

Messaging

“An army marches on its stomach.”

—*Napoleon Bonaparte*

With the introduction of the message-driven bean in the EJB 2.0 specification, Enterprise JavaBean applications can now easily be integrated with messaging systems. Java 2 Platform Enterprise Edition (J2EE)-compliant application servers are required to provide messaging capabilities. Before the message-driven bean, EJB applications could still send Java Message Service (JMS) messages and listen for those messages by including a JMS listener object; however, messages had to be processed in a synchronous manner. Message-driven beans are now the ideal way to expose business logic to messaging applications.

This chapter primarily focuses on problems associated with message-driven bean development. In this chapter, you will find recipes dealing with these topics:

- Sending JMS messages
- Creating a message-driven EJB
- Processing messages first in, first out
- Putting business logic in message-driven beans
- Streaming data with JMS
- Triggering multiple message-driven beans
- Speeding up message delivery
- Using message selectors
- Handling errors in a message-driven bean
- Sending email asynchronously
- Handling rollbacks in a message-driven bean

6.1 Sending a *publish/subscribe* JMS message

◆ **Problem**

You want to send a JMS message to a message topic (known as *publish/subscribe* messaging).

◆ **Background**

Enterprise applications can now use the JMS to communicate to outside applications or other application servers. EJBs can use JMS to decouple communication with these other systems in an asynchronous manner using a *publish/subscribe* pattern. JMS message topics create a one (the sender) to many (the receiver) relationship between messaging partners. In addition, *publish/subscribe* topics can

be used to store messages even when no entity is ready to retrieve them (referred to as *durable subscriptions*).

◆ **Recipe**

The code in listing 6.1 shows a private method, `publish()`, that can be used in any object that wishes to send a JMS message to a *publish/subscribe* topic.

Listing 6.1 The `publish()` method

```
private void publish(String subject, String content) {
    TopicConnection      topicConnection = null;
    TopicSession         topicSession = null;
    TopicPublisher       topicPublisher = null;
    Topic                topic = null;
    TopicConnectionFactory topicFactory = null;
    try{
        Properties props = new Properties();
        props.put (Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        props.put (Context.PROVIDER_URL, url);
        InitialContext context = new InitialContext( props );

        topicFactory =
            ( TopicConnectionFactory )
            context.lookup("TopicFactory");

        topicConnection =
            topicFactory.createTopicConnection();

        topicSession =
            topicConnection.createTopicSession( false,
                Session.AUTO_ACKNOWLEDGE );

        topic = ( Topic ) context.lookup( "ProcessorJMSTopic" );
        topicPublisher = topicSession.createPublisher(topic);

        MapMessage message = topicSession.createMapMessage();
        message.setString("Subject", subject );
        message.setString("Content", content);
        topicPublisher.publish(message);
    }catch(Exception e){
        e.printStackTrace();
    }
}
```

Creates an InitialContext for the Weblogic application server

Looks up the topic factory

Creates a topic connection and session

Finds the topic and builds a publisher

Builds and sends the message

◆ **Discussion**

Publish/subscribe messaging allows you to send a single message to many message listeners. In fact, you can create message Topic destinations as durable, allowing message listeners to retrieve messages that were sent before the listener subscribed to the topic.

To send a message to a JMS topic, you first need to create a Java Naming and Directory Interface (JNDI) context and retrieve the JMS connection factory for topics in the JMS environment. Next, you must create a topic connection in order to establish a topic session. Once you have a session, you can find the actual topic to which you want to send a message, and build a publisher object for transmission of your message. Finally, simply construct your message and send it using the publisher. For more about the JMS API, visit <http://java.sun.com>

◆ **See also**

- 6.2—Sending a point-to-point JMS message
- 6.3—Creating a message-driven Enterprise JavaBean
- 7.8—Securing a message-driven bean

6.2 *Sending a point-to-point JMS message*

◆ **Problem**

You want to send a point-to-point message.

◆ **Background**

Like the publish/subscribe messaging model shown in recipe 6.1, the point-to-point model allows applications to send messages asynchronously to remote message listeners. Point-to-point messaging differs in that it creates a one-to-one relationship between sender and receiver—that is, a single receiver consumes a single message. No message will be duplicated across multiple consumers.

◆ **Recipe**

The code in listing 6.2 shows a private method, `send()`, that can be used in any object that wishes to send a JMS point-to-point message.

