Exercises for Chapter 13

Exercise 13.1: JMS as a Resource

This exercise demonstrates JMS as a resource to which the TravelAgent EJB publishes a message. In this exercise we modify the TravelAgentEJB to use the publish-subscribe model of asynchronous messaging to publish a simple text message to a JMS Topic. For a detailed explanation of the Point-Point and Publish-Subscribe messaging domains refer the to the EJB Book.

Building the application for Exercise 13.1

Download the *ex13-1.jar* file from the download site. Downloads may be available for the complete bundle of Exercises.

Copy the downloaded file and extract the file using *WinZip* or *jar* with `xvf` flags to *$EXAMPLES_HOME*. Example: *c:\> EJBBook* directory.

Change directory to *$EXAMPLES_HOME\src\ex13_1*.

Open a command prompt and type `Ant` or `Ant -buildfile build.xml`

This should create *$EXAMPLES_HOME\build\ex13_1* directory and compile the required files.

$EXAMPLES_HOME\pre-built\ex13_1 directory contains pre-packaged *ear* file and the client *jar* file for this exercise.

We also need to create the required JMS resource information that will be detailed in the following sections.

We will use the pre-built application archive for this example. We will walk through the settings for the TravelAgentBean EJB as well as the JMS resource mappings.

Database schema for Ex13_1

There are no changes in the database schema for chapter 13 compared to chapter 12. We will be using the application to deploy and create the corresponding tables on deployment.

Creating required JMS resources

This exercise demonstrates the Publish and Subscribe Java messaging domain model. We modify the TravelAgentBean EJB to publish a message to a topic upon reservation. We create a subscriber using a Java Client application. This Java Client shows interest in the same topic by subscribing to the same topic on the JMS Server. Upon arrival of the message, the message is delivered to all the subscribers listening for the message.

J2EE™ SDK comes with a built-in JMS server. We will create the following resources in the J2EE Server:

- A JMS Connection Factory called titan-TopicFactory

- A JMS Topic called titan-TicketTopic.

These resources can be created from the command-line using the J2EEAdmin utility.

JMS Factory Creation

Open a command prompt and type the following to create a JMS TopicFactory:

```
$prompt\> j2eeadmin –addJmsFactory titan-TopicFactory topic
```

This will create a JMS Connection factory to handle topics with a JNDI name of **titan-TopicFactory**. You can also run the Deploytool and open **Tools→Server Configuration**. Navigate to the JMS ConnectionFactory node and create a topic factory called titan-TopicFactory and type `topic`.

JMS Destination Creation

Open a command prompt and type the following to create a JMS Destination of resource type Topic.

```
$prompt\> j2eeadmin –addJmsDestination titan-TicketTopic topic
```

This will create the required JMS Destination of type Topic with the JNDI name of **titan-TicketTopic**. You can run the Deploytool and open **Tools→Server Configuration** and navigate to JMS Destinations and validate the creation of the required JMS Destination.

Building the Application Archive for Ex13_1

1. Run the Deploytool by typing in at the command prompt `c:\>deploytool`.
2. Create a new application and call it *Titan13_1App.ear*.

A pre-built application is available in *$EXAMPLES_HOME/pre-built* directory.

This exercise will use all the EJBs created in chapter 12 with a variation to the TravelAgentBean EJB.

You can use the pre-built application archive.

3. Open the Titan13_1EJB's *jar* node and make sure all the resource references for the set.

AddressBean, PhoneBean, CreditCardBean, ReservationBean, CruiseBean use TitanUID JDBC resource to get the sequence ID.

- If these EJBs do not have a Resource Reference of type javax.sql.Datasource called **jdbc/TitanUID** and then the **userID** and **password** are not set, the following exception will appear when you run; it can be somewhat confusing.

```
javax.ejb.EJBException: nested exception is: java.sql.SQLException:
java.lang.IllegalStateException: Local transaction already has 1
non-XA Resource: cannot add more resources
```

4. Verify the resource reference for ProcessPaymentBean EJB and TravelAgentBean EJB. These EJBs use the regular **jdbc/Titan** Datasource. Enter the **userID, password** and **JNDI name** for this resource.

As we mentioned earlier TravelAgentBean EJB is modified to publish a message to the topic. Create a Resource Reference to the Topic Connection Factory as shown below.

5. Click on the **Resource Refs** tab and then click on the **Add** button.
6. This EJB references the **titan-TopicFactory** to create a connection. We need to add the reference information here. Add the following information.

- Coded Name: **jms/TopicFactory**
- Type: **javax.jms.TopicConnectionFactory**
- Authentication: **Container**
- Sharable: Select the checkbox.

7. Enter `titan-TopicFactory` for the **JNDI name** and enter the **UserID** and **Password** fields. You can use **scott** and **tiger** or any other user that has access to the J2EE server.
8. Click on the **Resource Environment References** tab and click the **Add** button to add the following information.

    - Coded Entry: **jms/TicketTopic**
    - Type: **javax.jms.Topic**
    - JNDI-Name: **titan-TicketTopic**

This example uses both JSP clients and a Java Client. The web Client pre-populates the database with the required data and creates the reservations. The Java Client subscribes to the topic and listens for the messages on the topic.

Select **File→New→Web Component** from the menu to create a new web archive. Select all JSPs from *$EXAMPLES_HOME/build/ex13_1/jsp* directory. Go through the wizard setting up the EJB reference for the CustomerEJB and ProcessPaymentEJB with their references for Remote home and EJBObject interfaces. If using the pre-built archive all the information is already set.

Similarly select **File→New Application Client** from the menu to create a new application client. This client has references to the `javax.jms.TopicConnectionFactory` and `javax.jms.Topic` similar to TravelAgentBean EJB. Verify all the entries. The client uses these references to subscribe to the topic.

Give the web-context as **Titan13_1** for this application.

Select one of the Entity EJBs and click on the Entity Tab. Click on the deployment settings and generate the default sql by setting the Datasource, userID and password. Select both the following options for the generation.

- **Create Table on Deploy**
- **Delete table on UnDeploy**

This completes the configuration of the Titan12_1App.

Run the verifier to make sure there are no errors.

- Sometimes the verifier complains about transaction demarcation errors. It seems that the xml descriptors don't get updated. Simply go to the transactions tab of the EJB and set the transaction attribute to **required** even though it shows **required**. This triggers the update of the descriptor files needed.

