Exercises for Chapter 12

Exercise 10.1: A Stateless Session EJB

This exercise demonstrates a simple Stateless Session EJB that persists payment information during the booking process.

Building the application for Exercise 12.1

1. Download the *ex12-1.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_1*.
4. Open a command prompt and type `Ant` or `Ant -buildfile build.xml`

This should create *$EXAMPLES_HOME\build\ex12_1* directory and compile the required files.

$EXAMPLES_HOME\pre-built\ex12_1 directory contains pre-packaged *.ear* file and the client *.jar* file for this exercise.

SQL for creating the database table **PaymentBeanTable is** provided in *$EX12_1_HOME/sql* directory.

Also a build target is provided to create the table if the cloudscape server is running.

Database schema for the Cabin EJB

The build file has a target `create-payment-table` that can be used to create the database table. To use this target you need to have the Cloudscape server running. You can also use *cloudview* to open the database and create the required database table.

You can also create one with the following schema:

```
CREATE TABLE "PaymentBeanTable"
   (
      "_customer_id" INT NOT NULL ,
      "type" CHAR(10),
      "checkBarCode" VARCHAR(50),
      "checkNumber" INT,
      "creditNumber" CHAR(20),
      "creditExpDate" DATE,
      "amount" DECIMAL(10,2)
);
```

Building the Application Archive for Ex12_1

1. Run the *Deploytool* by typing in at command prompt `c:\>deploytool`
2. Create a new application and call it *Titan12_1App.ear*.

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

This exercise will use the *CustomerBean* EJB, *AddressBean* EJB along with the `Name` and `AddressDO` objects from the exercise 6.3.

You can use the pre-built application archive. This exercise provides high-level steps to create the archive from the compiled classes.

1. Create a new EJB Jar and give it a name **Titan12_1EJBs**. Add the *com* directory from the *$EXAMPLES_HOME\build\ex12_1* directory to the EJB *.jar*.
2. Create the *CustomerEJB* and *AddressEJB* as Entity EJBs with CMP 2.0 support and also create the one-one relationship between *CustomerEJB* and *AddressEJB*.
3. Create *ProcessPayment* EJB as a Stateless Sesssion EJB. This EJB uses `javax.sql.DataSource` to connect to the Database.
4. Take the default selection for transaction management, i.e. container managed transaction. Select the default **Use Caller ID** for security Identity.
5. Click on the **ProcessPaymentBean** Node in the *Deploytool* and select the **Resource Ref** tab.
6. Click on the **Add** button and add the following :
    - Coded Name: **jdbc/Titan**
    - Type: **javax.sql.DataSource**
    - Authentication: **Container**
    - Sharable: Select the checkbox
7. Click on the **EJB Refs** tab and then click on the **Add** button.
8. This EJB references the *CustomerEJB* through the remote interface. We need to add the reference information here. Add the following information.
    - Coded Name: **ejb/CustomerHomeRemote**
    - Type: **Entity**
    - Interfaces: **Remote**
    - Home Interface: **com.titan.customer.CustomerHomeRemote**
    - Local/Remote Interface : **com.titan.customer.CustomerRemote**
9. Select the **JNDI Name** radio button and enter **CustomerHomeRemote** and the name.

ProcessPayment EJB also uses a threshold value for the check number from the environment.

10. Add an environment entry for the EJB.
11. Click on the **Env. Entries** tab and click the **Add** button to add the following information.
    - Coded Entry: **minCheckNumber**
    - Type: **Integer**
    - Value: **2000**

This example also provides the Java client application and is left to the reader to create the application client *.jar* and test it.

12. Select **File→New→Web Component** from the menu to create a new web archive. Select all JSPs from *$EXAMPLES_HOME/build/ex12_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.
13. Give the web-context as *Titan12_1* for this application.

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 mark the return of the client *.jar*. This should generate the required container classes and create the required **CustomerBeanTable** and **AddressBeanTable** in the database.

Examine the ProcessPayment EJB characteristics.

This exercise consists of a two-client application. We will examine the EJB contents before we run the client application. This exercise demonstrates the characteristics of a Stateless Session Bean with container-managed transactions.