Listing 6.2 The send() method

```

private void send(String subject, String content) {
    QueueConnection    queueConnection = null;
    QueueSession       queueSession=null;
    QueueSender        queueSender=null;
    Queue              queue=null;
    QueueConnectionFactory queueFactory = null;

    try{
        Properties props = new Properties();
        props.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        props.put(Context.PROVIDER_URL, url);
        InitialContext context = new InitialContext( props );

        queueFactory =
            (QueueConnectionFactory) context.lookup("QueueFactory");

        queueConnection = queueFactory.createQueueConnection();
        queueSession = queueConnection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);

        queue = (Queue)context.lookup("BookJMSQueue");
        queueSender = queueSession.createSender(queue);

        MapMessage message =
            queueSession.createMapMessage();
        message.setString("Symbol", symbol);
        message.setString("Description", description);
        queueSender.send(message);
    }
    catch(Exception e){
        log("Error Publishing Message");
        e.printStackTrace();
    }
}

```

Creates an InitialContext for the Weblogic application server

Looks up the topic factory

Creates a topic connection and session

Finds the topic and builds a sender

Builds and sends the message

◆ Discussion

To send a point-to-point message, you must send a message to a JMS message queue. To send the message, you first have to create a JNDI context and retrieve the JMS connection factory for a message queue in the JMS environment. Next, you must create a queue connection in order to establish a queue session. Once you have a session, you can find the actual queue to which you want to send a message, and build a sender object for transmission of your message. Finally, you simply construct your message and send it using the sender.

Message queues guarantee that messages are consumed by only one receiver and are never duplicated across multiple listeners (unlike a JMS topic). Message queues are ideal for messages that should be processed concurrently but only once. Many receivers can be pulling messages from a queue for processing at the same time, but each message will be sent to only one consumer.

◆ **See also**

- 6.1—Sending a publish/subscribe JMS message
- 6.3—Creating a message-driven Enterprise JavaBean
- 7.8—Securing a message-driven bean

6.3 **Creating a message-driven Enterprise JavaBean**

◆ **Problem**

You want to create a message-driven bean to contain business logic that will be triggered by a JMS message.

◆ **Background**

Message-driven beans (added to the EJB 2.0 specification) are assigned to receive messages from a particular JMS message destination. These EJBs are ideal for executing business logic asynchronously and for exposing EJB applications to enterprise messaging systems. Message-driven beans use the same transaction models (see chapter 5) and declarative security (see chapter 7) as do session and entity beans. Another advantage of message-driven beans is that they can be used to process messages concurrently. EJB containers can create a pool of identical message-driven beans that are able to process messages at the same time, generating a great deal of processing power.

◆ **Recipe**

This recipe illustrates how to build a simple message-driven bean and create its XML descriptor. The class in listing 6.3 defines a message-driven bean. It implements the required `MessageDrivenBean` interface and the necessary `MessageListener` interface that allows the bean to receive JMS messages.

Listing 6.3 SampleMDB.java

```

public class SampleMDB implements MessageDrivenBean, MessageListener
{
    private MessageDrivenContext ctx;
    public void ejbRemove() { }
    public void ejbPassivate() { }
    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        this.ctx = ctx;
    }
    public void ejbCreate () throws CreateException { }
    public void ejbActivate() { }
    public void onMessage( Message msg ) {
        MapMessage map = ( MapMessage ) msg;
        try {
            processMessage( map );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
    private void processMessage( MapMessage map ) throws Exception
    {
        //implementation not shown
    }
}

```

← **Implements the MessageDrivenBean and MessageListener interfaces**

← **Handles incoming messages**

Listing 6.4 contains the partial deployment XML file for the bean; notice how it indicates the source type of messages for the bean (either point-to-point or publish/subscribe).

Listing 6.4 Deployment descriptor

```

<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>SampleMDB</ejb-name>
      <ejb-class>SampleMDB</ejb-class>
      <transaction-type>Container</transaction-type>
    
```

```

        <message-driven-destination>
            <destination-type>javax.jms.Topic</destination-type>
        </message-driven-destination>
    </message-driven>

</enterprise-beans>

<assembly-descriptor>
</assembly-descriptor>

</ejb-jar>

```

Describes the messaging type for this bean

Finally, you must perform the vendor-specific steps to assign the bean to an actual JMS message destination. The deployment XML describes only the type of messaging used by the message-driven bean, not the actual name of a topic or queue. Consult your application server documentation for more information. For example, the following XML could be used for the Weblogic application server:

```

<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
        <ejb-name>SampleMDB</ejb-name>
        <message-driven-descriptor>
            <destination-jndi-name>BookJMSTopic</destination-jndi-name>
        </message-driven-descriptor>
        <jndi-name>ejb/SampleMDB</jndi-name>
    </weblogic-enterprise-bean>
</weblogic-ejb-jar>

```

◆ Discussion

As with all other types of EJBs, security and transaction control is implemented in the usual way. In some cases, transactions and security do have special considerations that you must take into account when dealing with message-driven beans. For example, you need a good way to prevent unwanted clients from sending messages to your message-driven EJBs and triggering business logic, and you also need to know how to handle rollbacks in the `onMessage()` method. In addition, you should keep in mind that message-driven beans are stateless, and you should therefore not attempt to keep any state information stored at a class level in-between `onMessage()` invocations.

