Exercise 7:

Creating Associations Using BMP Entity EJB™

Case Study: Creating Associations Using BMP Entity EJBs

Estimated completion time: 1 hour 15 minutes.

In this exercise, you create an association between the `Customer` and `Account` BMP entity EJB™. It is quite common for a bank customer to have multiple accounts with a single bank that serve a customer's savings, checking, and credit needs. The one-to-many relationship between a customer and their accounts suggests a corresponding data model that couples some basic customer information with a listing of the customer's accounts. The data model bean used in the bank application to represent the one-to-many relationship between a customer and their accounts is illustrated in Figure 1. The class, named `CustomerWithAcctsData` contains a field named `customerData` of type `CustomerData` to store the required customer information and a `Vector` of `AccountData` objects in a field named `accounts` containing information on the customer's various accounts.

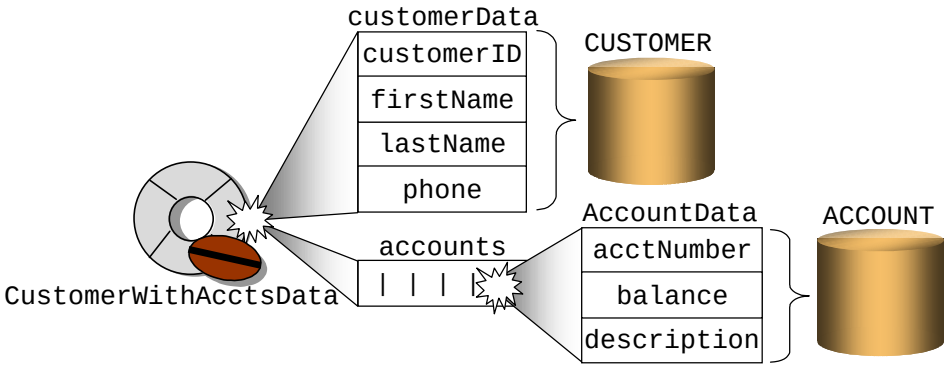

Figure 1 - The `CustomerWithAcctsData` Model Bean

At the database level, an association between database tables is done using a `JOIN` command. You can provide this functionality using EJBs in a variety of ways, two of which are discussed here.

As shown in Figure 2, a simple approach to implementing a one-to-many relationship between two tables uses a business method on a session EJB to retrieve the necessary data elements from the associated entity EJBs and construct the corresponding data object. In this example, the client application invokes the session bean's `getCustomerWithAcctsData` business method. The session bean retrieves a customer data object from the `customer` EJB and combines this with a `Vector` of the customer's accounts obtained from the `Account` EJB to construct a data model bean. The functionality of the entity EJBs is segregated such that they operate independently of each other.

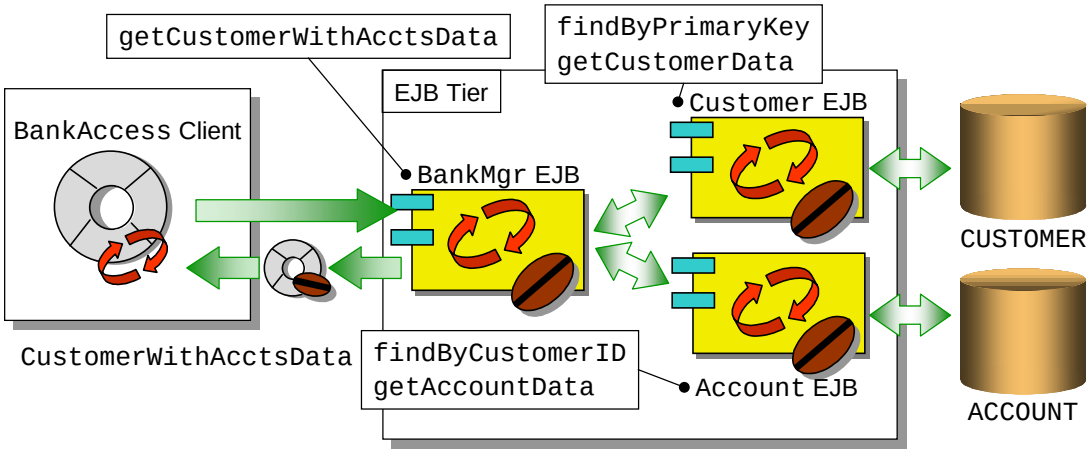

Figure 2 - Retrieving `CustomerWithAcctsData` (option 1)