ProcessPaymentRemote.java

Typically Stateless Session EJBs are used for process logic. They have method duration as the scope of the client context and are effectively used for scalability purposes for process logic.

```
public boolean byCheck(CustomerRemote customer, CheckDO check,
double amount) throws RemoteException, PaymentException;
public boolean byCash(CustomerRemote customer, double amount)
      throws RemoteException, PaymentException;
```

```
public boolean byCredit(CustomerRemote customer, CreditCardDO card,
double amount) throws RemoteException, PaymentException;
```

All these pubic methods delegate the request to a private method called `process`. This EJB accesses the Customer EJB for customer information and directly uses the JDBC DataSource to save the payment information to the database.

Recall that a mapping for Resource Reference was set in the packaging process to refer to JDBC/Titan DataSource object.

```
private Connection getConnection() throws SQLException {
         try {
               InitialContext jndiCntx = new InitialContext();
               DataSource ds = (DataSource)
                     jndiCntx.lookup("java:comp/env/jdbc/Titan");
               return ds.getConnection();
         } catch(NamingException ne){throw new EJBException(ne);}
   }
```

The private `getConnection` method is used to obtain a database connection as part of the process method to persist information to the database. Also this method is declaratively marked as using Container-managed transaction. For this reason the access to *CustomerEJB* is also performed within the context of the same transaction even though it is of no consequence in this situation and only the process of writing to the database is important. You can use the explicit transaction handle with Bean Managed transaction using `UserTransaction` around the insertion of Payment information to the database.

```
con = getConnection();
            ps = con.prepareStatement
             ("INSERT INTO \"PaymentBeanTable\" (\"_customer_id\",
               \"amount\", \"type\", \"checkBarCode\",
               \"checkNumber\", \"creditNumber\",
               \"creditExpDate\")"+
                " VALUES (?,?,?,?,?,?,?)");
            ps.setInt(1,customerID.intValue());
            ps.setDouble(2,amount);
            ps.setString(3,type);
            ps.setString(4,checkBarCode);
            ps.setInt(5,checkNumber);
            ps.setString(6,creditNumber);
            ps.setDate(7,creditExpDate);
            int retVal = ps.executeUpdate();
      ……..
```

This is a typical approach to encapsulate the process logic and use Stateless Session Bean to use JDBC directly to persist information to the database for increasing performance of the system. The public exposed methods take in `CustomerRemote EJBObject` that is serializable and use the remote reference to obtain information from the *CustomerEJB* and then process payment.

Examine the Standard EJB Descriptor File

```
<session>
   <display-name>ProcessPaymentBean</display-name>
   <ejb-name>ProcessPaymentBean</ejb-name>
   <home>com.titan.processpayment.ProcessPaymentHomeRemote</home>
   <remote>com.titan.processpayment.ProcessPaymentRemote</remote>
   <ejb-class>com.titan.processpayment.ProcessPaymentBean</ejb-
class>
   <session-type>Stateless</session-type>
   <transaction-type>Container</transaction-type>
   <env-entry>
         <env-entry-name>minCheckNumber</env-entry-name>
         <env-entry-type>java.lang.Integer</env-entry-type>
         <env-entry-value>2000</env-entry-value>
   </env-entry>
   <ejb-ref>
         <ejb-ref-name>ejb/CustomerHomeRemote</ejb-ref-name>
         <ejb-ref-type>Entity</ejb-ref-type>
         <home>com.titan.customer.CustomerHomeRemote</home>
         <remote>com.titan.customer.CustomerRemote</remote>
   </ejb-ref>
   <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>
</session>
```

The session element describes the semantics of the *ProcessPayment* EJB. Notice the `<env-entry>` element that corresponds to the environment variable that was created to be used by the Session Bean. Additionally notice the `<resource-ref>` and `<ejb-ref>` elements that define the corresponding references from the ProcessPaymentBean stateless session EJB to the `java.sql.DataSource` and *CustomerBean* EJB.

Examine and Run the Web Clients for the application

This exercise provides two client programs: *Client_121.jsp* and *Client_122.jsp*.

Client_121.jsp

This client program performs the following operations.

- Obtains a reference to the *CustomerBean* EJB using the remote reference to CustomerHomeRemote.
- Creates a Customer and associates an address to the *CustomerEJB*. This example performs the setup information needed to test the *Client_122.jsp* that uses the stateless session bean.