The `MessageDrivenBean` interface must be implemented in order to provide the bean with the appropriate bean methods, such as `ejbRemove()` and `ejbCreate()`. The `Context` object set in the bean is an instance of the `MessageDrivenContext`, which provides many of the methods found in the session and entity bean context classes. However, due to the nature of the message-driven bean, many of the

context methods will throw an exception if used. Since a message-driven bean has no real EJB client (only the container that delivers the message), the `getCallerPrincipal()` and `isCallerInRole()` methods throw a runtime exception. In addition, message-driven beans have no home interfaces (and therefore have no home objects), so `getEJBHome()` and `getEJBLocalHome()` also throw runtime exceptions if used. Finally, since no EJB clients exist for a message-driven bean, the transaction context for the start of the `onMessage()` method is started by the container in the case of container-managed transactions, or by the bean itself in the case of bean-managed transactions.

◆ **See also**

- 6.1—Sending a publish/subscribe JMS message
- 6.2—Sending a point-to-point JMS message
- 7.8—Securing a message-driven bean

6.4 Processing messages in a FIFO manner from a message queue

◆ **Problem**

You want to ensure that a message in a queue is processed only after any previous message has finished processing.

◆ **Background**

While some business logic operated by a message-driven bean can process messages in any order, other business functions might need messages supplied in a specific order. For instance, you might want to process incoming JMS messages according to the order in which they were received to preserve a specific data-driven workflow. Each message can be a step in a workflow, and the next step cannot begin without the previous one completing. Refer to recipe 6.2 for a discussion of using message queues.

◆ **Recipe**

The client shown in listing 6.5 publishes messages onto a message queue for a message-driven bean to pick up.

Listing 6.5 Client.java

```
public class Client
{
    private QueueConnection    queueConnection = null;
    private QueueSession      queueSession = null;
    private QueueSender       queueSender = null;
    private Queue             queue = null;
    private QueueConnectionFactory queueFactory = null;
    private String            url = getURL();

    public Client() throws JMSEException, NamingException {
        Context context = getInitialContext();

        queueFactory = (QueueConnectionFactory)
            context.lookup("BookJMSFactory");
        queueConnection = queueFactory.createQueueConnection();
        queueSession = queueConnection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);
        queue = (Queue) context.lookup("BookJMSQueue");
        queueSender = queueSession.createSender(queue);
    }

    public void send() throws JMSEException {
        MapMessage message = null;

        for(int i=0;i<10;i++){
            message = queueSession.createMapMessage();
            message.setInt("Index",i);
            queueSender.send(message);
        }
    }

    public void close() throws JMSEException {
        queueConnection.close();
    }

    public static void main(String[] args) {
        Client sender = null;

        try{
            sender = new Client();
            sender.send();
            sender.close();
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Notice that the client sends a counter value in the messages. The message-driven bean will use that value to show that the messages are received and processed one at a time. The message-driven bean shown in listing 6.6 picks up messages from the message queue and prints out the counter value that each message contains.

Listing 6.6 MessageBean.java

```
public class MessageBean implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx;

    public void onMessage( Message msg ) {
        MapMessage map = (MapMessage) msg;

        try {
            int index = map.getInt("Index");
            System.out.println( "Processing message: " + index );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    //other bean methods not shown
}
```

Since we made use of a message queue, we are guaranteed that messages will be removed from the queue in the order in which they were placed. To ensure that one message is completely processed before the next message begins, you should deploy only a single message-driven bean to remove messages from the queue.

Listing 6.7 contains the deployment XML for the bean; notice how it indicates the source type of messages for the bean.

Listing 6.7 Deployment descriptor

```
<ejb-jar>
  <enterprise-beans>

    <message-driven>
      <ejb-name>fifoMDB</ejb-name>
      <ejb-class>fifo.MessageBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
        <destination-type>javax.jms.Queue</destination-type>
      </message-driven-destination>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

```

</enterprise-beans>
<assembly-descriptor>
</assembly-descriptor>
</ejb-jar>

```

To ensure that the second message is not consumed before the first message processing has completed, you must have only one bean listening to the queue. This is set up in the vendor-specific deployment file. For example, you can use XML like that shown in listing 6.8 for the Weblogic application server.

Listing 6.8 Weblogic deployment descriptor

```

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>fifoMDB</ejb-name>
    <message-driven-descriptor>
      <pool>
        <max-beans-in-free-pool>1</max-beans-in-free-pool>
        <initial-beans-in-free-pool>1</initial-beans-in-free-pool>
      </pool>
      <destination-jndi-name>BookJMSQueue</destination-jndi-name>
    </message-driven-descriptor>
    <jndi-name>fifo.MBD</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>

```

**Creates the
message-
driven bean
pool of size 1**

◆ **Discussion**

Message queues guarantee that only one consumer will process each single message. By limiting the number of consumers assigned to a queue to a single message-driven bean, you ensure that all the messages will be processed in the order in which they were received. In addition, using one consumer guarantees that each message will completely process before the next one begins processing. Otherwise, you can create a pool of message-driven beans (by increasing the pool size in a vendor-specific manner) to pull messages faster from the queue. Messages will still only be delivered to a single message-driven bean instance, but using many message-driven bean instances with the same queue results in faster message processing.