Figure 3 illustrates an alternative mechanism that adds functionality to the primary object. In this case, the `Customer` EJB, which enables the retrieval of data elements from the associated `Account` EJB. The primary object uses the retrieved data elements combined with its own data to construct a data model bean. This second method is the approach implemented in this lab.

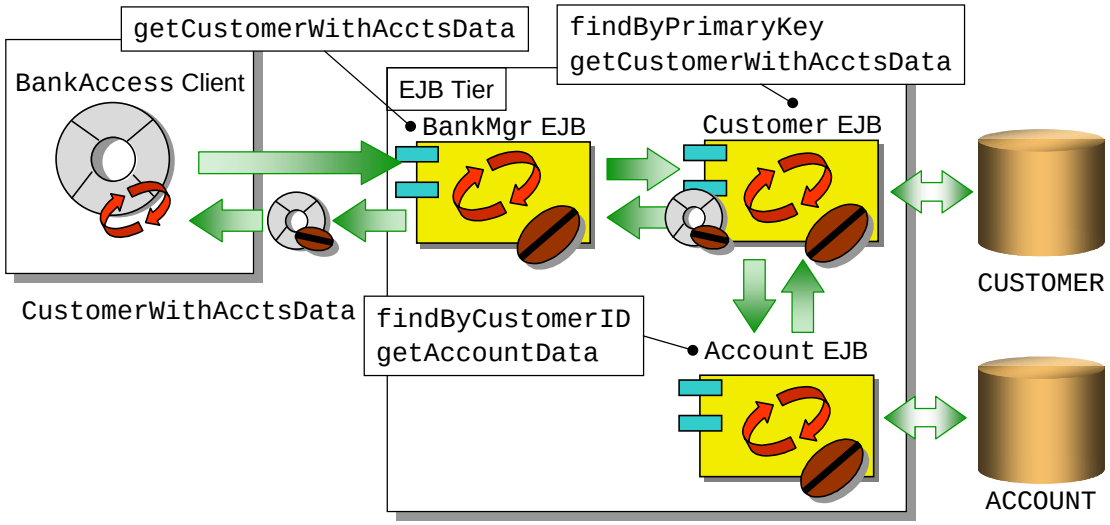

Figure 3 - Retrieving `CustomerWithAcctsData` (option 2)

There are four sections to this lab exercise. In the first section, you create the data model bean used to represent the customer with accounts relationship. In Section 2, you code selected portions of the `Customer` BMP entity EJB to interact with the `Account` EJB and construct a data model bean. In Section 3, you modify the `BankAccess` test client and `BankMgr` session EJB to invoke the new functionality on the `Customer` BMP entity EJB. Finally, in Section 4, you deploy and test the application components.

After completing this lab, you will be able to:

- Describe several ways of implementing associations between objects in a data store using BMP entity EJB's
- Describe the association between data elements in a one-to-many relationship
- Deploy an EJB that contains an EJB reference

Discussion Topics:

- What would be the results of issuing a `JOIN` command on the `J2EECUSTOMER` and `J2EEACCOUNT` tables using the field `CUSTOMERID` as the joining data element?
- What mechanism does an EJB use when it needs to find another EJB in the runtime namespace?

Important Terms and Concepts

Refer to the JDK Javadoc or the lab Appendix for information on the important classes and methods used in this exercise, including parameters and return values.

java.rmi

```
public class RemoteException extends IOException
```

java.util

```
public interface Collection
public class Vector extends AbstractList implements List, Cloneable,
                                    Serializable
   public void addElement(Object obj)
public interface Iterator
   public boolean hasNext()
```

javax.naming

```
public interface Context
public class InitialContext extends java.lang.Object implements
                                    Context
   java.lang.Object lookup(String name)
```

javax.rmi

```
public class PortableRemoteObject extends Object
   public static Object narrow(Object narrowFrom, Class narrowTo)
         throws ClassCastException
```

Setup Instructions

This lab exercise requires no special setup. Modify the existing application files for the `Customer` and `BankMgr` EJBs and the client components when completing this exercise. You can view an example of the completed exercise in the “Solution Code” section of this lab module.