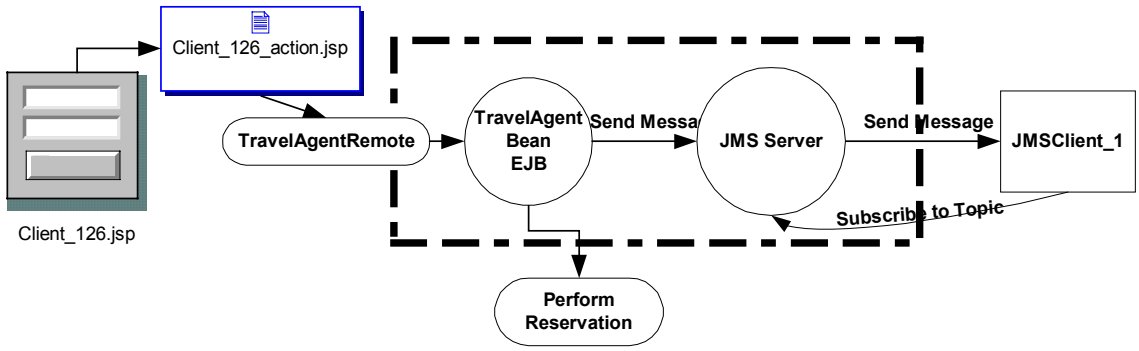
Deploy the application and select the checkbox "return the client Jar". This will generate the required container classes and create the required tables.

Examine the TravelAgent EJB characteristics.

This exercise demonstrates the characteristics of JMS as a resource using Publish-Subscribe model.

The following figure shows the interaction of the components for this exercise.

Figure 1: Simple Publish-Subscribe model for Ex13_1



TravelAgentBean.java

In this exercise the `bookPassage()` method is modified to send the message to the topic after a successful completion of reservation.

```
public TicketDO bookPassage(CreditCardDO card, double price)
throws IncompleteConversationalState {

TopicConnection connect = null;
……..
……..
TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
```

```
String ticketDescription = ticket.toString();
System.out.println("Obtaining reference to jms topic factory EJB");
TopicConnectionFactory factory = (TopicConnectionFactory)
jndiContext.lookup("java:comp/env/jms/TopicFactory");
System.out.println("Obtaining reference to jms TicketTopic ");
Topic topic = (Topic)
   jndiContext.lookup("java:comp/env/jms/TicketTopic");

connect = factory.createTopicConnection();
TopicSession session =
   connect.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
TopicPublisher publisher = session.createPublisher(topic);
TextMessage textMsg = session.createTextMessage();
textMsg.setText(ticketDescription);
System.out.println("Sending text message to jms/TicketTopic..");
publisher.publish(textMsg);
return ticket;
.....
.....
```

Notice that this method does a JNDI lookup for the TopicConnectionFactory. Earlier we set up the coded name for the resource references as **jms/TopicFactory** and **jms/TicketTopic**. These resource references were mapped correspondingly to the resources as defined in the JNDI. Also note the parameters in the `createTopicSession` method call. As per the specification we should be able to ignore the acknowledgement mode since in this exercise Container managed transactions are used. Further the session transacted value if **true** should be valid. Unfortunately it raises an exception that shows the message "`Local transaction already has 1 non-XA Resource: cannot add more resources.`"

Examine the Standard EJB Descriptor File

```
<session>
   <display-name>TravelAgentBean</display-name>
   <ejb-name>TravelAgentBean</ejb-name>
   <home>com.titan.travelagent.TravelAgentHomeRemote</home>
   <remote>com.titan.travelagent.TravelAgentRemote</remote>
   <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
   <session-type>Stateful</session-type>
   <transaction-type>Bean</transaction-type>
   .....
   .....
   <security-identity>
         <description/>
         <use-caller-identity/>
   </security-identity>
   <resource-ref>
```

```
        <res-ref-name>jdbc/Titan</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
    </resource-ref>
    <resource-ref>
        <res-ref-name>jms/TopicFactory</res-ref-name>
        <res-type>javax.jms.TopicConnectionFactory</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
    </resource-ref>
    <resource-env-ref>
        <resource-env-ref-name>
            jms/TicketTopic
        </resource-env-ref-name>
        <resource-env-ref-type>
            javax.jms.Topic
        </resource-env-ref-type>
    </resource-env-ref>
</session>
```

Notice the additional `<resource-ref>` element that describes the reference to the TopicConnectionFactory. The `<resource-env-ref>` element describes the destination for the Publisher. The rest of the descriptor is similar to the one described in the previous chapter.

Examine the Clients for the application

This exercise provides both the java client and the web client. The Web client contains the same JSPs as in the previous chapter. *Client_125.jsp* creates some preliminary data for the rest of the exercise. *Client_126.jsp* and *Client_126_action.jsp* are used in creating the reservation. Once the reservation is finished the TravelAgentBean EJB publishes a message to the topic. The Java client JmsClient_1 will be run from the command line and will listen for the messages for the topic.

JmsClient_1.java

This client program performs the following operations.

- Obtains a reference to the `titan-TopicFactory` connection factory.
- Obtains a reference to the Topic `titan-TicketTopic`.
- Opens a connection and creates a topic session.
- Creates a Topic Subscriber and sets itself as the message listener for the topic.

```
public class JmsClient_1 implements javax.jms.MessageListener {
……..
```

```
……..
public JmsClient_1() throws Exception
{

    Context jndiContext = getInitialContext();

    TopicConnectionFactory factory = (TopicConnectionFactory)
    jndiContext.lookup("titan-TopicFactory");

    Topic topic = (Topic) jndiContext.lookup("titan-TicketTopic");
    TopicConnection connect = factory.createTopicConnection();

    TopicSession session =
        connect.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

    TopicSubscriber subscriber = session.createSubscriber(topic);

    subscriber.setMessageListener(this);

    System.out.println("Listening for messages on titan-
TicketTopic...");
    connect.start();
}

public void onMessage(Message message)
{
    try {
        TextMessage textMsg = (TextMessage)message;
        String text = textMsg.getText();
        System.out.println("\n RESERVATION RECEIVED:\n"+text);
    } catch(JMSException jmsE) {
        jmsE.printStackTrace();
    }
}
……..
}
```

This client program implements the **javax.jms.MessageListener** interface. The client program listens for the messages and when a message arrives, the `onMessage()` method is invoked. The program then prints the receipt of the Reservation.

Running the web and java clients for the Exercise

Start the Java client program to listen for the messages and then deploy the application.

Open a command prompt and go to the directory where the application has been loaded.

There are multiple ways to run the Java client. You can set up the environment and run the client as shown below:

If the application was deployed and the application client is returned to *C:\EJBBOOK\build\deploymentUnits\ears directory*, change directory accordingly. Verify that the *Titan13_1App.ear* is in the directory also.

Run the following command from the directory as shown below and provide the userID and password so that client program starts listening for messages.

```
C:\EJBBOOK\build\deploymentUnits\ears>runclient -client
Titan13_1App.ear -name JmsClient_1 -textauth
Initiating login ...
Enter Username:scott
Enter Password:tiger
……..
Warning : Illegal connection factory access to titan-TopicFactory.
J2EE Components should access JMS Connection Factories through a
resource-ref in java:comp/env
Java(TM) Message Service 1.0.2 Reference Implementation (build b14)
Listening for messages on titan-TicketTopic...
```

This warning can be ignored or the Java client program can be set to use the `<resource-ref>` element and these can be set for the Client Jar as part of the *ear*. If you have trouble running the *runclient* shell for the Java Client you can also do the following, in this case you need the JNDI names rather than the coded name for the reference.

```
C:\> setenv   ( $J2EE_HOME\bin should be in the path)
C:\> cd EJBBook\build\ex13_1
C:\EJBBook\build\ex13_1>java -cp %cpath%;c:\ejbbook\build\ex13_1
com.titan.clients.JmsClient_1
```

This will start the client listening for the messages.