◆ **See also**

6.2—Sending a point-to-point JMS message

6.5 Insulating message-driven beans from business logic changes

◆ **Problem**

You want to prevent changing your message-driven EJB classes when the business logic they invoke changes.

◆ **Background**

Message-driven EJBs are ideal for executing business logic via JMS messages. However, when developing an enterprise application in a changing environment (or a shorter development cycle), you might find that you spend too much time upgrading the business logic contained in your message-driven beans. It would be ideal to encapsulate your business logic and insulate your message-driven beans from unnecessary changes.

◆ **Recipe**

To shield your message-driven beans from business logic changes, encapsulate the business logic with an intermediary object. The message-driven EJB shown in listing 6.9 uses an instance of the `BusinessLogicBean` class.

Listing 6.9 `MessageBean.java`

```
public class MessageBean implements MessageDrivenBean, MessageListener
{
    private MessageDrivenContext ctx;

    public void onMessage(Message msg) {
        MapMessage map= (MapMessage)msg;

        try {
            String symbol = map.getString("Symbol");
            String description = map.getString("Description");

            BusinessLogicBean bean =
                new BusinessLogicBean( symbol, description );
            bean.executeBusinessLogic();
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Invokes the
encapsulated
business logic

◆ **Discussion**

The `BusinessLogicBean` class has a single purpose: to encapsulate business logic. This class is a simple object that allows the message-driven bean to execute business logic by passing in parameters and invoking a method. Using a class like this allows the EJB to shield itself from changes to the business logic. In addition, it is good practice to build business logic into reusable classes. An alternative to using a simple object is to invoke a session EJB that already encapsulates some business logic. By encapsulating all business logic with session beans, you ensure that the logic is available to both message-driven beans and any other EJB clients.

◆ **See also**

6.3—Creating a message-driven Enterprise JavaBean

6.6 Streaming data to a message-driven EJB

◆ **Problem**

You want to send a stream of data to a message-driven EJB.

◆ **Background**

Message-driven beans can receive all types of JMS messages. Because of that capability, you can use the most appropriate JMS message type for the data you want to send. For instance, when you want to send a large amount of binary data (like an image), you should stream the data to conserve bandwidth and memory. Refer to recipe 6.1 for more information on using message topics.

◆ **Recipe**

This solution demonstrates a client and a message-driven bean using streamed data. Listing 6.10 shows a client that streams a message containing data from a binary file to a message-driven EJB. It uses a message topic as a message destination.

Listing 6.10 Client.java

```
public class Client
{
    private TopicConnection    topicConnection = null;
    private TopicSession      topicSession = null;
    private TopicPublisher     topicPublisher = null;
    private Topic              topic = null;
    private TopicConnectionFactory topicFactory = null;
```

```
private String          url= "http://myjndihost";

public Client( String factoryJNDI, String topicJNDI )
    throws JMSEException, NamingException {

    // Get the initial context, implementation not shown
    Context context = getInitialContext();

    // Get the connection factory
    topicFactory = ( TopicConnectionFactory )
        context.lookup( factoryJNDI );

    // Create the connection
    topicConnection = topicFactory.createTopicConnection();

    // Create the session
    topicSession=topicConnection.createTopicSession( false,
        Session.AUTO_ACKNOWLEDGE );

    // Look up the destination
    topic = ( Topic ) context.lookup( topicJNDI );

    // Create a publisher
    topicPublisher = topicSession.createPublisher( topic );
}

public void sendToMDB( String filename ) throws JMSEException
{
    byte[]          bytes = new byte[1024];
    FileInputStream istream = null;
    int             bytesRead = 0;

    try{
        BytesMessage message = topicSession.createBytesMessage();
        Istream = new FileInputStream(filename);
        while( (bytesRead = istream.read( bytes,0,bytes.length ) ) > 0 )
        {
            message.writeBytes( bytes, 0, bytesRead);
        }
        istream.close();
        topicPublisher.publish(message);
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

public void close() throws JMSEException {
    topicSession.close();
    topicConnection.close();
}

public static void main(String[] args) {
    Client publisher = null;
    String filename = null;

    try{
```

**Creates the
BytesMessage
instance**

**Writes the data
to the message**

```

        publisher = new Client( "BookJMSFactory", "BookJMSTopic" );
        System.out.println("Publishing message:");

        if( args.length > 0){
            filename = args[0];
            publisher.sendToMDB(filename);
            publisher.close();
        }
    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
}

```

Listing 6.11 shows the sample message-driven bean that receives data from a streamed message. This bean simply prints out the data it receives.

Listing 6.11 MessageBean.java

