Exercise 3:

JDBC™ and Resource Factories

Case Study: JDBC and Resource Factories

Estimated completion time: 1 hour 15 minutes.

In this exercise, you create a stateless session EJB to validate user logins. The users in this example are customers logging in to a banking application using the `BankMgr` (Bank Manager) session EJB to review their accounts. The `BankMgr` session EJB uses a resource factory reference to access the `J2EEAPPUSER` database table containing user names, passwords, and associated user ID codes.

The `J2EEAPPUSER` database table has the following structure:

```
CREATE TABLE J2EEAPPUSER(
      USERNAME VARCHAR(20) CONSTRAINT PK_J2EEAPPUSER PRIMARY KEY,
      PASSWORD VARCHAR(20),
      USERID   VARCHAR(6));
```

The validation method on the session EJB is named `loginUser`. The `loginUser` method takes two `String` parameters, `pUsername` and `pPassword`, and returns the `userid` for the validated user. The banking application locates customer information and account data for the current customer based on the customer ID code.

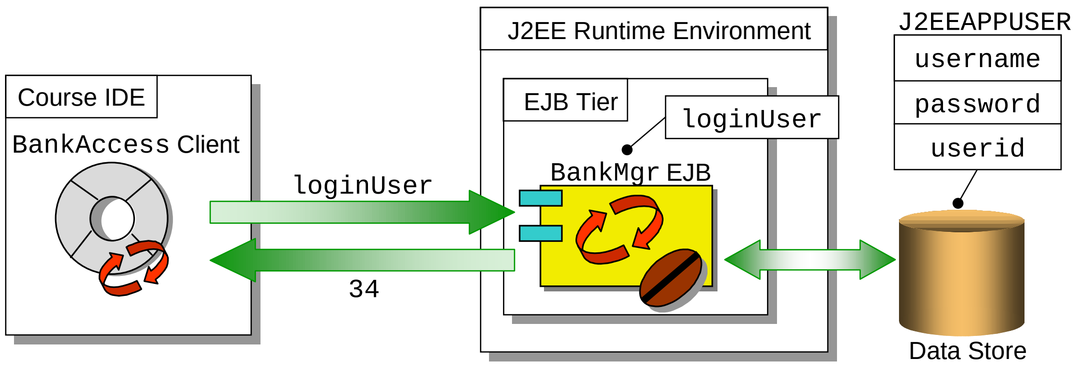

Figure 1 - Logging On Using the BankMgr Session EJB

There are three sections to this lab exercise. In the first section, you code the home and remote interfaces and create a bean class for the `BankMgr` session EJB. The bean class implements the EJB's business and life cycle methods. In Section 2, you create a test client to access the `BankMgr` session EJB. Finally, in Section 3, you deploy the `BankMgr` session EJB and test your work.

After completing this lab, you will be able to:

- Describe the function of a resource factory
- List the various Java language interfaces, classes, and methods associated with connecting to and operating on a data store
- Create a method to access a database table utilizing SQL and a `Connection` object
- Describe how to retrieve data elements from a `ResultSet` returned by the `executeQuery` method

Discussion Topics:

- What is a resource factory? Give an example of a Java class or interface that serves as a resource factory for database connections.
- What is the difference between classes that implement the `Statement` and `PreparedStatement` interfaces?
- What must be done before exiting a method in which a `Connection` object is instantiated?
- What happens to a `ResultSet` when the statement that generated it is closed?