Section 1

Create a Data Model Bean for a Business Class

The following steps guide you through creating a data model bean named `CustomerWithAcctsData` that represents the one-to-many relationship between the customer and account business classes. The `CustomerWithAcctsData` model bean requires a constructor and a `toString` method to display the class data using the `println` method.

1. Create a data model bean inside the `J2EEApp` package using the following criteria:
   - Name: `CustomerWithAcctsData`
   - Visibility: `public`
   - Superclass: `Object`
   - Implements: `java.io.Serializable`
2. Add the following properties to the `CustomerWithAcctsData` class. Select the check boxes in the property creation wizard to automatically create class fields and return methods.
   - Name: `customerData`; Type: `CustomerData`; Mode: Read-only; Accessor methods: `get`
   - Name: `accounts`; Type: `java.util.Vector`; Mode: Read-only; Accessor methods: `get`
3. The `CustomerWithAcctsData` class requires two constructors as shown below. Initialize the class's private fields with the values in the parameter list when implementing the parameterized constructor.
   - `CustomerWithAcctsData()`
   - `CustomerWithAcctsData(CustomerData pCustomerData, java.util.Vector pAccounts)`
4. Add a `toString` method to the new data model bean with code to display the contents of a `CustomerWithAcctsData` object.
   - Name: `toString`
   - Visibility: `public`
   - Returns: `String`
   - Parameters: None
   - Functionality: Code the method to return a `String` containing labels and values for the `customerData` and `accounts` fields. The simplest way to do this (although not the most efficient) is to use the "+" operator to concatenate the string values. Use the `CustomerData` object's `toString` method to convert the `customerData` field to a `String`. For example:

     ```
     String theString = (this.customerData).toString()+this.accounts;
     return theString;
     ```

Verify Your Work

5. Compare your work to the solution code provided for this lab exercise. Compile the `CustomerWithAcctsData` data model bean. Correct any errors or omissions before continuing.

Discussion Topics:

- What impact does deleting a row from the customer table have on the account table?

Section 2

Modify the `Customer` BMP Entity EJB

In this section, you modify the `Customer` entity EJB by adding method used to retrieve a `CustomerWithAcctsData` object. Use the "Guidelines for Creating a BMP Entity EJB" or the accompanying course material as a resource when completing this exercise.

1. Modify the `Customer` entity EJB's remote interface as follows:
    - Business methods:
        - `CustomerWithAcctsData getCustomerWithAcctsData`
2. Add a method to the `Customer` EJB bean class:
    - Name: `getCustomerWithAcctsData`
    - Visibility: `public`
    - Returns: `CustomerWithAcctsData`
    - Parameters: None
    - Functionality, code the method to:
        - Look up the `Account` entity EJB's home interface using JNDI
        - Retrieve the suite of associated accounts for the current customer using the `Account` EJB's `findByCustomerID` method
        - Construct a vector of `AccountData` objects using the results of `findByCustomerID`
        - Construct and return a `CustomerWithAcctsData` bean

Verify Your Work

3. Compare your work to the solution code provided for this lab exercise. Compile the `Customer` EJB's components. Correct any errors or omissions before continuing.

Discussion Topics:

- Why does the `Customer` EJB still use JNDI to locate the `Account` EJB's home interface in the runtime namespace even though both the `Customer` and `Account` EJBs are running in the same J2EE execution environment?

Section 3

Modify the BankMgr Session EJB

The BankMgr session EJB provides the mechanism for accessing the functionality provided by the Customer entity EJB. In this section, you add a method to the BankMgr session bean to retrieve a CustomerWithAcctsData data model object for a specified customer. The code for this method is similar to the code for the getCustomerData method created in an earlier exercise.