```

public class MessageBean implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx;
    public void ejbRemove() { }
    public void ejbPassivate() { }
    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        this.ctx = ctx;
    }
    public void ejbCreate () throws CreateException { }
    public void ejbActivate() { }
    public void onMessage( Message msg )
    {
        BytesMessage message = ( BytesMessage ) msg;
        int bytesRead = 0;
        byte[] bytes = new byte[1024];
        try {
            while( (bytesRead = message.readBytes(bytes, 1024) ) > 0 ){
                System.out.println( new String( bytes, 0 , bytesRead ) );
            }
        }catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

**Reads the data
off the message**

◆ **Discussion**

Streaming large amounts of data helps you to avoid building a single large message. In addition, message streams are ideal for sending binary file data. By using message streams, you can more easily build messaging systems that can restart message transmission from the point of failure, rather than retransmit data. The client uses the `BytesMessage` message class. This message type is used specifically for sending large amounts of data to a message listener. The message-driven bean uses its `onMessage()` method to receive the message, as it would any other message type. The message-driven bean in this recipe only printed out the data it received from the streamed message, but it could instead store it in a database or create a new file containing the data.

◆ **See also**

- 6.1—Sending a publish/subscribe JMS message
- 6.3—Creating a message-driven Enterprise JavaBean

6.7 **Triggering two or more message-driven beans with a single JMS message**

◆ **Problem**

You want to start two or more business methods concurrently with a single JMS message.

◆ **Background**

Message-driven beans give other parts of an enterprise application the ability to execute business logic asynchronously. However, sending multiple JMS messages to execute multiple pieces of business logic can be time-consuming and redundant. To improve the efficiency of code, you should send a single message that triggers multiple message-driven beans.

◆ **Recipe**

To execute two pieces of business logic with a single message, you need only have two different message-driven beans listen for the same message. To do this, you must use a JMS message topic. Topics create a one-to-many relationship between sender and receiver(s). For this example, we will use two simple message-driven

beans (listings 6.12 and 6.13). The `onMessage()` method simply prints out a statement indicating it has received a message.

Listing 6.12 MessageBean.java

```
public class MessageBean implements MessageDrivenBean, MessageListener {
    public void onMessage(Message msg) {
        MapMessage map = ( MapMessage ) msg;
        try {
            String symbol = map.getString("Symbol");
            String description = map.getString("Description");
            System.out.println("MDB 1 received Symbol : " + symbol
                + " " + description );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
    //other bean methods not shown
}
```

Listing 6.13 MessageBean2.java

```
public class MessageBean2 implements MessageDrivenBean, MessageListener {
    public void onMessage(Message msg) {
        MapMessage map=(MapMessage)msg;
        try {
            String symbol = map.getString("Symbol");
            String description = map.getString("Description");
            System.out.println("MDB 2 received Symbol : " + symbol
                + " " + description );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Listing 6.14 contains the XML descriptor for these beans. As you can see, the descriptor indicates the JMS destination type.

Listing 6.14 Deployment descriptor

```
<enterprise-beans>
  <message-driven>
    <ejb-name>MDB</ejb-name>
    <ejb-class>multiSubscriber.MessageBean</ejb-class>
    <transaction-type>Container</transaction-type>
    <message-driven-destination>
      <destination-type>javax.jms.Topic</destination-type>
    </message-driven-destination>
  </message-driven>
  <message-driven>
    <ejb-name>MDB2</ejb-name>
    <ejb-class>multiSubscriber.MessageBean2</ejb-class>
    <transaction-type>Container</transaction-type>
    <message-driven-destination>
      <destination-type>javax.jms.Topic</destination-type>
    </message-driven-destination>
  </message-driven>
</enterprise-beans>
```

Assigns the message-driven bean to a JMS topic

The actual topic used by the message-driven beans is specified in a vendor-specific manner. For example, listing 6.15 shows the XML used by the Weblogic application server to specify the JMS topic for each bean.

Listing 6.15 Weblogic deployment descriptor

```
<weblogic-ear-jar>
  <weblogic-enterprise-bean>
    <ejb-name>MDB</ejb-name>
    <message-driven-descriptor>
      <destination-jndi-name>BookJMSTopic</destination-jndi-name>
    </message-driven-descriptor>
    <jndi-name>multiSubscriber.MDB</jndi-name>
  </weblogic-enterprise-bean>
  <weblogic-enterprise-bean>
    <ejb-name>MDB2</ejb-name>
    <message-driven-descriptor>
      <destination-jndi-name>BookJMSTopic</destination-jndi-name> #1
    </message-driven-descriptor>
    <jndi-name>multiSubscriber2.MDB</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ear-jar>
```

Assigns the message-driven bean to the BookJMSTopic topic

◆ **Discussion**

Recipe 6.1 provides more information on JMS topics. Since they allow multiple message-driven beans (even message-driven beans of different Java classes) to receive the same incoming message, you can use them to create concurrent processing for sections of business logic. Sending a single message, you can trigger two completely unrelated business functions to start processing at the same time.

In this recipe, each message-driven bean simply prints out a statement indicating it has received a message. However, in a practical application the two message-driven beans should each contain an important business function. To ensure that both message-driven beans receive the same message, they both need to subscribe to a JMS topic. For both beans to be triggered by a single message, both EJBs need to use the same topic.

◆ **See also**

- 6.1—Sending a publish/subscribe JMS message
- 6.3—Creating a message-driven Enterprise JavaBean
- 6.9—Filtering messages for a message-driven EJB

6.8 **Speeding up message delivery to a message-driven bean**

◆ **Problem**

You want to reduce the time it takes for a message to start processing in a message-driven bean.

◆ **Background**

In most enterprise situations, you want your asynchronous business functions to complete as quickly as possible. Since a message-driven bean processes a single message at a time, the waiting time for a single message increases as the number of messages delivered before it increases. In other words, if a single message takes a long period of time to complete, other messages experience a delay before processing. In critical applications, these messages should be processed as quickly as possible.

◆ **Recipe**

To speed up the consumption of messages, use a pool of message-driven beans. Each EJB is an instance of the single EJB class. With a pool of message-driven beans, you can consume more messages in a shorter time. Listing 6.16 shows a simple message-driven bean used to consume messages.

Listing 6.16 MessageBean.java