The following output demonstrates the client side execution upon pointing to the following URL *http://localhost:80001/Titan12_1/Client_121.jsp*.

```
Client_121 Create a Customer and Address for use by subsequent
programs
Creating Customer 1..
Creating AddressDO data object..
Setting Address in Customer 1...
Contents of CUSTOMER/ADDRESS tables
CustomerBeanTable
Table name is :: CustomerBeanTable cloumn count is :: 5 string not
null for delete link creation _homeAddress_id=177 firstName=NULL
hasGoodCredit=false id=1 lastName=NULL Delete Row

Delete all rows in CustomerBeanTable table
AddressBeanTable
Table name is :: AddressBeanTable cloumn count is :: 6 string not
null for delete link creation __reverse_homeAddress_id=1 city=Austin
id=177 state=TX street=1010 Colorado zip=78701 Delete Row

Delete all rows in AddressBeanTable table
```

Finally, we remove the Entity Bean by calling remove on the EJBObject that calls the `ejbRemove` on the *ShipEJB* and deletes the record from the database.

Client_122.jsp

This JSP uses the *ProcessPaymentBean* EJB to process a payment and insert data into the database. It uses the PaymentBeanTable created earlier in the application build process. This client program performs the following steps:

- Obtains a reference to JNDI and performs a lookup on `ProcessPaymentHomeRemote` and `CustomerHomeRemote` to obtain the remote references to the corresponding EJBs. Notice that the JNDI names are used instead of full context information.

```
System.out.println("Looking up home interfaces..");

 Object ref = jndiContext.lookup("ProcessPaymentHomeRemote");
```

```
ProcessPaymentHomeRemote procpayhome = (ProcessPaymentHomeRemote)
PortableRemoteObject.narrow(ref,ProcessPaymentHomeRemote.class);

ref = jndiContext.lookup("CustomerHomeRemote");

CustomerHomeRemote custhome = (CustomerHomeRemote)
PortableRemoteObject.narrow(ref,CustomerHomeRemote.class);
```

- Next it creates the *ProcessPaymentBean* EJB and calls `findByPrimaryKey` to retrieve the *CustomerBean* EJB.

```
ProcessPaymentRemote procpay = procpayhome.create();
CustomerRemote cust = custhome.findByPrimaryKey(new Integer(1));
```

- Then the client program calls the public methods `byCash,` creates a `CheckDO` value object calls `byCheck` and creates the `CreditCardDO` value object and calls `byCredit.`

```
System.out.println("Making a payment using byCash()..");
procpay.byCash(cust,1000.0);

System.out.println("Making a payment using byCheck()..");
CheckDO check = new CheckDO("01001010111010100011", 3001);
procpay.byCheck(cust,check,2000.0);

System.out.println("Making a payment using byCredit()..");
Calendar expdate = Calendar.getInstance();
expdate.set(2005,1,28); // month=1 is February
CreditCardDO credit = new CreditCardDO("370000000000002",
                      expdate.getTime(),"AMERICAN_EXPRESS");
procpay.byCredit(cust,credit,3000.0);

System.out.println("Making a payment using byCheck() with a low
check number..");
CheckDO check2 = new CheckDO("111000100111010110101", 1001);
try
{
   procpay.byCheck(cust,check2,9000.0);
}
catch (PaymentException pe)
{
   System.out.println("Caught PaymentException: "+pe.getMessage());
}
```

Notice the `try-catch` block around the second time that we call the `byCheck` method. This is to make sure that the PaymentException a serializable callable exception is thrown by the *ProcessPaymentBean* EJB since the threshold set for the minimum check number is not met (1001< 2000).

- Finally we remote the EJB from the container by calling the `remove` method on the *ProcessPayment* session bean.

The output of the program should be as follows:

```
Client_122 Example Demonstrating use of ProcessPayment EJB
Looking up home interfaces..
Making a payment using byCash()..
Making a payment using byCheck()..
Making a payment using byCredit()..
Making a payment using byCheck() with a low check number..
Caught PaymentException: Check number is too low. Must be at least
2000
.....
PaymentBeanTable
Table name is :: PaymentBeanTable cloumn count is :: 7 string not
null for delete link creation _customer_id=1 type=CASH
checkBarCode=NULL checkNumber=-1 creditNumber=NULL
creditExpDate=NULL amount=1000.00 Delete Row
string not null for delete link creation _customer_id=1 type=CHECK
checkBarCode=01001010110101010100011 checkNumber=3001
creditNumber=NULL creditExpDate=NULL amount=2000.00 Delete Row
string not null for delete link creation _customer_id=1 type=CREDIT
checkBarCode=NULL checkNumber=-1 creditNumber=370000000000002
creditExpDate=2005-02-28 amount=3000.00 Delete Row
```

This completes the simple ProcessPaymentBean stateless session EJB exercise. Undeploy the application. This process will remove the `CustomerBeanTable` as well as `AddressBeanTable` tables from the database.