Open the Browser and go to *http://localhost:8000/Titan13_1*. Select the *Client_125.jsp* to pre-populate the database with some data.

Next select *Client_126.jsp* and enter the information based on the CruiseID displayed for the lookup of the table data.

- CustomerID: **1**
- CruiseID: **42** (Depends on the Sequence ID)
- Cabin ID: **108**
- Amount: **1000.00**

The output for the JSP is as shown below.

```
Client_126_action Results of bookPassage call
Return to Client_126.jsp
```

```
Customer ID = 1
Cruise ID = 42
Cabin ID = 108
Price = 1000.00
Finding reference to Customer 1
Starting TravelAgent Session...
Setting Cruise and Cabin information in TravelAgent..
Booking the passage on the Cruise!
Result of bookPassage:
Bob Smith has been booked for the Bermuda or Bust cruise on ship
Nordic Prince. Your accommodations include Suite 108 a 1 bed cabin
on deck level 1. Total charge = 1000.0
```

The output in the command window will be as shown below:

```
RESERVATION RECEIVED:
Bob Smith has been booked for the Bermuda or Bust cruise on ship
Nordic Prince.
Your accommodations include Suite 108 a 1 bed cabin on deck level 1.
Total charge = 1000.0
```

This client program keeps listening continuously. After the successful run of the client programs, undeploy the application before proceeding to the next Exercise.

Exercise 13.2: The Message-Driven Bean (MDB)

This final exercise will utilize all the EJBs from the previous chapters. You will also notice an additional EJB called ReservationProcessBean which is a Message Driven Bean. This exercise explores the usage of the JMS Queue and JMS Reply-To Queues.

In this exercise we will be using the pre-built archive and walk through the configuration of the additional components needed for running the examples.

Building the application for Exercise 12.2

1. Download the *ex12-2.jar* file from the download site. Downloads may be available for the complete bundle of Exercises.
2. Copy the downloaded file and extract the file using *WinZip* or *jar* with `xvf` flags to *$EXAMPLES_HOME*. Example *c:\> EJBBook* directory.
3. Change directory to *$EXAMPLES_HOME\src\ex12_2*.
4. Open a command prompt and type `Ant` or `Ant -buildfile build.xml.`

This should create *$EXAMPLES_HOME\build\ex12_2* directory and compile the required files.

$EXAMPLES_HOME\pre-built\ex12_2 directory contains pre-packaged *ear* file and the client *jar* file for this exercise.

This example will use the pre-built application archive to demonstrate the concepts. Readers are encouraged to go through the process of creating their own archives after going through the pre-built example.

Database schema for the Cabin EJB

This exercise uses all the tables generated by the deployment process as well as previously created PaymentBeanTable and SequenceTable tables.

Building the Application Archive for Ex13_1

Before we look into the contents of the Application archive and add components to it, let us understand the example.

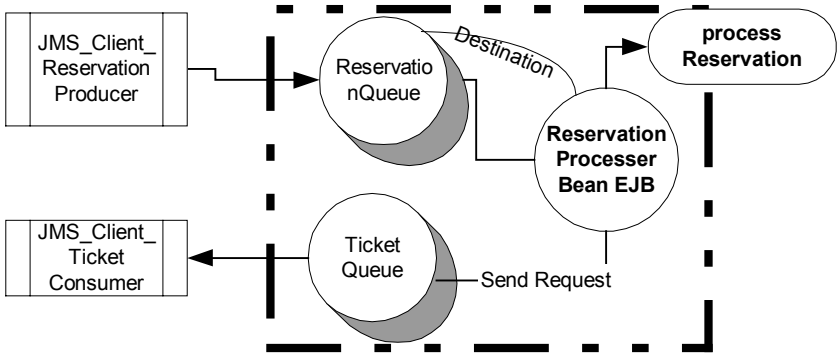
This exercise introduces MDB which is a server-side component and not accessible to the client. This is a component that processes messages asynchronously. This exercise demonstrates the MDB listening for a Reservation Queue for messages. The TravelAgentBean EJB bookPassage logic is not used to create reservations, but similar logic is placed in the MDB.

- *JmsClient_ReservationProducer* sends a message to the ReservationQueue to book a reservation.

- ReservationProcessorBean MDB listens for this message and then extracts the information and obtains the corresponding EJB references to create a reservation.
- Upon successful completion of Reservation it sends a message to the Ticket Queue passing the serializable TicketDO value object.
- *JMSClient_TicketConsumer* listens for the Ticket Queue and receives the message from the Ticket Queue and acknowledges the ticket creation.

The following figure shows the interaction between the components.

Figure 2: Message Drive Bean interaction with the Queues



Configuration for the Queues and the ReservationProcesserBean MDB

This exercise uses JMS Queues. You need to create two Queues and a Queue Connection factory. Use the j2eeAdmin utility to create the corresponding resources as shown below.

```
C:\> j2eeamdin -addJmsFactory titan-QueueFactory queue
```