```
public class MessageBean implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx;

    public void ejbRemove() {
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        this.ctx = ctx;
    }

    public void ejbCreate () throws CreateException {
    }

    public void onMessage(Message msg) {
        MapMessage map= (MapMessage) msg;

        try {
            String symbol = map.getString("Symbol");
            String description = map.getString("Description");

            System.out.println("MDB received Symbol : " + symbol
                + " " + description );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

To ensure a single message is not duplicated across instances in the message-driven bean pool, the message-driven bean instances should use a message queue as the destination type. Listing 6.17 contains the deployment descriptor for the bean.

Listing 6.17 Deployment descriptor

```

<enterprise-beans>
  <message-driven>
    <ejb-name>concurrentMDB</ejb-name>
    <ejb-class>concurrent.MessageBean</ejb-class>
    <transaction-type>Container</transaction-type>
    <message-driven-destination>
      <destination-type>javax.jms.Queue</destination-type>
    </message-driven-destination>
  </message-driven>
</enterprise-beans>

```

Message-driven bean instance pools are created in a vendor-specific manner. Listing 6.18 shows how this is accomplished using the Weblogic application server. Notice the vendor XML creates a pool maximum size of five beans, with an initial size of two beans.

Listing 6.18 Weblogic deployment descriptor

```

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>concurrentMDB</ejb-name>
    <message-driven-descriptor>
      <pool>
        <max-beans-in-free-pool>5</max-beans-in-free-pool>
        <initial-beans-in-free-pool>2</initial-beans-in-free-pool>
      </pool>
      <destination-jndi-name>BookJMSQueue</destination-jndi-name>
    </message-driven-descriptor>
    <jndi-name>concurrent.MBD</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>

```

Sets up the message-driven bean pool

◆ Discussion

Using a bean pool is a quick and dirty way to achieve concurrent processing of messages. The pool, combined with a message queue, provides a way to process many messages at once without duplicating messages across instances. Creating an environment like this allows messages to start processing instead of waiting for previous messages to complete. You should use this type of processing only when

concurrent processing of messages will not cause problems in your business logic or invalid states in your data.

◆ **See also**

- 6.1—Sending a publish/subscribe JMS message
- 6.3—Creating a message-driven Enterprise JavaBean

6.9 Filtering messages for a message-driven EJB

◆ **Problem**

You want your message-driven beans to receive only the messages that they are intended to process.

◆ **Background**

Message-driven beans that subscribe to a topic or receive messages from a queue should be able to handle messages of the wrong type (which should not invoke the message-driven business logic). Beans should just politely discard these messages when they are encountered. This is especially true for message-driven beans that exist in an environment with many different beans that watch a single source for incoming messages. However, it would be more efficient to avoid the execution time used for discarding messages and instead avoid receiving unwanted messages.

◆ **Recipe**

To selectively deliver messages to a message-driven bean, the bean should be deployed with a *message selector*. The bean source needs no changes in order to use the message selector. Listing 6.19 shows a simple message-driven bean that only wants messages that contain an attribute `UserRole` set to `"BuyerRole"`. The bean prints out the role of the incoming message for verification.

Listing 6.19 MessageBean.java

```
public class MessageBean implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx;

    public void onMessage(Message msg) {
        MapMessage map= (MapMessage)msg;

        try {
            String role = map.getString( "UserRole" );
```

```

        System.out.println("Received Message for Role: " + role);
        ProcessTheMessage( message );
    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
}
}
}

```

In the XML descriptor for the bean, you describe the message selector that filters undesired messages for the message-driven bean. Listing 6.20 shows the partial XML descriptor that describes the simple EJB and its message selector.