Take a break before the next Exercise that introduces the Stateful Session EJB.

Exercise 12.2: A Stateful Session EJB

This exercise will combine most of the EJBs that we have used so far and will add another Stateful Session Bean to the mix. The EJBs used in chapter 7 are modified to accommodate the inclusion of the `TravelAgentBean` Stateful Session EJB. The code in the chapter is not modeled against the example in the EJB book, rather is modeled very similar to the BEA™ Weblogic workbook.

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 the 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 Ex12_1

The following is the list of modifications to the earlier created EJBs and brief description of the contents of the *Titan12_2App.ear* contents:

- *CustomerBean* EJB in this exercise contains both Local and Remote interfaces. It is legal for the EJB to support both of the interfaces, and the exercise demonstrates the use of both types of interface.
- *CustomerBean* EJB uses the value object `AddressDO` to expose the Customer-Address one-to-one relationship to the remote interface.
- *PaymentProcessBean* Stateless Session EJB from 12.1 is included in the mix.

- A new *TravelAgentBean* Stateful Session EJB is added to the application and uses `TicketDO` value object to book passage for the customer and is passed by value to the client via the Remote interface.
- *ReservationBean* EJB is modified and an additional `create` method is added to support the process of creating the reservation using other Entity Beans.
- *TravelAgentBean* is created as a stateful session bean. Stateful session beans preserve the conversational state for a client for a specified amount of time. Stateful session beans are not typically shared between clients.
- *TravelAgentBean* has references to *ShipBean*, *CruiseBean* and *ReservationBean* through local interfaces and to CustomerBean through remote interfaces. We will follow the explanation later when examining the *ejb-jar.xml* descriptor file.
  - It is required by the parser to define the Remote interfaces before the local interface references for the `<session>` element. The order in which the references are defined is the order in which the contents are generated for the *ejb-jar.xml*. The verifier will fail if the *CustomerBean* remote interface is defined after the other EJB for the *TravelAgentBean*. *TravelAgentBean* EJB uses remote interface to communicate with *CustomerBean* and uses local interfaces for the other EJBs.
- *TravelAgentBean* also has reference to *ProcessPaymentBean* EJB. This reference is not defined in the EJB references since it uses the JNDI name for the Bean.
- Notice that the *TravelAgentBean* also has a resource reference defined for the JDBC connection. This EJB uses the JDBC/Titan data source to connect to the database to list the available cabins based on ship and cruise information.
- The Web Application has references to *CustomerBean* and *TravelAgentBean* through remote interfaces.

Examine the java files for the exercise

Before we examine the contents of the Java files let us understand the sequence of operations performed in this exercise.

- The Client program first obtains references to the *CustomerBean*'s remote object and then performs a lookup for the *TravelAgentBean*'s remote home interface.
- It then creates the *TravelAgentBean* EJB by passing in the CustomerRemote EJBObject.
- The client program then invokes methods to set the Cruise and Cabin references for the same Stateful Session Bean. Since this is a stateful session bean, it preserves the references in its instance variables for the duration of the client conversation.
- The client program then invokes remote `bookPassage` method to perform a reservation and book a ticket for the customer. It returns a `TicketDO` value object to the client.

This is the normal flow of operation for this exercise. We will next examine the source files for the modified EJBs as well as *TravelAgentEJB* to understand the concepts.

CustomerHomeRemote.java

```
    public CustomerRemote create(Integer id)
    throws CreateException, RemoteException;

    public CustomerRemote findByPrimaryKey(Integer id)
    throws FinderException, RemoteException;
```

`CustomerHomeRemote` defines create and finder methods very similar to the `CustomerHomeLocal` interface.

