code less portable since you will need to hardcode a vendor specific class in your application. One way of avoiding this last problem is using JDBC's way of instantiating a driver. You would get the name of the class to be instantiated from, for example, a properties file and then use the Class.newInstance method call to create it.

Whatever method you use to get a connection factory, the next step is to use it to create a connection:

```
QueueConnection qc = qcf.createQueueConnection();
```

Next we start the connection with:

```
qc.start();
```

Starting a connection is only required in order to enable message retrieval. The next step is to create a queue session using the connection as follows:

```
QueueSession qs = qc.createQueueSession( false, Session.AUTO_ACKNOWLEDGE);
```

The first argument of the above method indicates whether this should be a transactional session. If a message is put in a transactional session and the application terminates abnormally the just placed message is removed from the queue. That is, any changes to the queue are rolled back. Messages are not permanently added to the queue unless you explicitly commit them with:

```
qs.commit();
```

If, on the other hand, you would like to rollback the changes in a transactional session you can do it with:

```
qs.rollback();
```

The second argument in the createQueueSession method is the acknowledgement mode. This is a complex topic that cannot be explained in a short article like this. Trying to summarize it briefly, it relates to acknowledgement of messages between consumers of messages and the messaging server. In addition to AUTO_ACKNOWLEDGE it can also be set to CLIENT_ACKNOWLEDGE and DUPS_OK_ACKNOWLEDGE.

Next, we create a queue object where we will place our message(s). This is an object that is a representation of a physical queue in the messaging server. Physical queue creation techniques are vendor dependent so you should consult your vendor's manual for details. The argument passed to the createQueue method is usually the physical queue name.

```
Queue q = qs.createQueue("testqueue");
```

The queue session also allows us to create a queue sender, which is the object we use to put a message into the queue:

```
QueueSender qSender = qs.createSender(q);
```

However, if we were reading a message from the queue we would need to create a QueueReceiver instead:

```
QueueReceiver qReceiver = qs.createReceiver(q);
```

The session object is also used to create a message. The most common message type is a text message:

```
TextMessage msg = qs.createTextMessage("This is a test message");
```

The other message types are BytesMessage, ObjectMessage, MapMessage and StreamMessage.

Finally, we are ready to send the message to the queue, which we do as follows:

```
qSender.send(msg);
```

Once we are done we close all the objects created to save system resources:

```
qSender.close();  
qs.close();  
qc.close();
```

The above is, in broad outline, how we place a message in a queue. The details may vary slightly depending on the vendor.

We would have used the queue receiver if our application were trying to read a message from the queue:

```
Message msg = qReceiver.receive();
```

Assuming the message retrieved is a text message, we can get the text payload from it as follows:

```
String s = ((TextMessage) msg).getText();
```

Note that since many of the commands mentioned above can throw an exception you should be prepared to catch the JMSEException object.

Compiling and testing your application

Start by downloading and installing a JMS implementation. For our example we will be using SwiftMQ because it is a complete and free implementation of the API. You can download it from www.swiftmq.com. If you choose to use another vendor's implementation, you may need to change the code presented here slightly to conform to its conventions.

In the case of SwiftMQ, after unzipping the files, add the product's jar files (found in the jars directory) to your CLASSPATH.

Compile your code the usual way using javac if you are using Sun's JDK. The complete code example for this application can be found in the listing that follows. Note in the listing that we are using the test queue that is created by default at SwiftMQ's install time so there is no need for you to create a physical queue.

With the code compiled, go to the scripts directory and start the SwiftMQ message server with the command:

```
smqr1
```

Then, in a separate command window execute the code as normal using, say, the java command if you are using Sun's JDK. If everything is working correctly you should see the text "This is a test message" displayed in the system output.

At this point, you should have a good understanding of the need and purpose of the Java Messaging Service API. You probably also have a basic understanding of how to use the API. For more information, visit the sites listed in the article sidebars. You may also consider attending SkillBuilders' Developing Applications Using the Java Message Service API course for an in-depth discussion of this valuable API.

Program listing

```
/*-----*/
/* This application puts a message in a queue and then reads it back. A more realistic example */
/* would consist of two applications, one that puts the message and a separate one that processes it. */
/*-----*/

import java.util.*;
import javax.jms.*;
import javax.naming.*;

public class P2P
{
    public static void main( String[] args )
    {
        try
        {
            /*-----*/
            /* Initialize the JNDI context with SwiftMQ's specific parameters. This info would */
            /* normally be placed in a properties files in a real world application. */
            /*-----*/
            Hashtable env = new Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                    "com.swiftmq.jndi.InitialContextFactoryImpl");
            env.put(Context.PROVIDER_URL,"smqp://localhost:4001/timeout=10000");
            Context context = new InitialContext(env);

            System.err.println("Retrieving connection factory from JNDI");
            QueueConnectionFactory qcf = (QueueConnectionFactory)
                context.lookup("QueueConnectionFactory");

            System.err.println("Creating a queue connection");
            QueueConnection qc = qcf.createQueueConnection();

            System.err.println("Starting the connection to enable message retrieval");
            qc.start();

            System.err.println("Creating a queue session");
            QueueSession qs = qc.createQueueSession( false,
                Session.AUTO_ACKNOWLEDGE);

            System.err.println("Creating a queue");
            Queue q = qs.createQueue("testqueue@router1");
        }
    }
}
```

```
System.err.println("Creating a queue sender");
QueueSender qSender = qs.createSender(q);
```

```
System.err.println("Creating a text message");
TextMessage msg = qs.createTextMessage();
msg.setText("This is a test message");
```

```
System.err.println("Sending a message to the queue");
qSender.send(msg);
```

```
System.err.println("Creating a receiver");
QueueReceiver qReceiver = qs.createReceiver(q);
```

```
System.err.println("Receiving a message");
msg = (TextMessage)qReceiver.receive();
```

```
System.err.println("Extracting the message payload");
String s = msg.getText();
System.out.println("Message read from queue: " + s);
```

```
System.err.println("Closing all opened objects");
qReceiver.close();
qSender.close();
qs.close();
qc.close();
context.close();
```

```
    }
    catch( JMSEException je )
    {
        System.out.println("Caught JMSEException: " + je.toString());
    }
    catch (NamingException e)
    {
        System.out.println("Caught JNDI exception: " + e.toString());
    }
}
}
```