Listing 6.20 Deployment descriptor

```

<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>MDB</ejb-name>
      <ejb-class>messageSelector.MessageBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-selector>
        <![CDATA[ UserRole = 'BuyerRole' ]]>
      </message-selector>
      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
      </message-driven-destination>
    </message-driven>
  </enterprise-beans>
</ejb-jar>

```

**Specifies a
message
selector**

Here is a simple publish method that appropriately creates messages for the message-driven bean message selector:

```

public void publish(String role) throws JMSEException {
    MapMessage message = topicSession.createMapMessage();
    message.setString("UserRole", role);
    message.setStringProperty("UserRole", role);

    System.out.println( "Publishing message to Role:" + role );
    topicPublisher.publish(message);
}

```

◆ Discussion

When sending particular messages, we must assign a value to the property `User-Role`. The message selector will pick out the messages that meet its criteria and deliver them to the message-driven bean. Message selectors operate using the property values that are set in JMS messages. Any property that is set in the message can be examined by a message selector for filtering purposes.

Message selection strings can be any length and any combination of message property comparisons. The following is an example of a more complex message selector:

```
"DollarAmount < 100.00 OR (ShareCount < 100 AND ( CreditAmount  
- DollarAmount > 0 ) ) AND Role in ('Buyer', 'ADMIN' )"
```

You can make other familiar comparisons using the following operators as well: `=`, `BETWEEN`, and `LIKE` (using a `%` as a wildcard). As mentioned in the recipe, message selectors operate upon messages by examining the properties set in the message using its `setStringProperty()` method. If a property is not present in a message, the selector considers that a nonmatching message. To specify the message selector in the deployment XML, you must use the `CDATA` tag to avoid XML parsing errors due to the use of special characters like `<` or `>`.

◆ See also

6.3—Creating a message-driven Enterprise JavaBean

6.10 Encapsulating error-handling code in a message-driven EJB

◆ Problem

Rather than handle errors in a message-driven bean, you want your beans to off-load errors to an error-handling system.

◆ Background

Handling errors across all your message-driven beans should be consistent and exact. By keeping the error-handling code in your message-driven beans, you open your beans to tedious changes if your error-handling strategy changes. If you must change the error-handling code in one bean, you might have to change it in all your message-driven beans. Passing exceptions to an error-handling object or

session bean allows you to avoid rollbacks and gracefully handle errors in a consistent manner.

◆ **Recipe**

Instead of acting upon any errors, the message-driven bean catches any exceptions and forwards them on to an error-handling session bean. The message-driven bean should be implemented as usual; the only new addition is the error-handling mechanism (see listing 6.21).

Listing 6.21 MessageBean.java

```
public class MessageBean implements MessageDrivenBean, MessageListener {
    private MessageDrivenContext ctx;

    public void onMessage(Message msg) {
        MapMessage map = (MapMessage) msg;
        String symbol = null;
        String description = null;
        ErrorHome errorHome = null;

        try {
            symbol = map.getString("Symbol");
            description = map.getString("Description");

            System.out.println("Received Symbol : " + symbol);
            System.out.println("Received Description : " + description);

            processEquityMessage( symbol, description );
        }
        catch(Exception e){
            e.printStackTrace();
            System.out.println("Error Creating Equity with Symbol:"+symbol);
            System.out.println("Consuming error and "
                + "passing on to error Handler");

            try{
                handleError( e, msg );
            }
            catch(Exception errorExc){}
        }
    }

    private void handleError( Exception e, Message msg )
    {
        ErrorHandler handler = lookupErrorEJB();
        handler.handleMessageDrivenError( e.getMessage(), msg );
    }
}
```

Looks up and
uses the error-
handling
session EJB

The `handleError()` method looks up a session EJB that handles specific errors. For example, the following remote interface could expose error-handling functionality to an entire EJB application:

```
public interface ErrorHandler extends javax.ejb.EJBObject
{
    public void handleMessageDrivenError( String message, Message msg );
    public void handleSessionError( Object errorMessage );
    public void handleEntityError( Object errorMessage );
}
```

◆ **Discussion**

The message-driven EJB shown in the recipe processes messages containing equity information. The actual message-processing logic is not shown, so instead let's examine the `handleError()` method invoked only when an exception occurs during message processing. The session EJB interface shown in the recipe declares methods for handling different types of errors. For example, the session bean has a specific way it can handle session bean errors, entity bean errors, and message-driven bean errors. Using an error-handling system like this does not have to take the place of a normal transactional system. Instead, it acts as a way to store information on errors occurring in your application—acting as a logger of errors, and possibly offloading them to a management system.

◆ **See also**

6.12—Handling rollbacks in a message-driven bean

6.11 *Sending an email message asynchronously*

◆ **Problem**

You want to provide your EJBs with the ability to send email in an asynchronous manner.

◆ **Background**

The ability to send email is an important part of many enterprise applications. Email can be used to send notifications, alerts, and general information, such as price quotes or contract information. When sending an email from an EJB, you should be able to continue processing without waiting for an email to be sent. Sending email using the Java mail package is a simple process.