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.sql
   public interface Statement
      public ResultSet executeQuery(String sql) throws SQLException
      public void close() throws SQLException
   public interface ResultSet
      public boolean next() throws SQLException
      public void close() throws SQLException
      double getDouble(int columnIndex) or double getDouble(String
                                        columnName)
      int getInt(int columnIndex) or int getInt(String columnName)
      String getString(int columnIndex) or String getString(String
                                        columnName)
   public interface Connection
      public Statement createStatement() throws SQLException
      public PreparedStatement prepareStatement(String sql) throws
                                        SQLException
      public void close() throws SQLException
javax.naming
   public interface Context
   public class InitialContext extends java.lang.Object implements
                                        Context
      java.lang.Object lookup(String name)
javax.sql
   public interface DataSource
      java.sql.Connection getConnection() throws java.sql.SQLException
```

Setup Instructions

There is no special setup required for this exercise. The `BankAccess` class created in a previous exercise serves as the basis for the test client used in this lab. Use the “Guidelines for Creating a Session EJB” or the accompanying course material as a resource when completing this exercise. You can view an example of the completed exercise in the “Solution Code” section of this lab module.

Section 1

Define the Session EJB's Home and Remote Interface

Use the "Guidelines for Creating a Session EJB" or the accompanying course material as a resource when completing this exercise.

1. Create a home interface for the `BankMgr` session EJB in the existing `J2EEApp` package.
   - Name: `BankMgrHome`
   - Creator methods:
     - `create`
2. Create a remote interface for the `BankMgr` EJB in the `J2EEApp` package.
   - Name: `BankMgr`
   - Business methods:
     - `String loginUser(String pUsername, String pPassword)`– Returns the user ID for a banking customer after validating the customer's user name and password against the `J2EEAPPUSER` database table.

Define the Session EJB's Bean Class

3. In the `J2EEApp` package, create a skeleton for the `BankMgr` EJB bean class containing methods that implement the corresponding method stubs on the home and remote interfaces. Include stubs for any additional methods required by the container and implemented in the bean class but are not defined on the bean's home or remote interfaces. Do not code any of the bean class's method bodies at this time.
   - Name: `BankMgrEJB`
   - Imports:
     - `javax.naming.*`; `javax.sql.*`; `java.sql.*`

Implement the Bean Class Methods

4. Modify `BankMgrEJB` as follows:
   - Add a field to store the JDBC™ resource factory reference:
     - Name: `jdbcFactory`
     - Type: `javax.sql.DataSource`
     - Visibility: `private`
   - Modify the `ejbCreate` method to use JNDI to locate the resource factory reference associated with the JNDI name "BankMgrDB", and store it in `jdbcFactory.`
5. Code the EJB's `loginUser` method to:
   - Obtain a connection to the database using `jdbcFactory`.
   - Create a SQL select statement that searches the `J2EEAPPUSER` table for a row where `USERNAME = pUsername` and `PASSWORD = pPassword` and returns the value from the `USERID` column.

- Catch and handle any exceptions thrown by the process. Note: In an actual application, it may be desirable to add a user-defined exception that is thrown when a logon attempt fails.
- Return the user ID value to the calling method.

Compile

6. Build the files contained in the `J2EEApp` package. The entire package should compile without errors. Fix any errors before continuing with this exercise. Verify your work against the solution code provided for this module, if necessary.

Discussion Topics:

- What is the function of the `jdbcFactory` field on the `BankMgr` EJB?
- Why does the solution set for this exercise use a `Statement` instead of a `PreparedStatement` to execute the SQL commands required to access the `J2EEAPPUSER` table and validate user login requests?

Section 2

Modify the `BankAccess` Client

The `BankAccess` class created in a previous exercise serves as the test client for banking application components including the `BankMgr` session bean. To access `BankMgrEJB.loginUser`, you modify the `BankAccess` class by adding a `loginUser` method to forward login requests to the `BankMgr` EJB and then updating `BankAccess.main` to call `BankAccess.loginUser` and test login functionality. See "Banking Application Data Elements" in the lab appendix for a listing of users contained in the banking application's database tables.

1. Add the following Java package to the import list for the `BankAccess` class:
   - `javax.naming.*`
2. Add a field to the `BankAccess` bean used to store the `BankMgr` EJB's home interface reference.
   - Name: `myBankMgrHome`
   - Visibility: `private`
   - Type: `BankMgrHome`
3. Add a method to the `BankAccess` bean.
   - Name: `initMyBankMgrHome`
   - Return Type: `void`
   - Visibility: `private`
   - Parameters: None
     - Functionality: Code the method to look up the `BankMgr` EJB's home interface using the JNDI name "bankmgrServer" and store the home interface reference to the field `myBankMgrHome`.
4. Code the `BankAccess`'s `main` method to:
   - Instantiate a `BankAccess` object
   - Initialize `myBankMgrHome`
   - Validate several users using the `BankMgr` EJB's `loginUser` method and print the results to the output screen

Compile

5. Compile the `BankAccess` class. Correct any errors before continuing with this exercise.

Discussion Topics:

- What is the function of the private field `myBankMgrHome` created in the `BankAccess` class?

Section 3

Deploy and Test

In this section, you deploy and test the `BankMgr` session EJB. 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 `BankMgr` session EJB into the J2EE runtime environment. Use “bankmgrServer” as the JNDI name for the session EJB and “jdbc/BankMgrDB” as the JNDI name for the database factory resource. To assemble and deploy the session bean using the J2EE reference implementation deployment tool you must:
    - Start the J2EE runtime, Cloudscape database, and deployment tool
    - Use the deployment tool to:
        - Create a new application named `BankApp` and store the `.ear` file in the *`<J2EEAppHome>`* directory.
        - Add the `BankMgr` session bean to the application:
            - **New Enterprise Bean Wizard - EJB JAR**
                - Enterprise Bean Will Go In: `BankApp`
                - JAR Display Name: `BankJAR`
                - Contents - Add:
                    - Choose the system workspace as the root directory. For example, `C:\`*`<IDEHome>`*`\Development`.
                    - Open the `J2EEApp` package.
                    - Select and add `BankMgr.class`, `BankMgrEJB.class`, and `BankMgrHome.class` from the `J2EEApp` package.
            - **New Enterprise Bean Wizard - General**
                - Enterprise Bean Class: `J2EEApp.BankMgrEJB`
                - Home Interface: `J2EEApp.BankMgrHome`
                - Remote Interface`: J2EEApp.BankMgr`
                - Display Name: `BankMgrBean`
                - Bean Type: Session/Stateless
            - **New Enterprise Bean Wizard - Resource References**
                - Resource Factories Referenced in Code:
                    - `jdbc/BankMgrDB`; `javax.sql.DataSource`; Container
            - **New Enterprise Bean Wizard - Transaction Management**
                - Container-Managed Transactions: Checked
                - `loginUser` Transaction Type: Required

- Deploy the application:
  - Select the option to Return Client Jar, and store the client `.jar` file in the *`<J2EEAppHome>`* directory.
  - **Deploy BankApp - JNDI Names**
    - Use “bankmgrServer” as the JNDI name for the `BankMgr` session bean.
    - Map the resource factory reference 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 `BankAccess` client with the path to the client `.jar` file, and and execute the `BankAccess` client. View the results in the IDE’s run console. Output generated by `println` statements in an EJB deployed to the J2EE environment can be seen in the command window used to start the J2EE runtime system.

Discussion Topics:

- Describe the two types of transaction management options available when deploying a session EJB.
- What is the purpose of the client `.jar` file generated when deploying an application into the J2EE runtime environment?

Solution Code: Section 1

