Published on **The O'Reilly Network** (http://www.oreillynet.com/)
http://www.oreillynet.com/pub/a/onjava/2001/05/22/ejb_msg.html
See this if you're having trouble printing code examples

# EJB 2 Message-Driven Beans

by Dion Almaer
05/22/2001

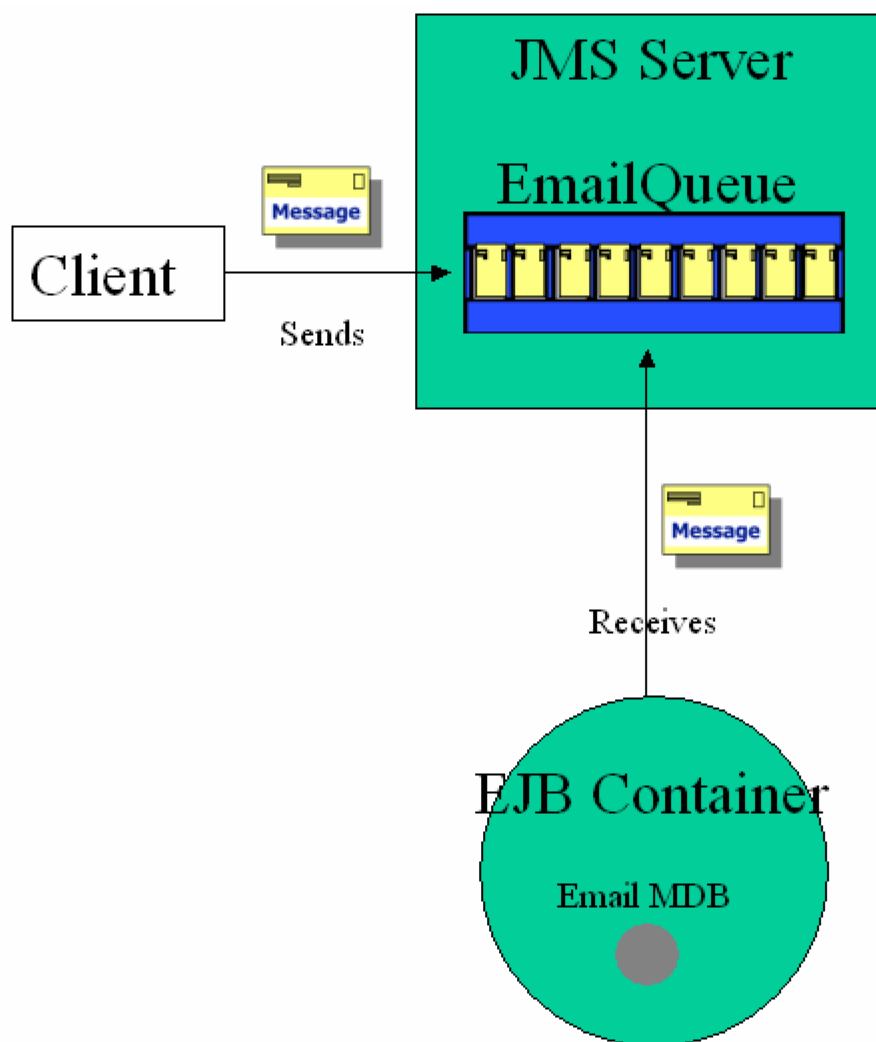
**Working with the new Message Driven Beans and JMS**

This article discusses the new EJB 2.0 Message Driven Beans. I walk through an example of using this new bean, along with the corresponding JMS infrastructure that surrounds it.

All of the code from this article should work with any EJB server that has support for Message Driven Beans (EJB 2.0 container); and you will need a messaging server that supports JMS to hold the queues. This article assumes that you know about Enterprise JavaBeans and their fundamentals.

The system that we will build in this article will be an email queuing system. The parts of the system are

- **JMS Server:** A JMS server will hold the message queue for our email system
- **JMS Queue:** A Queue will hold the JMS messages from the client. The messages on this queue will be MapMessages that will allow us to store name/value pair information about the email to be sent out
- **Email Message Client:** A client will create a JMS message, and put it on the JMS Queue. This message will hold the information for the email message to be sent out
- **Email Message Driven Bean:** A message driven bean will be responsible for taking in the JMS MapMessage and for mailing it out.