CustomerRemote.java

```
public void setAddress(String street, String city,
                           String state, String zip)
        throws RemoteException, CreateException, NamingException;

public void setAddress(AddressDO address)
        throws RemoteException, CreateException, NamingException;

public AddressDO getAddress() throws RemoteException;

public Name getName() throws RemoteException;
public void setName(Name name) throws RemoteException;

public void addPhoneNumber(String number, byte type)
        throws NamingException, CreateException, RemoteException;
public void removePhoneNumber(byte typeToRemove)
        throws RemoteException;
public void updatePhoneNumber(String number, byte typeToUpdate)
        throws RemoteException;
public Vector getPhoneList() throws RemoteException;

public boolean getHasGoodCredit() throws RemoteException;
public void setHasGoodCredit(boolean flag) throws RemoteException;
```

Notice the difference in `getAddress` and `setAddress` business methods that return the value objects that represent the Customer-Address relationship. Contrast these methods to the public methods defined in the CustomerLocal interface to establish the Customer-Address (HomeAddress) relationship.

```
// Customer-homeAddress-Address relationship definition in the local
// interface CustomerLocal.java
public AddressLocal getHomeAddress();
public void setHomeAddress(AddressLocal address);
```

ReservationHomeLocal.java

```
public ReservationLocal create(CustomerRemote customer,
      CruiseLocal cruise, CabinLocal cabin, double price,
      java.sql.Date dateBooked) throws javax.ejb.CreateException;
```

Notice the newly added `create` method that takes in CustomerRemote object along with EJB local objects for *Cruise* and *Cabin* beans. The corresponding definition of the `ejbCreate` and `ejbPostCreate` method is shown below for the *ReservationBean*.

ReservationBean.java

The first method in the *ReservationBean* is a convenience method called `buildSampleData`. This method is used by the first client program *Client_125.jsp* to create some *Customer*, *Cabin*, *Cruise* and *Ship* EJBs and saves the information into the database. Look into the source file for more information.

The main methods `ejbCreate` and `ejbPostCreate` are shown below.

```
public Integer ejbCreate(CustomerRemote customer,
                         CruiseLocal cruise,
                         CabinLocal cabin, double price,
                         java.sql.Date dateBooked)
   throws CreateException
{
   System.out.println("ReservationBean::ejbCreate");
   setAmountPaid(price);
   setDate(dateBooked);
   setId(getNextUniqueKey());
   return null;
}

public void ejbPostCreate(CustomerRemote customer,
                          CruiseLocal cruise,
                          CabinLocal cabin, double price,
                          java.sql.Date dateBooked)
throws javax.ejb.CreateException
{

   System.out.println("ReservationBean::ejbPostCreate");
   setCruise(cruise);
   // Our bean has many cabins, use the cmr set method here..
   Set cabins = new HashSet();
   cabins.add(cabin);
   this.setCabins(cabins);
   try
   {
    Integer primKey = (Integer)customer.getPrimaryKey();
```

```
     javax.naming.Context jndiContext = new InitialContext();
     CustomerHomeLocal home = (CustomerHomeLocal)

     jndiContext.lookup("java:comp/env/ejb/CustomerHomeLocal");
     CustomerLocal custL = home.findByPrimaryKey(primKey);
     // Our bean has many customers, use the cmr set method here..
     Set customers = new HashSet();
     customers.add(custL);
     this.setCustomers(customers);
    }
    catch (RemoteException re)
   {
      throw new CreateException("Invalid Customer - Bad Remote
       Reference");
    }
    catch (FinderException fe)
    {
          throw new CreateException("Invalid Customer - Unable to
          Find Local Reference");
    }
    catch (NamingException ne)
    {
          throw new CreateException("Invalid Customer - Unable to
          find CustomerHomeLocal Reference");
    }
}
```

This demonstrates the *ReservationBean* performing the process of creating reservations by setting the *CruiseBean* and *CabinBean* references and then obtaining a `CustomerLocal` reference by executing a finder method obtaining the primary key from that passed in `CustomerRemote` object.