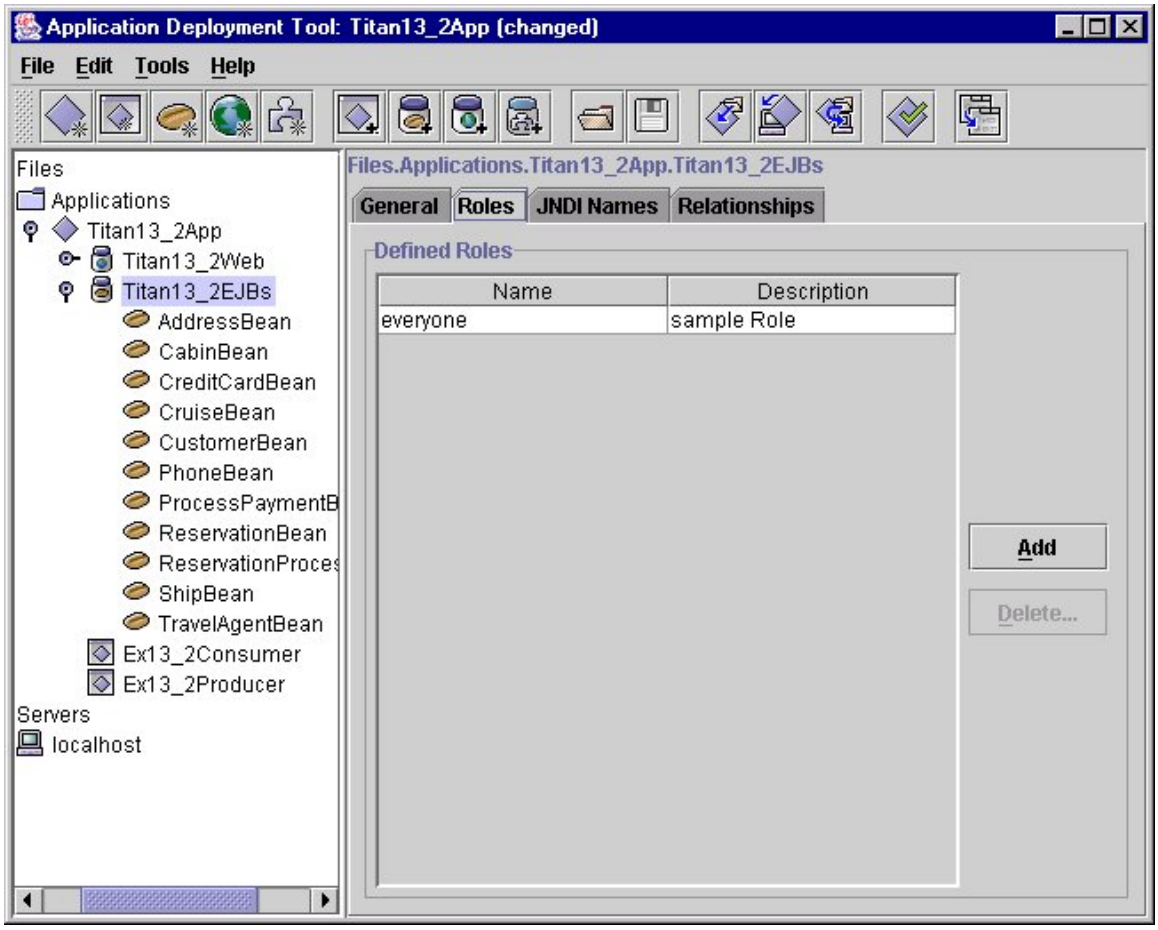
This creates the required javax.jms.QueueConnectionFactory

```
C:\> j2eeadmin -addJmsDestination titan-ReservationQueue queue
C:\> j2eeadmin -addJmsDestination titan-TicketQueue queue
```

These commands create the required queues for this exercise. Verify the corresponding resources by using the `list<ResourceType>` option using `J2eeadmin` utility or browse in the server configuration in the Deploytool.

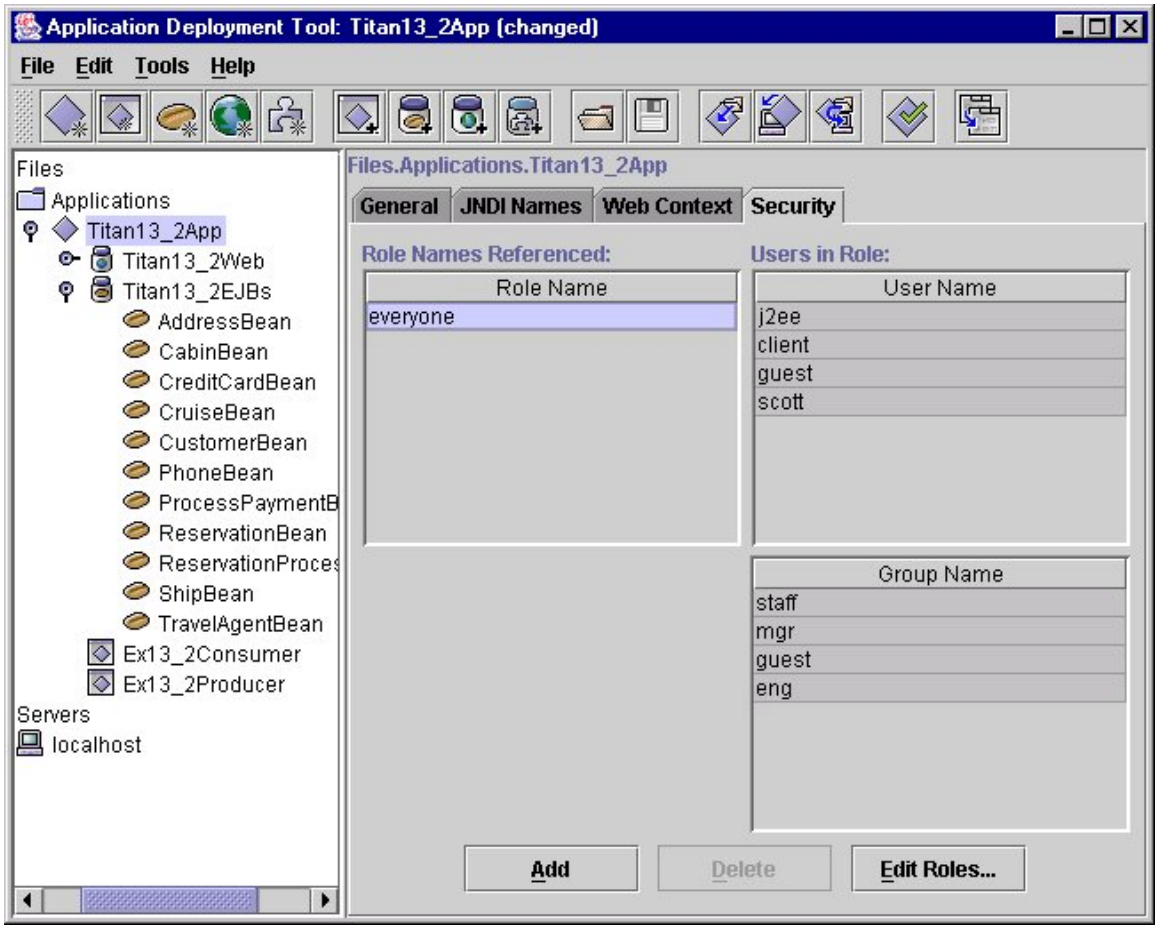
1. ReservationProcesserBean EJB is a message driven bean and is instantiated by the server. The server thread runs this EJB in the context of the container. It cannot be run with the caller-identity as the container and the JMS Server do the invocation of any method on this EJB. The following section examines the configuration of the MDB. The rest of the components in the application archive are similar to those in the previous exercise.
2. Click on the EJB Jar node for the Titan13_2EJBs and add a new Role called **everyone.**

Figure 3: Role Creation



3. Click on the **Titan13_2App** node for the application and select the **security** tab and click on the **Edit Roles** button to add the newly created role **everyone**.
4. Select the **everyone** role in the pane and click the **Add** button to add the users and the groups to the role.
5. Add at least the user **scott** to this role. The following figure shows the configuration with all the users and the groups added to this role.

Figure 4: User/Group mapping for the Role for this application



This sets up the required users with the given role of **everyone** available for this application.