### JMS Architecture

Before we can talk about Message Driven Beans, we need to talk about JMS, the Java Messaging Service. There are lots of messaging systems, each with its own API. Messaging systems provide a way to exchange events and data asynchronously. As a programmer, I can send some information to the messaging service and continue processing, without waiting for a response from the system. The JMS API describes a standard way to access any messaging system, just like JDBC allows me to talk to Oracle, Sybase, and SQL Server using the same API. As well as being able to call services asynchronously, we also have the added benefit of having a loose coupling between the code that originates the request, and the code that services the request. Different clients put messages onto a certain destination; and, separately, receivers take messages off the destination and deal with them.

Let's quickly go through some of the basics of the JMS APIs:

- Messaging Domains
- JMS Messages

### Messaging Domains

Message systems have several models of operation. The JMS API provides separate domains that correspond to the different models. A JMS provider may implement one or more of the domains. The two most common domains are Point-to-Point and Publish/Subscribe. Both of the domains have the following concepts.

- **Destination:** the object a client uses to specify the target of messages it sends/receives
- **Producer:** a client that sends a message to a destination
- **Consumer:** a client that receives messages from a destination

## Point-to-Point (PTP)

A point-to-point application has the following characteristics:

- A PTP producer is a sender
- A PTP consumer is a receiver
- A PTP destination is a queue
- A message can only be consumed by one receiver

For example, a call center application may use a PTP domain. A phone call enters the queue, and an operator takes care of that call. The phone call doesn't go to all of the operators, only one.

## Publish/Subscribe (pub/sub)

A pub/sub application has the following characteristics:

- A pub/sub producer is publisher
- A pub/sub consumer is a subscriber
- A pub/sub destination is a topic
- A message may have multiple subscribers

For example, an email newsletter application may use a publish/subscribe model. Everyone who is interested in the newsletter becomes a subscriber, and when a new message is published (say the head of HR sends out new info), that message is sent to all subscribers.

## Our Example

Our email application will be modeled using the PTP domain. When an email is placed on the queue we only want one receiver, else the email could be sent out multiple times.

## JMS Messages

The item that is put on the queue is a JMS Message. This is a generic type that has message headers and contents. There are various subtypes of the JMS Message. For example,

| Type | Description |
|---|---|
| `TextMessage` | This message holds a String. You manipulate the `TextMessage` via `msg.setText("foo")` and `msg.getText()` |
| `ObjectMessage` | This message stores a Serializable object. You manipulate the `ObjectMessage` via `msg.setObject(Object o)` and `msg.getObject()` |
| `MapMessage` | This message holds name/value pairs. You manipulate the `MapMessage` via `msg.setString(key, value)` and `msg.getString(key)`. There are getter and setters for the various base java types (e.g. `getBoolean`, `getInt`, `getObject`, etc) |
| `BytesMessage` | This message is a stream of bytes. This can be used for wrapping existing message formats. |
| `StreamMessage` | This message allows you to work with a stream of primitives |

We will use a `MapMessage` in our example, as it is one way that we can place email header information and the email body, into the message.

## EJB 2.0: Message Driven Bean

Now we have talked about the fundamentals of JMS, we can talk about the new component type that is in the EJB 2.0 specification. (The EJB 2.0 spec if a Public Draft at the time of this article, although the Message Driven Bean part is not rumored to change.)

Let's review the design so far. We have a sender that puts messages on the queue, and a receiver that will read the messages and use that information to send out email. This receiver could be a program that starts up, subscribes to the "EmailQueue", and deals with the messages that come in. Instead of doing this every time we need a receiver, it would be nice to have a component architecture that allows for the concurrent processing of a stream of messages, is transactionally aware, and takes care of the infrastructure code, letting us work on the business logic. This is where Message Driven Beans come in. A Message Driven Bean (MDB) is simply a JMS message consumer. A client cannot access a MDB directly (as you do with session and entity beans). Your only interface to the MDB is by sending a JMS message to the destination of which the MDB is listening. To allow reuse, as with the other EJBs, a lot of information is provided in the EJB deployment descriptor. This means we do not worry about where we go to get the message (whether to a queue or topic), we just write the onMessage (Message msg) code to deal with the message.