TravelAgentBean.java

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

   if (customer == null || cruise == null || cabin == null)
   {
          throw new IncompleteConversationalState();
   }
   try
   {
    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("ejb/ProcessPaymentHomeRemote");
    ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
                    PortableRemoteObject.narrow(ref,
                    ProcessPaymentHomeRemote.class);

    ProcessPaymentRemote process = ppHome.create();
    process.byCredit(customer, card, price);
    process.remove();

    TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
    return ticket;
   }
   catch (Exception e)
   {
         throw new EJBException(e);
   }
}
```

This client program invokes this method passing in the required parameters. Notice that an exception is thrown if the previous call to the *TravelAgentBean* does not set the references to the required *Customer*, *Cruise* and *Cabin* EJBs. This demonstrates the conversational state behavior of the Stateful Session EJB. This method then creates a Reservation and invokes the `byCredit`on the *ProcessPaymentBean* EJB as part of the same transaction to create the payment record. They both use the same JDBC datasource to commit the transaction as an atomic transaction.

A value object of `TicketDO` is passed back to the caller. The relationship tables are similar to those in Chapter 7 and are not explained here. The limitations with respect to the CMP generation of the J2SDK™ toolkit require us to model some unidirectional many-many relationships as bi-directional relationships.

The method is marked transactional and managed by the container. So the transaction is completed and the corresponding data persisted to the database only after the successful return of the `TicketDO` value object to the client.

The following method `listAvailableCabins` is used by another client program to perform a direct SQL query to provide a list of available cabins to make a reservation and book a ticket. This method uses a complex SQL query to perform joins on the relationship tables as shown below:

```
public String [] listAvailableCabins(int bedCount)
         throws IncompleteConversationalState
{
```

```
if (cruise == null)
      throw new IncompleteConversationalState();

Connection con = null;
PreparedStatement ps = null;;
ResultSet result = null;
try {
      Integer cruiseID = (Integer)cruise.getPrimaryKey();
      Integer shipID = (Integer)cruise.getShip().getPrimaryKey();
      con = getConnection();
      ps = con.prepareStatement(
            "select \"cabin\".\"id\", \"cabin\".\"name\",
            \"cabin\".\"deckLevel\" from \"CabinBeanTable\"
            \"cabin\" , \"ShipBean_cabins_CabinBean_shipTable\"
            \"ship_cabin\" "+ "where
            \"ship_cabin\".\"_ShipBean_id\" = ? and
            \"cabin\".\"bedCount\" = ?  and
            \"ship_cabin\".\"_CabinBean_id\" = \"cabin\".\"id\"
            and \"cabin\".\"id\"  NOT IN " +
                  "( SELECT \"RCL\".\"_CabinBean_id\"  FROM
                     \"ReservationBean_cabins_CabinBean_Table\"
                     AS \"RCL\" , \"ReservationBeanTable\"  AS
                     \"R\",
          \"CruiseBean_reservations_ReservationBean_cruiseTable\"
                    AS \"CR\" " + " WHERE
                    \"RCL\".\"_ReservationBean_id\"  =
                    \"R\".\"id\" AND \"R\".\"id\" =
                    \"CR\".\"_ReservationBean_id\"  AND
                    \"CR\".\"_CruiseBean_id\" = ?)");

      ps.setInt(1,shipID.intValue());
      ps.setInt(2,bedCount);
      ps.setInt(3,cruiseID.intValue());
      result = ps.executeQuery();
      Vector vect = new Vector();
      while(result.next())
      {
            StringBuffer buf = new StringBuffer();
            buf.append(result.getString(1));
            buf.append(',');
            buf.append(result.getString(2));
            buf.append(',');
            buf.append(result.getString(3));
            vect.addElement(buf.toString());
      }
      String [] returnArray = new String[vect.size()];
```

```
            vect.copyInto(returnArray);
            return returnArray;
      }
      catch (Exception e)
      {
            throw new EJBException(e);
      }
      finally
      {
            try
            {
                  if (result != null) result.close();
                  if (ps != null) ps.close();
                  if (con!= null) con.close();
            }
            catch(SQLException se)
            {
                  se.printStackTrace();
            }
      }
}
```

Notice the complexity in the SQL query and the joins it has to perform to exclude the cabins that already have reservations and to list the available cabins. This could have been easily resolved by a finder method in the *CabinEJB* to obtain the available Cabins.

Examine the ejb-jar.xml descriptor file

In this section we examine the *TravelAgentBean* section of the *ejb-jar.xml* file. The rest of the descriptor file is very much an inclusion of **Ex7.3** Entity Beans and **Ex12.1** Session Bean.

The *TravelAgentBean* is represented in a `<session>` 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.