◆ **Recipe**

Combining email-sending code with a message-driven bean provides the asynchronous behavior that is ideal for enterprise applications. Listing 6.22 contains a message-driven bean that sends email using property values passed to it via a JMS message.

Listing 6.22 EmailBean.java

```
import javax.jms.*;

public class EmailBean implements MessageDrivenBean, MessageListener {

    private MessageDrivenContext ctx;

    public void onMessage( Message msg ) {
        MapMessage map = (MapMessage) msg;

        try {
            sendEmail( map.getProperty( "Recipient" ),
                      map.getProperty("message" ) );
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    private void sendEmail(String recipient, String text)
        throws Exception {

        Session mailSession = null;
        javax.mail.Message msg = null;

        try{
            System.out.println( "Sending Email to: " + rcpt );

            mailSession = (Session) ctx.lookup("BookMailSession");

            msg = new MimeMessage(mailSession);
            msg.setFrom();
            msg.setRecipients(Message.RecipientType.TO,
                              InternetAddress.parse( recipient , false));
            msg.setSubject("Important Message");
            msg.setText(text);
            Transport.send(msg);
            System.out.println("Sent Email to: "+rcpt);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Retrieves the email address and text from the JMS message

Sends the email message

◆ **Discussion**

When using a message-driven bean to send an email message, you need to be sure to send a JMS message with all the values that you need for the email. For instance, the solution in the recipe only retrieved the email address and text from the JMS message and populated the subject of the email with a hardcoded value.

Another improvement you can make to your message-driven email beans is to only send JMS messages that contain the email recipient address and the type of email to send. For instance, a message-driven bean can be initialized with standard email message texts to use for your various email needs in your enterprise application (purchase confirmation, error, contract status, etc.). This would include the subject and message. All your application needs to do is supply a valid email address and the type of email to send. This way, you won't have to transmit the body of an email message to the EJB. In addition, you could pass parameters to the EJB for formatting an already loaded email body.

◆ **See also**

4.5—Sending an email from an EJB

6.12—Handling rollbacks in a message-driven bean

6.12 Handling rollbacks in a message-driven bean

◆ **Problem**

When a transaction in a message-driven bean rolls back, the application server can be configured to resend the JMS message that started the transaction. If the error that caused the rollback keeps occurring, you could potentially cause an endless loop.

Background

Rollbacks in message-driven beans occur in the same way that they can happen in other beans—an error occurs in executing logic. However, in the case of a message-driven bean using a durable subscription, the application server will most likely attempt to redeliver the message that caused the rollback in the bean. If the error is not corrected, the rollback will continue on, consuming more processing time and memory. You need your message-driven beans to be insulated from rollback loops and able to handle error-causing messages without a rollback every time.

◆ **Recipe**

To handle rollbacks in a message-driven bean, keep track of the number of times a particular message has been delivered to the bean. Once a certain retry limit is reached, you want to discard the message from that point on. Listing 6.23 shows the `onMessage()` method from a message-driven bean that tracks and checks message redelivery attempts.

Listing 6.23 The `onMessage()` method

```
private HashMap previousMessages;
private int count = 0;

public void onMessage( Message incomingMsg )
{
    // get the unique message Id
    String msgId = incomingMsg.getJMSMessageID();
    if ( previousMessages.containsKey(msgId) ) ← Checks for previous
        count = ( (Integer) msgMap.get(msgId) ).intValue();      attempts
    else
        count = 0;

    // if msg has been retried couple of times, discard it.
    // and remove the stored id.
    if ( count < _MAX_REDLIVERY_CONST_ ) ← Checks the number of attempts
    {
        logMessage(incomingMsg);
        previousMessages.remove( msgId );
        return;
    }

    //perform business logic for message
    boolean error = doBusinessFunction();

    if ( error )
    {
        mdbContext.setRollBackOnly(); ← Checks for
        previousMessages.put( msgId, new Integer(++count) );      necessary
    }
    else
    {
        if( previousMessages.containsKey( msgId ) )
            previousMessages.remove( msgId );
    }
}
}
```

◆ **Discussion**

Some application servers and some JMS vendors allow you to specify the redelivery count of a rolled-back message delivery to a message-driven bean. However, to ensure your message-driven EJBs are the most secure and portable, you can implement a simple message tracker like the one shown in the recipe. In this code, the EJB maintains a Map of message IDs and the number of times they have been delivered. If the delivered count for a particular message reaches a predefined constant value, the bean simply logs the message and returns. By returning successfully, the EJB ensures that the EJB container commits the transaction and the message will not be delivered again.

If the message makes it past the count check, the bean will attempt to perform its business function. After attempting the business logic, the EJB will check to see if it is necessary to mark the current transaction for rollback. If so, the EJB uses its `MessageDrivenContext` instance to mark the transaction and returns. The container will roll back the transaction and will attempt to redeliver the message. The `previousMessages` Hashtable will store only those message IDs that caused errors. If the message succeeds, no ID is stored (and any previously stored ID is removed).

◆ **See also**

Chapter 5, “Transactions”