```
// BankMgrHome.java

package J2EEApp;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface BankMgrHome extends EJBHome {
   BankMgr create() throws RemoteException, CreateException;
}

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

// BankMgr.java

package J2EEApp;

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

public interface BankMgr extends EJBObject {
   public String loginUser(String pUsername, String pPassword)
      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;

   public BankMgrEJB() {}

   public void ejbCreate() {
      System.out.println("Entering BankMgrEJB.ejbCreate()");
      Context c = null;
      if (this.jdbcFactory == null) {
         try {
            c = new InitialContext();
            this.jdbcFactory = (javax.sql.DataSource)
               c.lookup("java:comp/env/jdbc/BankMgrDB");
         } catch (Exception e){ System.out.println("Error: " + e); }
      }
      System.out.println("Leaving BankMgrEJB.ejbCreate()");
   }

   public String loginUser(String pUsername, String pPassword) {
      System.out.println("Entering BankMgrEJB.loginUser()");
      String theID = null;
      try {
         Connection conn = this.jdbcFactory.getConnection();
         Statement  stmt = conn.createStatement();
         String sqlString =
            "select UserID from J2EEAPPUSER where Username = '" +
               pUsername + "' and Password = '" + pPassword + "'";
         System.out.println(sqlString);
         ResultSet rs = stmt.executeQuery(sqlString);
         if ( rs.next() ) {
            theID = rs.getString("UserID");
            System.out.println("userID is: " + theID);
         }
         rs.close();
         stmt.close();
         conn.close();
      }
      catch ( Exception e ) { System.out.println("Error: " + e); }
      System.out.println("Leaving BankMgrEJB.loginUser()");
      return theID;
   }

   public void setSessionContext(SessionContext sc) {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
}
```

Solution Code: Section 2

```
// BankAccess.java

package J2EEApp;

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

public class BankAccess extends Object {

   private BankMgrHome myBankMgrHome;