1. Modify the BankMgr session EJB's remote interface as follows:
    - Business methods:
        - `CustomerWithAcctsData getCustomerWithAcctsData(String customerID)`
2. Perform the following modifications to the BankMgr EJB's bean class:
    - Add a method to the BankMgr bean class to implement the getCustomerWithAcctsData method added to the EJB's remote interface above:
        - Name: getCustomerWithAcctsData
        - Visibility: public
        - Returns: CustomerWithAcctsData
        - Parameters:
            - String customerID
        - Functionality, code the method to:
            - Locate a row in the customer table associated with the parameter customerID using the Customer EJB's findByPrimaryKey method
            - Return a CustomerWithAcctsData data model object for the customer by invoking the Customer EJB's getCustomerWithAcctsData method for the specified customer

Modify the BankAccess Test Client

The BankAccess class serves as the test client for this exercise. The BankAccess class uses the methods available on the BankMgr session EJB to access the functionality provided by the Customer EJB. For this lab, the BankAccess client retrieves a CustomerWithAcctsData object for a specified customer using the BankMgr EJB's getCustomerWithAcctsData method. The following steps guide you through modifying the BankAccess class's main method to invoke the BankMgr EJB's getCustomerWithAcctsData method.

3. Modify the BankAccess's main method to call the BankMgr EJB's getCustomerWithAcctsData method, and retrieve customer information for one or more bank customers as follows:
    - Create a new BankMgrEJB instance
    - Call the BankMgrEJB's getCustomerWithAcctsData method to retrieve a CustomerWithAcctsData object for at least one specified customer and print the results
    - Call BankMgrEJB's remove method

Compile

4. Build all of the files in the `J2EEApp` package. Correct any errors before continuing with this exercise.

Discussion Topics:

- Why doesn't the `BankMgr` session bean invoke the `remove` method on the `Customer` entity EJB when it is through using it?

Section 4

Deploy and Test

In this section, you deploy and test the application EJBs. Consult the "Assembly and Deployment Guidelines" for information on deploying EJB's into the J2EE container used for this course. Use the "IDE Configuration Guidelines" to assist you in modifying the IDE runtime configuration to access components running in the J2EE container.

1. Using the J2EE assembly/deployment tool provided for this course, assemble and deploy the `Customer` entity EJB into the J2EE runtime environment. Use "customerServer", "bankmgrServer", "accountServer", and "jdbc/BankMgrDB" as the JNDI name for the `Customer`, `BankMgr`, and `Account` EJBs and the database factory resource, respectively. To assemble and deploy the application components using the J2EE Reference Implementation deployment tool you must:
   - Start the J2EE runtime, cloudscape database, and deployment tool.
   - Use the deployment tool to:
     - Undeploy the `BankApp` application.
     - Update the application files for `BankApp`.
     - Use the deployment tool to add the .class file for the `CustomerWithAcctsData` data model bean class to the `BankJAR` file. This is a helper class for the `CustomerBean`.
     - Update the method transaction attributes for the `CustomerBean` and the `BankMgrBean`. Set the transaction attribute for `getCustomerWithAcctsData` to "required".
     - Deploy the application:
       - Select the option to Return Client Jar, and store the client `.jar` file in the *<J2EEAppHome>* directory.
       - Use "bankmgrServer", "customerServer", and "accountServer" as the JNDI names for the `BankMgr` session and `Customer` and `Account` entity EJBs, respectively.
       - Map the resource factory reference (`jdbc/BankMgrDB`) for the `BankMgr` and `Customer` EJBs to the name of the data source as registered with the J2EE runtime; for example, use "jdbc/Cloudscape" if you are using the Cloudscape database supplied with the J2EE Reference Implementation.
2. Configure the test client with the path to the client `.jar` file, and execute the test client. View the results in the IDE's run console. Output generated by `println` statements in the EJB can be seen in the command window used to start the J2EE runtime system.