```
<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>Container</transaction-type>
   <ejb-ref>
         <ejb-ref-name>ejb/CustomerHomeRemote</ejb-ref-name>
         <ejb-ref-type>Entity</ejb-ref-type>
         <home>com.titan.customer.CustomerHomeRemote</home>
```

```
            <remote>com.titan.customer.CustomerRemote</remote>
        </ejb-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/ShipHomeLocal</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>com.titan.ship.ShipHomeLocal</local-home>
            <local>com.titan.ship.ShipLocal</local>
            <ejb-link>ShipBean</ejb-link>
        </ejb-local-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/ReservationHomeLocal</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>
                com.titan.reservation.ReservationHomeLocal
            </local-home>
            <local>com.titan.reservation.ReservationLocal</local>
            <ejb-link>ReservationBean</ejb-link>
        </ejb-local-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/CruiseHomeLocal</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>com.titan.cruise.CruiseHomeLocal</local-home>
            <local>com.titan.cruise.CruiseLocal</local>
            <ejb-link>CruiseBean</ejb-link>
        </ejb-local-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>com.titan.cabin.CabinHomeLocal</local-home>
            <local>com.titan.cabin.CabinLocal</local>
            <ejb-link>CabinBean</ejb-link>
        </ejb-local-ref>
        <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>
    </session>
```

Notice that the `<session>` element first declares the `<ejb-ref>` elements and then defines the `<ejb-local-ref>` elements. We also define a JDBC Datasource reference for this EJB.

Notice the following description for the *CustomerBean* EJB section in the descriptor file.

```
<entity>
   <display-name>CustomerBean</display-name>
   <ejb-name>CustomerBean</ejb-name>
   <home>com.titan.customer.CustomerHomeRemote</home>
   <remote>com.titan.customer.CustomerRemote</remote>
   <local-home>com.titan.customer.CustomerHomeLocal</local-home>
   <local>com.titan.customer.CustomerLocal</local>
   <ejb-class>com.titan.customer.CustomerBean</ejb-class>
   <persistence-type>Container</persistence-type>
   <prim-key-class>java.lang.Integer</prim-key-class>
   <reentrant>False</reentrant>
   <cmp-version>2.x</cmp-version>
   <abstract-schema-name>Customer</abstract-schema-name>
   ……..
</entity>
```

CustomerBean EJB defines both the local and the remote interfaces. Remote interfaces are used by the distributed clients, and the local interfaces are used the clients within the container as well as the Entity Beans that have relationships defined as per CMP 2.0 specification for local interfaces. For further examination inspect the source code for the *CustomerBean* and *ReservationBean* EJBs.

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. Give the application a web-context of Titan12_2. Go to *http://localhost:8000/Titan12_2*. This provides a listing of client programs for this exercise.

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:

```
Client_125 Create Customer, Cabin, Ship, and Cruise objects for use
in this exercise
Calling TravelAgentBean to create sample data..
Finished TravelAgentBean to create sample data..
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 342, 343, 344
```

```
Created cruises on ShipB with IDs 345, 346, 347
Made reservation for Customer 1 on Cruise 342 for Cabin 103
Made reservation for Customer 1 on Cruise 347 for Cabin 208
Made reservation for Customer 2 on Cruise 343 for Cabin 105
Made reservation for Customer 2 on Cruise 347 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 for the primary key. Readers must use the Primary keys for the *Client_126.jsp* from the contents of the tables shown below the form for *Client_126.jsp*. *Client_125.jsp* also displays the contents of the tables as part of the output. Readers can select the required CruiseID, CabinID and ShipID for the next client program from this output.

Client_126.jsp and Client_126_action.jsp

Client_126_action.jsp is the main client program that returns the ticketing information after performing a booking for the customer. This client program calls the `bookPassage` method on the *TravelAgentBean* EJB that was described earlier. The *Client_126.jsp* provides a form for the user to enter the CustomerID, CruiseID, CabinID and price information to perform `bookPassage` operation on the *TavelAgentBean* EJB.