   /** Creates new BankAccess */
   public BankAccess() {}

   public static void main (String args[]) {
      System.out.println("Entering BankAccess.main()");
      BankAccess ba = new BankAccess();
      String myID = null;
      try {
         ba.init();
         BankMgr myBankMgr = ba.myBankMgrHome.create();
         myID = myBankMgr.loginUser("Al", "password");
         System.out.println("User ID for user Al is: " + myID);
         myID = myBankMgr.loginUser("Anita", "password");
         System.out.println ("User ID for user Anita is: " + myID);
         myID = myBankMgr.loginUser("Bryan", "password");
         System.out.println("User ID for user Bryan is: " + myID);
         myBankMgr.remove();
      }
      catch (Exception e) { System.out.println("Error: " + e); }
   }

   public void init() {
      System.out.println("Entering BankAccess.init()");
      try {
         if (myBankMgrHome == null) initMyBankMgrHome();
      }
      catch (Exception e) {
         System.out.println("Error in BankAccess.init(): " + e);
      }
      System.out.println("Leaving BankAccess.init()");
   }

   private void initMyBankMgrHome() {
      System.out.println("Entering BankAccess.initMyBankMgrHome()");
      try {
         Context c = new InitialContext();
         Object result = c.lookup("bankmgrServer");
         myBankMgrHome =
            (BankMgrHome)javax.rmi.PortableRemoteObject.
               narrow(result,BankMgrHome.class);
      }
      catch (Exception e) { System.out.println(e); }
      System.out.println("Leaving BankAccess.initMyBankMgrHome()");
   }
}
```

Answers for Discussion Topics:

- What is a resource factory? Give an example of a Java class or interface that serves as a resource factory for database connections.

  Answer: A resource factory is an object used to create resources. For example, an object that implements the `javax.sql.DataSource` interface is a resource factory for JDBC connections.

- What is the difference between classes that implement the `Statement` and `PreparedStatement` interfaces?

  Answer: Classes that implement the `PreparedStatement` interface create an object that contains a pre-compiled SQL statement. A `PreparedStatement` is used to efficiently execute a SQL statement multiple times. SQL statements without parameters, also referred to as static statements, are normally executed using objects that implement the `Statement` interface. If the same SQL statement is executed many times, it is more efficient to use a `PreparedStatement`.

- What must be done before exiting a method in which a `Connection` object was instantiated?

  Answer: The `Connection` should be closed to free up the resources allocated when it was created.

- What happens to a `ResultSet` when the statement that generated it is closed?

  Answer: A `ResultSet` is automatically closed by the `Statement` that generated it when that `Statement` is closed, re-executed, or is used to retrieve the next result from a sequence of multiple results.

- What is the function of the `jdbcFactory` field on the `BankMgr` EJB?

  Answer: The field is used to store a local reference to the JDBC resource factory used to connect to the application database tables. It is more efficient to look up this reference once when the EJB is created and store it to a local variable for later use than to look it up every time the EJB's `loginUser` method is invoked. This reference will remain with the EJB throughout the bean's life cycle.

- Why does the solution set for this exercise use a `Statement` instead of a `PreparedStatement` to execute the SQL commands required to access the `J2EEAPPUSER` table and validate user login requests?

  Answer: Parameterized SQL commands are normally executed using pre-compiled `PreparedStatements`. The primary advantage to using a `PreparedStatement` is the benefit derived from pre-compiling a SQL statement that will be executed multiple times. Using a `PreparedStatement` to execute the SQL commands required to validate a user login would be appropriate and most likely make the operation more efficient.

- What is the function of the private field `myBankMgrHome` created in the `BankAccess` class?

  Answer: The field `myBankMgrHome` is used to store a reference to the `BankMgr` EJB's home interface. The field is initialized by the class's `initMyBankMgrHome` method immediately after a class instance is instantiated by the `main` method. Storing the reference to a private variable visible to the class methods eliminates the neccessity for each class method to look up the reference if and when it needs to access the `BankMgr` EJB.

- Describe the two types of transaction management options available when deploying a session EJB.

  Answer: When deploying a session bean you have a choice of bean-managed or container-managed transaction management. When choosing container-managed transactions, the container is responsible for starting, committing, and rolling back transactions based on some configuration settings chosen during deployment. In most cases, using container-managed transactions for session bean methods is desirable. Bean-managed transactions are used when the transaction management options available from the container do not satisfy the application requirements.

- What is the purpose of the client `.jar` file generated when deploying an application into the J2EE runtime environment?

  Answer: The client `.jar` file contains the stub classes for the application components deployed to the J2EE runtime environment that enable clients to connect with J2EE components remotely.

Filename:	lab03.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 10:02 AM
Change Number:	3
Last Saved On:	04/24/01 10:06 AM
Last Saved By:	Nancy Clarke
Total Editing Time:	2 Minutes
Last Printed On:	04/24/01 12:36 PM
As of Last Complete Printing	
Number of Pages:	14
Number of Words:	2,641 (approx.)
Number of Characters:	15,059 (approx.)