Solution Code: Section 1

```
// CustomerWithAcctsData.java

package J2EEApp;

import java.beans.*;

public class CustomerWithAcctsData
   extends Object
   implements java.io.Serializable {

   private CustomerData customerData;
   private java.util.Vector accounts;

   public CustomerWithAcctsData() { }

   public CustomerWithAcctsData(CustomerData pCustomerData,
                                java.util.Vector pAccounts) {
      System.out.println("Entering new CustomerWithAcctsData");
      this.customerData = pCustomerData;
      this.accounts = pAccounts;
      System.out.println("Leaving new CustomerWithAcctsData");
   }

   public java.lang.String toString() {
      String theString =
         (this.customerData).toString() + this.accounts;
      return theString;
   }

   public CustomerData getCustomerData() { return this.customerData; }

   public java.util.Vector getAccounts() { return this.accounts; }
}

************************************************************
```

Solution Code: Section 2

```
// Customer.java

package J2EEApp;

import java.rmi.RemoteException;
import javax.ejb.*;

public interface Customer extends EJBObject {

   public CustomerData getCustomerData()
      throws RemoteException;

   public CustomerWithAcctsData getCustomerWithAcctsData()
      throws RemoteException;
}

************************************************************

// CustomerEJB.java

package J2EEApp;

import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;

public class CustomerEJB implements EntityBean {

   private String customerID;
   private String firstName;
   private String lastName;
   private String address;
   private String phone;

   private EntityContext context;

   public CustomerEJB() {}

   public CustomerWithAcctsData getCustomerWithAcctsData() {
      System.out.println
         ("Entering CustomerEJB.getCustomerWithAcctsData()");
      java.util.Vector accounts = new java.util.Vector();
      try {
         Context c = new InitialContext();
         Object result = c.lookup("accountServer");
         AccountHome myAH =
            (AccountHome)javax.rmi.PortableRemoteObject.
               narrow(result,AccountHome.class);
         java.util.Collection acctList =
            myAH.findByCustomerID(this.customerID);
         if (acctList == null) {
            accounts = null;
         }
         else {
            java.util.Iterator ali = acctList.iterator();
```

```
            while ( ali.hasNext() ) {
               accounts.addElement
                  (((Account)ali.next()).getAccountData());
            }
         }
      } catch (Exception ex) {
         throw new EJBException
            ("CustomerEJB.getCustomerWithAcctsData " +
               ex.getMessage());
      }
      System.out.println
         ("Leaving CustomerEJB.getCustomerWithAcctsData()");
      return (new CustomerWithAcctsData
               (this.getCustomerData(), accounts));
   }

   public void ejbRemove() throws RemoveException {}

   public void setEntityContext(EntityContext ec) { ... }

   public void unsetEntityContext() { ... }

   public void ejbActivate() { ... }

   public void ejbPassivate() { ... }

   public void ejbLoad() { ... }

   private void loadRow() throws SQLException { ... }

   public void ejbStore() { ... }

   private void storeRow() throws SQLException { ... }

   public String ejbFindByPrimaryKey (String key)
         throws FinderException { ... }

   private boolean selectByPrimaryKey(String key)
         throws SQLException { ... }

   // business methods
   public CustomerData getCustomerData() { ... }

   // utility method
   private Connection initDBConnection() throws SQLException { ... }
}

************************************************************
```