We have talked about JMS, and Message Driven Beans; now it's time to build our example.

**Sample Files**

Download the sample files discussed in this article here.

## Email Application Steps

We will walk through the following steps to get the email application working.

1. Setup an "Email Queue" message queue in the JMS server
2. Write an email client, responsible for sending a java message to the email queue
3. Write a message driven bean that consumes these messages, and sends emails using their contents
4. Write the deployment descriptors for the MDB
5. Package the code

The code that we will create is

| Code | Description |
|---|---|
| `com.customware.client.EmailClient` | The EmailClient that will plug a message on the queue |
| `com.customware.ejb.EmailMDB` | The Message Driven Bean that consumes the JMS messages from the client, and then uses the EmailHelper to send out an email |
| `com.customware.util.EmailHelper` | A helper class that has the static method `sendmail(Map mail)` which sends mail using JavaMail |

## Step One: Setup an email message queue

This step will depend on your messaging server (e.g. IBM MQSeries, SonicMQ, and so on). We need to setup a JMS Queue. I will use the name `EmailQueue`, which will be used in both the client (sender) and Message Driven Bean (receiver) deployment descriptor.

## Step Two: Writing the email client (`EmailClient.java`)

Now we will write our client (JMS sender). This client will take in information about the email that we will eventually send out. The `main()` method takes in the arguments from the command line, builds a Hashtable (any Map would work), and calls the `sendmail(Map m)` method. The sendmail method takes this information, builds a `MapMessage` from it, and then sends the message to the `EmailQueue` queue via `sender.send(message)`.

The majority of the work is done in the constructor. It is there that we do all of the JMS work. The constructor does the following:

1. We connect to the JNDI service via the `getInitialContext()` helper method.
2. We lookup a connection factory for a queue [`(QueueConnectionFactory) ctx.lookup(CONNECTION_FACTORY)`]
3. We create a queue connection to our JMS server [`conFactory.createQueueConnection()`]
4. We create a JMS session, which is a context for producing and consuming messages [`connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE)`]
5. We lookup the queue that we will use to send the messages [`(Queue) ctx.lookup(QUEUE_NAME)`]
6. Finally, we create a sender, that can send messages to the queue that we just looked up, using the session that we created earlier [`session.createSender(chatQueue)`]

That's it. Now we have a client that can send messages to our queue. We can compile and run this client via:

```
% java com.customware.client.EmailClient dion@customware.com bob@harris.org "my subject" "my
email body"
```

We have a client. Now we build our consumer.

## Step Three: Write a Message Driven Bean (`EmailMDB.java`)

When writing session, or entity beans, you have to write a remote interface, home interface, and the bean class (and optionally a Primary Key class for an entity bean). With Message Driven Beans we only have the bean class, as there is no "client" that interfaces with the bean.

A Message Driven Bean must implement two interfaces:

| Interface | Description |
|---|---|
| **MessageListener** `javax.jms.MessageListener` | This is the JMS interface that provides the `onMessage(Message msg)` method. When a message is put on the queue, the MDB will have `onMessage` called, and the container will pass in the actual message to be consumed |
| **MessageDrivenBean** `javax.ejb.MessageDrivenBean` | This is the EJB interface that contains the EJB lifecycle methods:<br>• `ejbCreate()`: When the EJB is created the container will call this method<br>• `ejbRemove()`: When the container destroys the EJB it can call the `ejbRemove()` method<br>• `setMessageDrivenContext(MessageDrivenContext ctx)`: The context is passed into the EJB after the object is instantiated, and before `ejbCreate()` is called. The context has information that the container keeps up to date, and allows you to query: transactions (`getUserTransaction()`, `setRollbackOnly()`, `getRollbackOnly()`) `security (getCallerPrincipal(), isCallerInRole())` |

If you look at the code in `EmailMDB.java` you will see that the first few methods implement the `MessageDrivenBean` interface. All we are doing in these methods is printing that they are called. The `setMessageDrivenContext()` method saves away the context to an instance variable, in case we want to query it later. Nine times out of ten, this is what you would do. The last method is the callback `onMessage(Message msg)` from the `MessageListener` interface. This is where we consume the message and deal with it. First, we create a `MapMessage` by casting the message that was given to us. Then we loop through the name-value pairs in the map message, and plug in their values to a standard Hashtable. Note the methods that we use to go through the `MapMessage`:

```
// Get the names out of the Map Message
Enumeration mapnames = mapmessage.getMapNames();

// Get the string out of the map message, giving the key
String val = mapmessage.getString(key);
```

Finally, we call the `EmailHelper.sendmail(map)` method, which takes care of sending the email for us.