Click on the **Submit** button that posts the action to *Client_126_action.jsp*. *Client_126_action.jsp* then performs the complete process of obtaining references to the required EJBs and executes the required method on the *TravelAgentBean*.

```
Context jndiContext = getInitialContext();
Object obj =jndiContext.lookup(
                      "java:comp/env/TravelAgentHomeRemote" );
TravelAgentHomeRemote tahome = (TravelAgentHomeRemote)
               javax.rmi.PortableRemoteObject.narrow(obj,
               TravelAgentHomeRemote.class);

obj = jndiContext.lookup("java:comp/env/CustomerHomeRemote");
CustomerHomeRemote custhome = (CustomerHomeRemote)
               javax.rmi.PortableRemoteObject.narrow(obj,
               CustomerHomeRemote.class);

// Find a reference to the Customer for which to book a cruise
CustomerRemote cust = custhome.findByPrimaryKey(customerID);

// Start the Stateful session bean
TravelAgentRemote tagent = tahome.create(cust);

// Set the other bean parameters in agent bean
tagent.setCruiseID(cruiseID);
tagent.setCabinID(cabinID);
```

```
// Create a dummy CreditCard for this.  Really should use Customer's
card, but CreditCard bean is local
// and there is no getCreditCard() function on Customer which
returns a CreditCardDO object.
Calendar expdate = Calendar.getInstance();
expdate.set(2005,1,5);
CreditCardDO card = new
CreditCardDO("370000000000002",expdate.getTime(),"AMERICAN
EXPRESS");

// Book the passage
TicketDO ticket = tagent.bookPassage(card,price);

tagent.remove();

out.println("<H2>Result of bookPassage:</H2>");
out.println(ticket.description);
```

Providing the following information in *Client_126.jsp* runs this client program.

- CustomerID: **1**
- CruiseID**:** **340**
- CabinID: **200**
- Price: **1000.00**

Readers should look into the contents of the table. The sequence-generated IDs might be different for CruiseID. The *Client_126_action.jsp* looks up the TravelAgentHome and CustomerHome references. It then uses the finder method `findByPrimaryKey` to retrieve the Customer with ID value of 1. If the finder method fails to locate the respective *CustomerBean* EJB a `FinderException` will be thrown. The client program then creates the *TravelAgentBean* by passing the `CustomerRemote` object. The client program then invokes setters on CruiseID and CabinID. The implementation methods on the *TravelAgentBean* class perform a lookup to set the state variables to the corresponding local interfaces. Finally the client program invokes the `bookPassage` method by passing in a value object of type `CreditCardDO` and the price. Remember the *TravelAgentBean* is referenced through the remote interfaces thus requires serializable value objects as arguments and should not use Local interface for *CreditCardBean* EJB. The client then displays the information from the `TicketDO` object returned by the Stateful Session Bean.

If the references to the *Customer*, *Cruise* or *Cabin* set in the EJB are invalid the *TravelAgentBean* will throw an `IncompleteConversationalState` exception. The output of the program is shown below.

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

```
Customer ID = 1
Cruise ID = 340
Cabin ID = 200
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 Australian Highlights cruise on
ship Bohemian Rhapsody. Your accommodations include Suite 200 a 2
bed cabin on deck level 1. Total charge = 1000.0
```

Client_127.jsp and Client_127_action.jsp

This example demonstrates the execution of another method, `listAvailableCabins`, mentioned above in the examination of the *TravelAgentBean* EJB. The *Client_127.jsp* provides form fields to take CruiseID and bedcount, and upon submission of the form the *Client_127_action.jsp* provides the information with the list of available cabins for that particular cruise with the bedcounts.

The output of the program is shown below:

```
Client_127_action Results of listAvailableCabins call
Return to Client_127.jsp

Cruise ID = 344
Bed Count = 1
Starting TravelAgent Session...
Setting Cruise information in TravelAgent..
Calling listAvailableCabins()..
Result of listAvailableCabins:
109,Suite 109,1
108,Suite 108,1
107,Suite 107,1
106,Suite 106,1
105,Suite 105,1
104,Suite 104,1
103,Suite 103,1
102,Suite 102,1
101,Suite 101,1
100,Suite 100,1
```

This completes the roundup of all the major types of EJBs and their usage as specified by the EJB 2.0 specification. One additional type of EJB that is part of EJB 2.0 is Message Driven Beans, also referred as MDB for asynchronous processing. We will have an example in the next chapter.