6. Select the **Resource Refs** tab for the `ReservationProcesserBean` EJB and click on the **Add** button. Enter the following information for creating a reference to the QueueConnectionFactory.
    - Coded Name: **jms/QueueFactory**
    - Type: **javax.jms.QueueConnectionFactory**
    - Authentication: **Container**
    - Sharable: Select the **Checkbox**.
    - JNDI-Name: **titan-QueueFactory**

- UserID: **scott**
- Password: **tiger**

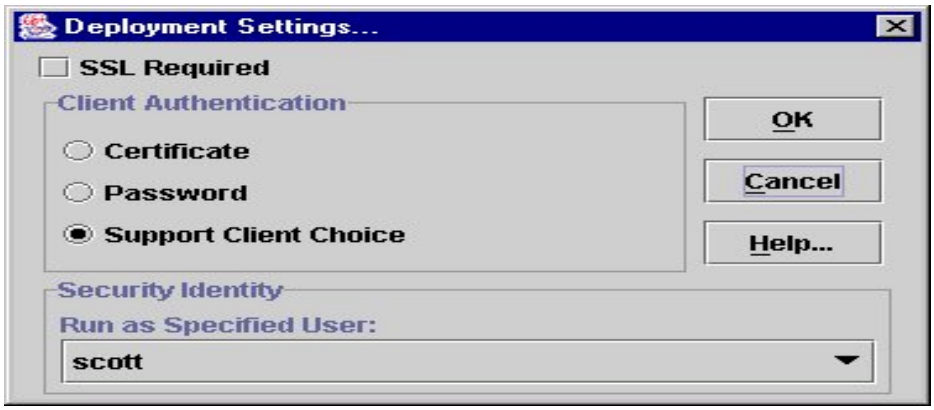
7. Click on the **Resource Environment References** tab and click the **Add** button to add the following information.
   - Coded Entry: **jms/TicketQueue**
   - Type: **javax.jms.Queue**
   - JNDI-Name: **titan-TicketQueue**

Notice that the Resource Environment reference is only for the TicketQueue which is used to send the message. Reservation queue is the destination for this MDB. This MDB listens for messages on the ReservationQueue and upon processing the message places the response message on the TicketQueue.

8. Select the `ReservationProcesserBean` EJB node and click on the **Message** tab. Select the corresponding JMS Destinations and the JMS Connection factory as shown below.
   - Radio Button Option: **Queue**
   - Destination: **titan-ReservationQueue**
   - Connection Factory: **titan-QueueFactory**
   - JMS Message-Selector: MessageFormat = 'Version 3.4'
9. Select the **Transaction** tab and verify that the MDB uses container managed transactions and has the transaction attribute set to **Required** for the `onMessage(..)` method.

The next few figures show the settings as well as walk through the security settings for the ReservationProcessorBean EJB.

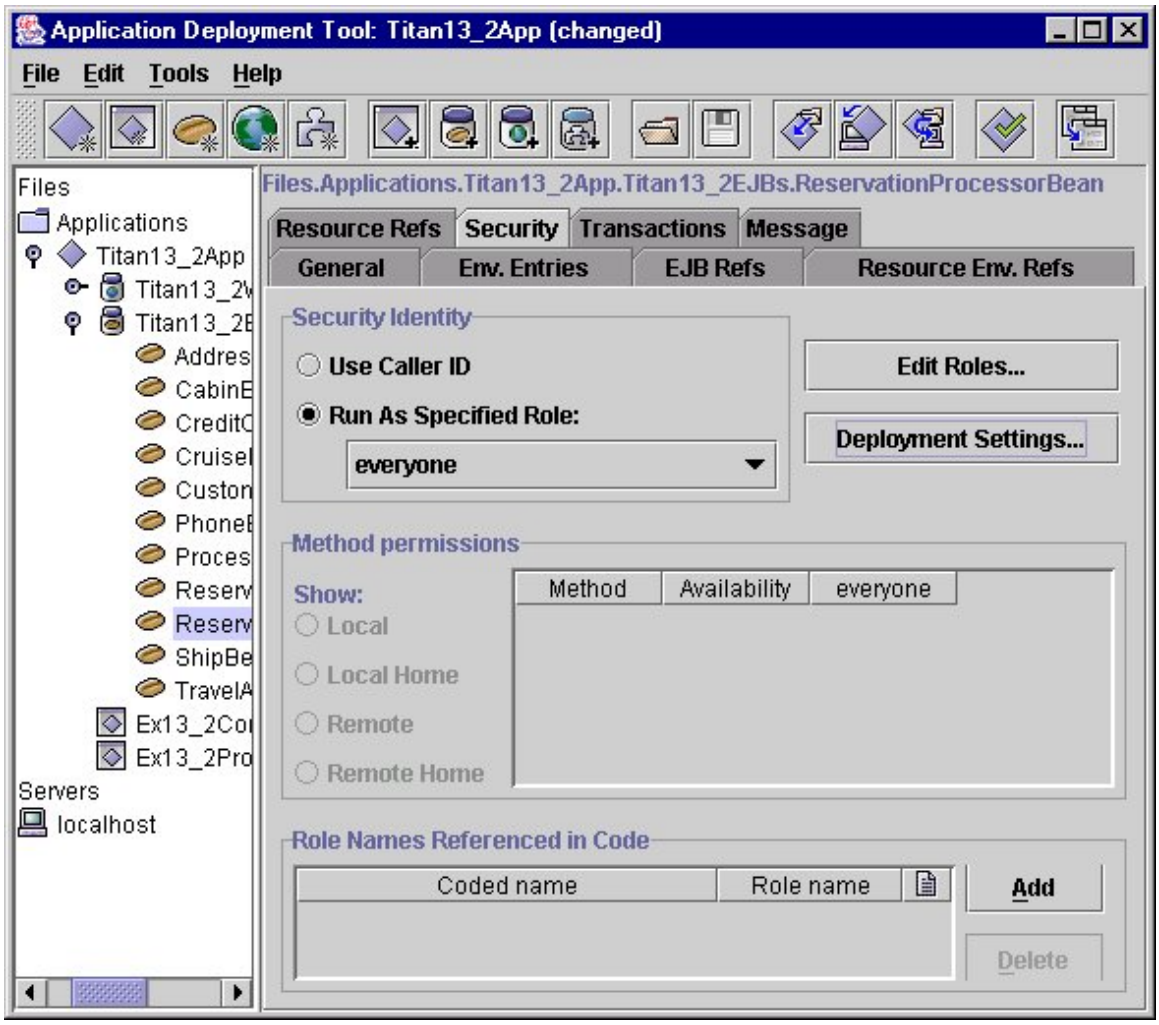
Figure 5: Security settings for the MDB



10. Select the **Security** tab for the MDB and click on the **Edit Roles** button. Click **Add** and enter the **everyone** role. Select **Run as specified Role** and select **everyone** from the drop down list. Click on the deployment settings and select **scott** as the user as shown above.