That was pretty easy. That is the great thing about MDBs. We don't have lots of ugly JMS code (like we had in the client, or like we would have if we wrote the consumer from scratch); in fact how does the MDB know where to get these messages? For that, we go to the deployment descriptors.

## Step Four: Write the deployment descriptors for the MDB (`ejb-jar.xml`)

At deployment time, we tell the container the information about our MDB. You use standard EJB deployment descriptors to setup MDBs. Therefore, we need a `META-INF` directory to house our descriptor files.

**META-INF\ejb-jar.xml**

In the `ejb-jar` file, we place the fully qualified class name of the MDB, the destination type of the bean, and security information etc. The file follows; notice the class name and the JMS destination type. (The tag `<destination-type>` was changed by the specication authors from `<jms-destination-type>`. You may need to use `<jms-destination-type>` if you are using BEA WebLogic 6.0sp1.)

```
<ejb-jar>
 <enterprise-beans>
  <message-driven>
   <ejb-name>EmailMDB</ejb-name>
   <ejb-class>com.customware.ejb.EmailMDB</ejb-class>
   <transaction-type>Container</transaction-type>
   <message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
   </message-driven-destination>
   <security-identity>
    <run-as-specified-identity>
     <role-name>system</role-name>
    </run-as-specified-identity>
   </security-identity>
  </message-driven>
 </enterprise-beans>
</ejb-jar>
```

Where do we tell it the NAME of the queue? This goes in the vendor specific file. For example, if you were deploying this on BEA WebLogic 6.0 you would have the `weblogic-ejb-jar.xml` file that looks like

**META-INF\weblogic-ejb-jar.xml**

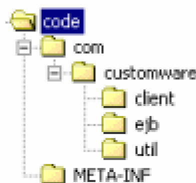
```
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>EmailMDB</ejb-name>
    <message-driven-descriptor>
      <pool>
        <max-beans-in-free-pool>200</max-beans-in-free-pool>
        <initial-beans-in-free-pool>5</initial-beans-in-free-pool>
      </pool>
      <destination-jndi-name>EmailQueue</destination-jndi-name>
    </message-driven-descriptor>
    <jndi-name>jms/EmailMDB</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

The `<ejb-name>` links you to the bean definition in `ejb-jar.xml`. Then we can setup pool information. In this example, we are going to have 5 MDBs instantiated at deployment with the ability to increase to 200. This will allow us to have 200

concurrent messages hitting the queue. The `<destination-jndi-name>` is where we tell the container to look at the EmailQueue. Since we do not have this in code, if the JMS environment changed, we could just change this XML file, and redeploy the bean.

### Step Five: Package up the code

Now we have gotten the code, and the deployment descriptors, we need to package it to deploy on the EJB server. The directory structure should look like




Within the `client`, `ejb`, and `util` directories should be the compiled classes; e.g. `../client/EmailClient.class`, `../ejb/EmailMDB.class`, and `../util/EmailHelper.class`.

Now we archive the code and the deployment descriptors via:

```
../code% jar cvf emailmdb.jar com META-INF
```

Now we have the Email MDB in a jar file that we deploy to an EJB server. To test, after deploying the bean, run the client, and you should see the EJB server sending out an email. You may need to make sure that the JavaMail API `mail.jar` is at the beginning of the `CLASSPATH` for your EJB server.

**Sample Files**

Download the sample files discussed in this article here.

### Conclusion

We have created a Message Driven Bean, showing how simple it is to tie into JMS as a consumer. Message Driven Beans are a nice addition to the EJB component architecture, offering a way for developers to create consumers that are pooled, transactional, and use the container's infrastructure.

*Dion Almaer is a Principal Technologist for The Middleware Company, and Chief Architect of TheServerSide.Com J2EE Community.*

---

Return to ONJava.com.

**oreillynet.com** Copyright © 2003 O'Reilly & Associates, Inc.