Solution Code: Section 3

```
// BankMgr.java

package J2EEApp;

import java.rmi.RemoteException;
import javax.ejb.*;

public interface BankMgr extends EJBObject {

   public String loginUser(String pUsername, String pPassword)
      throws RemoteException;

   public CustomerData getCustomerData(String customerID)
      throws RemoteException;

   public java.util.Vector getAllCustomers()
      throws RemoteException;

   public CustomerWithAcctsData
      getCustomerWithAcctsData(String customerID)
         throws RemoteException;
}

************************************************************

// BankMgrEJB.java

package J2EEApp;

import java.rmi.RemoteException;

import javax.ejb.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class BankMgrEJB implements SessionBean {

   private javax.sql.DataSource jdbcFactory;
   private CustomerHome myCustomerHome;

   public CustomerWithAcctsData
      getCustomerWithAcctsData(String customerID) {
      System.out.println
         ("Entering BankMgrEJB.getCustomerWithAcctsData()");
      CustomerWithAcctsData myCWAD = null;
      try {
         Customer myCustomer =
            myCustomerHome.findByPrimaryKey(customerID);
         if (myCustomer != null) {
            myCWAD = myCustomer.getCustomerWithAcctsData();
            System.out.println("CustomerWithAcctsData is: " + myCWAD);
         }
      }
      catch (Exception e) {
```

```
            System.out.println
               ("Error in BankMgrEJB.getCustomerWithAcctsData(): " + e);
         }
         System.out.println
            ("Leaving BankMgrEJB.getCustomerWithAcctsData()");
         return myCWAD;
      }

      public java.util.Vector getAllCustomers() { ... }

      public BankMgrEJB() {  }

      public void ejbCreate() { ... }

      public void setSessionContext(SessionContext sc) { }

      public void ejbRemove() { }

      public void ejbActivate() { }

      public void ejbPassivate() { }

      public String loginUser(String pUsername, String pPassword) { ... }

      public CustomerData getCustomerData(String customerID) { ... }
}

************************************************************

// BankAccess.java

package J2EEApp;

import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class BankAccess extends Object {

      private BankMgrHome myBankMgrHome;
      private AccountHome myAccountHome;

      public BankAccess() {}

      public static void main (String args[]) {
         BankAccess ba = new BankAccess();
         try {
            ba.init();
            String myID = null;
            BankMgr myBankMgr = ba.myBankMgrHome.create();
            myID = myBankMgr.loginUser("Al", "password");
            System.out.println("User ID for user Al is: " + myID);
            CustomerData myCD = myBankMgr.getCustomerData(myID);
            System.out.println("myCD is: " + myCD);
            CustomerWithAcctsData myCWAD =
               myBankMgr.getCustomerWithAcctsData(myID);
            System.out.println("myCWAD is: " + myCWAD);
            myBankMgr.remove();
         }
```

```
      catch (Exception e) {
         System.out.println("Error in BankAccess.main(): " + e);
      }
   }

   public void init() { ... }

   private void initMyBankMgrHome() { ... }
}
```

Answers for Discussion Topics:

- What would be the results of issuing a `JOIN` command on the `J2EECUSTOMER` and `J2EEACCOUNT` tables using the field `CUSTOMERID` as the joining data element?

  Answer: Executing a `JOIN` command produces a table or view that represents the union of the `JOIN`'d tables. It would include the columns from each table as specified in the `JOIN` command.

- What mechanism does an EJB use when it needs to find another EJB in the runtime namespace?
- **Answer: JNDI**
- What impact does deleting a row from the customer table have on the account table?

  Answer: It potentially creates orphaned accounts. Because it makes no sense to have accounts without a corresponding customer, the application requires business logic to prevent deleteing a customer as long as they have associated accounts. The system can either delete associated accounts when a customer is deleted or (more desirable) require that all associated accounts are deleted before enabling customer deletions.

- Why does the `Customer` EJB still use JNDI to locate the `Account` EJB's home interface in the runtime namespace even though both the `Customer` and `Account` EJBs are running in the same J2EE execution environment?

  Answer: The actual locations of an enterprise bean and EJB container are, in general, transparent to the client using the EJB. A client's JNDI namespace can be configured to include the home interfaces of enterprise beans installed in multiple EJB containers located on multiple machines in a network. A client always uses JNDI to locate an EJB's home interface.

- Why doesn't the `BankMgr` session bean invoke the `remove` method on the `Customer` entity EJB when it is through using it?

  Answer: Invoking the `remove` method on an entity EJB deletes the corresponding row from the data store.

Filename:	lab07.doc
Directory:	H:\FJ310_revC_0301\BOOK\EXERCISE\Exercise_Originals
Template:	C:\Program Files\Microsoft Office\Templates\Normal.dot
Title:	Case Study: The New User Interface
Subject:	
Author:	jdibble
Keywords:	
Comments:	
Creation Date:	04/24/01 10:16 AM
Change Number:	2
Last Saved On:	04/24/01 10:16 AM
Last Saved By:	Nancy Clarke
Total Editing Time:	0 Minutes
Last Printed On:	04/24/01 12:38 PM

As of Last Complete Printing

Number of Pages:	16
Number of Words:	2,951 (approx.)
Number of Characters:	16,823 (approx.)