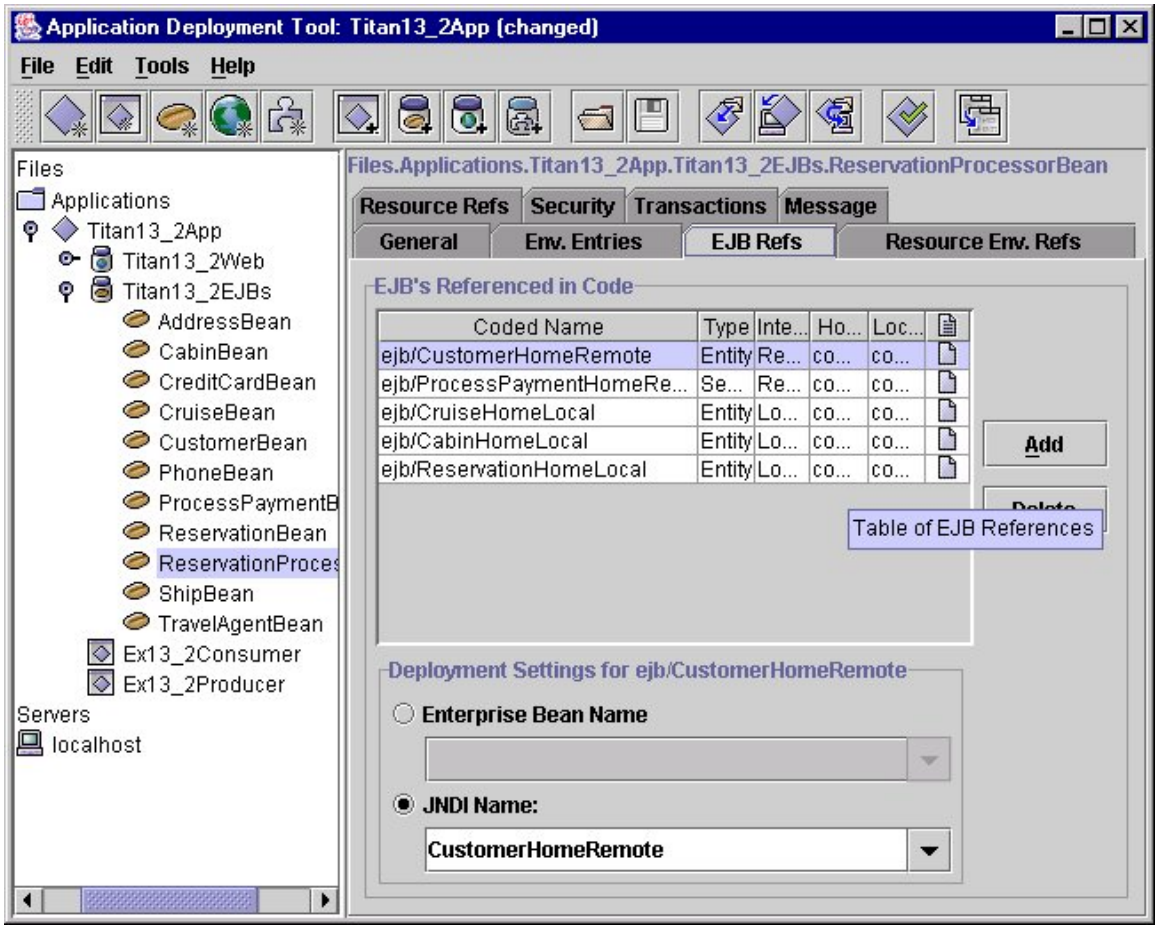
The final settings for the security tab should be as shown below.

Figure 6: Security settings overview for the MDB



This MDB references CustomerBean and ProcessPaymentBean through remote interfaces, and CabinBean, CruiseBean and ReservationBean through local interfaces. Verify the configuration for all the EJB references in the EJB Refs tab for the ReservationProcessorBean MDB.

Figure 7: EJB References for the ReservationProcessorBean EJB



11. Configure the *EX13_2Producer* and *EX13_2Consumer* client applications with corresponding resource references. Both the Java clients need Resource References for the javax.jms.QueueConnectionFactory.

EX13_2Producer has Resource Environment reference for both Ticket and Reservation Queues. This client sets the reference for send-queue as part of the message delivery.

EX13_2Consumer listens for messages on TicketQueue, so has reference only to the `titan-TicketQueue`.

12. Generate the SQL setting all the JDBC resources as mentioned in Exercise 13_1 for jdbc/Titan and jdbc/TitanUID data sources.

The application is ready for Deployment.

Examine the java files for the exercise

ReservationProcessorBean.java

MDB does not have any interfaces and only consists of the Bean class. ReservationProcessBean implements `javax.ejb.MessageDrivenBean` and `javax.jms.MessageListener` interfaces.

```
public class ReservationProcessorBean
implements javax.ejb.MessageDrivenBean, javax.jms.MessageListener {

    MessageDrivenContext ejbContext;
    Context jndiContext;

    public void setMessageDrivenContext(MessageDrivenContext mdc){
        ejbContext = mdc;
        try {
           jndiContext = new InitialContext();
        } catch(NamingException ne) {
           throw new EJBException(ne);
        }
    }
……….
}
```

The `setMessageDrivenContext(..)` is one of the methods required to satisfy the implementation of the MessageDrivenBean interface along with the `ejbRemove()` method.

The MessageListener interface requires the objects that implement the interface to define the `onMessage(..)` method.

```
public void onMessage(Message message) {
   try {

   System.out.println("ReservationProcessor::onMessage() called..");
    MapMessage reservationMsg = (MapMessage)message;

   Integer customerPk = (Integer)
                    reservationMsg.getObject("CustomerID");
   Integer cruisePk =   (Integer)
                    reservationMsg.getObject("CruiseID");
   Integer cabinPk =    (Integer)
                    reservationMsg.getObject("CabinID");

   double price = reservationMsg.getDouble("Price");
```

```
   String creditCardNum = reservationMsg.getString("CreditCardNum");
   Date creditCardExpDate = new Date(
                        reservationMsg.getLong("CreditCardExpDate") );
   String creditCardType = reservationMsg.getString(
                              "CreditCardType" );
   CreditCardDO card = new CreditCardDO(
                     creditCardNum,creditCardExpDate,creditCardType);

   System.out.println("Customer ID = "+customerPk+", Cruise ID =
"+cruisePk+", Cabin ID = "+cabinPk+", Price = "+price);

  CustomerRemote customer = getCustomer(customerPk);
  CruiseLocal cruise = getCruise(cruisePk);
  CabinLocal cabin = getCabin(cabinPk);

  ReservationHomeLocal resHome = (ReservationHomeLocal)
jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");

  ReservationLocal reservation =
     resHome.create(customer, cruise, cabin, price, new
                       java.sql.Date(System.currentTimeMillis()));
  Object ref =
jndiContext.lookup("java:comp/env/ejb/ProcessPaymentHomeRemote");

  ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
                    PortableRemoteObject.narrow(ref,
                             ProcessPaymentHomeRemote.class);
  ProcessPaymentRemote process = ppHome.create();
  process.byCredit(customer, card, price);

   TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
   System.out.println("Created Ticket and Delivering ticket");
   deliverTicket(reservationMsg, ticket);
 }
 catch(Exception e)
 {
   e.printStackTrace();
   throw new EJBException(e);
 }
}
```

When the MDB receives a message from the ReservationQueue it executes the `onMessage(..)` method. In the code above notice that it extracts information from the `MapMessage` and then obtains references to the CustomerBean, CabinBean, CruiseBean EJBs and then creates a ReservationBean EJB. It further processes the reservation using the ProcessPaymentBean EJB and then creates the TicketDO value object to deliver the message to the TicketQueue.

The following code shows the `deliverTicket()` method where the MDB gets a connection to the TicketQueue and places the ticketDO object. Since the TicketDO is a serializable object the JMSClient_TicketConsumer class path should have the TicketDO available for it to de-serialize the object and display the information. We will set the environment correspondingly when we run the clients.

```
public void deliverTicket(MapMessage reservationMsg, TicketDO
ticket)
   throws NamingException, JMSException {

 // create a ticket and send it to the proper destination
 System.out.println("ReservationProcessor::deliverTicket()..");

 Queue queue = (Queue)reservationMsg.getJMSReplyTo();
 System.out.println(queue.getQueueName());
 QueueConnectionFactory factory = (QueueConnectionFactory)
jndiContext.lookup("java:comp/env/jms/QueueFactory");
 QueueConnection connect = factory.createQueueConnection();
 QueueSession session = connect.createQueueSession(false,0);
 QueueSender sender = session.createSender(queue);
 ObjectMessage message = session.createObjectMessage();
 message.setObject(ticket);
 System.out.println("Sending message back to sender..");
 sender.send(message);
 connect.close();
}
```

This code demonstrates the JMS-Reply-To usage. Notice that we do not hardcode the TicketQueue in this method. The JMSClient_ReservationProducer sets the replyTo queue when sending the reservation request to the ReservationQueue. The MDB after receiving the message from ReservationQueue inquires the message for the Reply-To queue and then sends the ticket to the corresponding queue.

JMSClient_ReservationProducer.java

This client program obtains references to the Queue Connection Factory, creates the corresponding connections and then populates the MapMessage with the name-value pairs and sends the message to the ReservationQueue.

```
.....
QueueConnectionFactory factory =
(QueueConnectionFactory)jndiContext.lookup("java:comp/env/jms/QueueF
actory");
Queue reservationQueue =
(Queue)jndiContext.lookup("java:comp/env/jms/ReservationQueue");
        Queue ticketQueue =
(Queue)jndiContext.lookup("java:comp/env/jms/TicketQueue");
        QueueConnection connect = factory.createQueueConnection();
```

```
        QueueSession session = connect.createQueueSession(false,0);
        QueueSender sender = session.createSender(reservationQueue);
        for (int i = 0; i < count; i++)
        {
            MapMessage message = session.createMapMessage();
            //Used in ReservationProcessor to send Tickets.
            message.setJMSReplyTo(ticketQueue);

            message.setStringProperty("MessageFormat", "Version
                                                        3.4");
            message.setInt("CruiseID", cruiseID.intValue());
            message.setInt("CustomerID",i%2+1); // either Customer 1
                        or 2, all we've got in database
            message.setInt("CabinID",i%10+100);// cabins 100-109
                                                    only
            message.setDouble("Price", (double)1000+i);
            // the card expires in about 30 days
            Date expDate = new
                      Date(System.currentTimeMillis()+43200000);
            message.setString("CreditCardNum", "923830283029");
            message.setLong("CreditCardExpDate", expDate.getTime());
            message.setString("CreditCardType",
                                      reditCardDO.MASTER_CARD);
            System.out.println("Sending reservation message #"+i);
            sender.send(message);
            ……
        }
```

JMSClient_TicketConsumer.java

This client program listens for the messages on the TicketQueue and once it receives the messages it displays the information about the Ticket.

```
 public JmsClient_TicketConsumer() throws Exception
{
      Context jndiContext = getInitialContext();
      QueueConnectionFactory factory = (QueueConnectionFactory)
      jndiContext.lookup("java:comp/env/jms/QueueFactory");
      Queue ticketQueue = (Queue)
               jndiContext.lookup("titan-TicketQueue");
    QueueConnection connect = factory.createQueueConnection();
      QueueSession session =

  connect.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
       QueueReceiver receiver = session.createReceiver(ticketQueue);
       receiver.setMessageListener(this);
```

```
        System.out.println("Listening for messages on titan-
                                    TicketQueue...");
         connect.start();
  }
public void onMessage(Message message)
{
  try
  {
    ObjectMessage objMsg = (ObjectMessage)message;
    TicketDO ticket = (TicketDO)objMsg.getObject();
    System.out.println("*********************************");
    System.out.println(ticket);
    System.out.println("*********************************");
  } catch (JMSException jmsE)
  {
    jmsE.printStackTrace();
  }
  System.out.println("finished consuming");
}
```

Examine the ejb-jar.xml descriptor file

In this section we examine the ReservationProcessorBean section of the *ejb-jar.xml* file. The rest of the descriptor file is very much an inclusion of Ex13.1 Entity Beans and Session Bean.

The ReservationProcessorBean is represented in a `<message-driver>` element and contains both `<ejb-ref>` and `<ejb-local-ref>` elements for the corresponding EJBs that it performs `lookup` using the Remote and local interfaces. It also contains the Resource references for the Queue Connection Factory as well as the destination queues.

```
<message-driven>
   <display-name>ReservationProcessorBean</display-name>
   <ejb-name>ReservationProcessorBean</ejb-name>
   <ejb-class>
     com.titan.reservationprocessor.ReservationProcessorBean
   </ejb-class>
   <transaction-type>Container</transaction-type>
   <message-selector>
      MessageFormat =  'Version 3.4'
   </message-selector>
   <message-driven-destination>
         <destination-type>javax.jms.Queue</destination-type>
   </message-driven-destination>
   ……
      ……..
   <security-identity>
```

```
            <description/>
            <run-as>
                  <description/>
                  <role-name>everyone</role-name>
            </run-as>
      </security-identity>
      <resource-ref>
            <res-ref-name>jms/QueueFactory</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/TicketQueue</resource-env-ref-name>
         <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
      </resource-env-ref>
   </message-driven>
```

The MDB does not have local or remote interfaces. Client programs cannot create or invoke methods on the MDB.

The `<message-selector>` element instructs the container to filter the messages with the property name and value and only send those messages to this MDB. In this case the property is `MessageFormat` and the value is set to `Version 3.4`.

The Destination is defined using the `<message-driven-destination>` element as of type javax.jms.Queue. Notice that the destination type is defined but not the actual queue that is used. The actual queue name mapping is provided in the proprietary *sun-j2ee-ri.xml*.

The rest of the descriptor elements show the Resource reference to the QueueConnection Factory as well as the resource environment references to both the Ticket and the Reservation Queues.

Examine and run the client application

This exercise also uses the Web clients for testing the application. Deploy the application after running through the verifier. Make sure there are no errors other than the one mentioned below.

- Ignore the following failure when running through the verifier. The error is as shown:
  - Item: EJB Deployment Descriptor
  - Test Name: tests.dd.ParseDD
  - Result: Failed
  - [EJB] method-intf cannot be Bean. It has to be either Local, Remote, LocalHome or Home

Give the application a web-context of Titan13_2. Go to *http://localhost:8000/Titan13_2*. This provides a listing of client programs for this exercise. Select the *Client_125.jsp* to pre-populate the database with the some data.

Client_125.jsp

This client program builds the sample data that will be used for the remainder of the exercise. It uses the TravelAgentBean to create Customer, Cabin, Ship, and Cruise EJBs and saves the information to the database. It further creates two reservations for each CustomerBean EJB.

The output of the program is as shown below:

```
Created customers with IDs 1 and 2..
Created ships with IDs 101 and 102..
Created cabins on Ship A with IDs 100-109
Created cabins on Ship B with IDs 200-209
Created cruises on ShipA with IDs 46, 47, 48
Created cruises on ShipB with IDs 49, 50, 51
Made reservation for Customer 1 on Cruise 46 for Cabin 103
Made reservation for Customer 1 on Cruise 51 for Cabin 208
Made reservation for Customer 2 on Cruise 47 for Cabin 105
Made reservation for Customer 2 on Cruise 51 for Cabin 202
```

CustomerBean, CabinBean and ShipBean EJBs have their primary key provided by the client program, in this case the TravelAgentBean. CruiseBean and ReservationBean on the other hand use the SequenceTable to request the primary key. Readers can select the required CruiseID, CabinID and ShipID for the next client program from this output.

JMSClient_TicketConsumer

This client program listens for the messages on the ticket queue. As we mentioned above that we need TicketDO in the class path and since we are running these clients through the runclient shell change the *setenv.bat* file in the *$J2EE_HOME\bin* directory and add the following:

```
SET EX13_2PATH=C:\EJBBOOK\build\ex13_2
set
CPATH=%CLOUDJARS%;%CLASSESDIR%;%JMS_CLASSESDIR%;%J2EEJARS%;%J2EETOOL
_CLASSES%;%J2EETOOL_JAR%;%LOCALEDIR%;%J2EE_CLASSPATH%;%JAVATOOLS%;%J
AVAHELPJARS%;%EX13_2PATH%
```

Open a command prompt and type the following command:

```
C:\ejbbook\build> runclient -client Titan13_2App.ear -name
Ex13_2Consumer -textauth
```

The command shell should prompt for userID and password. Type **scott** and **tiger** for userID and password. This client program will now listen for messages on the ticketQueue. The output should be as shown below:

```
C:\EJBBOOK\build>runclient -client Titan13_2App.ear -name
Ex13_2Consumer -textauth
Initiating login ...
```

```
Enter Username:scott
Enter Password:tiger
Binding name:`java:comp/env/jms/TicketQueue`
Binding name:`java:comp/env/jms/QueueFactory`
Java(TM) Message Service 1.0.2 Reference Implementation (build b14)
Listening for messages on titan-TicketQueue...
```

JMSClient_ReservationProducer

The reservation producer client takes two command line arguments, namely cruiseID and the number of reservations. The program run with Java command is in the form of

```
JmsClient_ReservationProducer <cruiseID> <numrez>
```

As you did with the JMSClient_TicketConsumer client, go to *c:\ejbbook\build* directory and execute the following command in a different command shell.

```
C:\EJBBOOK\build>runclient -client Titan13_2App.ear -name
Ex13_2Producer 47 2 -textauth
Initiating login ...
Enter Username:scott
Enter Password:tiger
Binding name:`java:comp/env/jms/TicketQueue`
Binding name:`java:comp/env/jms/ReservationQueue`
Binding name:`java:comp/env/jms/QueueFactory`
Java(TM) Message Service 1.0.2 Reference Implementation (build b14)
Sending reservation message #0
Sending reservation message #1
C:\EJBBOOK\build>
```

Notice that the `cruiseID` is sequence generated. This output uses 47 as cruiseID based on the information of the *Client_125.jsp*. Towards the bottom of the *Client_125.jsp* you can see the contents of all the tables, including `CruiseBeanTable`.

Once the Reservation Producer sends the reservation requests it terminates. Now notice the J2EE Server output as well as the Ticket Consumer window output.

J2EE Server output:

```
ReservationProcessor::onMessage() called..
Customer ID = 1, Cruise ID = 47, Cabin ID = 100, Price = 1000.0
ReservationBean::ejbCreate
……..
ReservationBean::ejbPostCreate
process() with customerID=1 amount=1000.0
Obtained Connection
Execute on the PaymentBeanTable
Got customer Name
Finished the description of the Ticket
Created Ticket and Delivering ticket
```

```
ReservationProcessor::deliverTicket()..
titan-TicketQueue
Sending message back to sender..
ReservationProcessor::onMessage() called..
Customer ID = 2, Cruise ID = 47, Cabin ID = 101, Price = 1001.0
ReservationBean::ejbCreate
.....
ReservationBean::ejbPostCreate
process() with customerID=2 amount=1001.0
Obtained Connection
Execute on the PaymentBeanTable
Got customer Name
Finished the description of the Ticket
Created Ticket and Delivering ticket
ReservationProcessor::deliverTicket()..
titan-TicketQueue
Sending message back to sender..
```

Notice the `onMessage()` execution on the MDB as it goes through processing each reservation request and sends back a message to the titan-TicketQueue.

JMSClientConsumer output:

```
*******************************
Bob Smith has been booked for the Norwegian Fjords cruise on ship
Nordic Prince.
 Your accommodations include Suite 100 a 1 bed cabin on deck level
1.
 Total charge = 1000.0
*******************************
finished consuming
*******************************
Joseph Stalin has been booked for the Norwegian Fjords cruise on
ship Nordic Prince.
 Your accommodations include Suite 101 a 1 bed cabin on deck level
1.
 Total charge = 1001.0
*******************************
finished consuming
```

This shows the client continuously listening for messages on the queue and displays the ticket information for the processed reservations.

This finishes the exercise as well as the chapter and the workbook. Readers are encouraged to change the build files to package and verify using the ant build mechanism rather than DeployTool. In the workbook we took a novel approach of using the tool mostly to configure and deploy the components. It is very difficult and time-consuming to use this approach in an application with lot of components. Since J2EE™ SDK is a reference implementation and should

only be used to validate the specifications we wanted to harness the power of the tools provided by the toolkit.

Happy coding and enjoy EJB 2.0 and upcoming J2EE APIs.