UNISYS

ClearPath Application Development Solutions

ClearPath OS 2200 IDE *for Eclipse*™ Application Development Guide for Java EE Projects

ClearPath OS 2200 Release 13.1

February 2012 3839 3831–002

Contents

Section 4. Loading the Relational Database Server Database

Section 5. Running Java Applications on the OS 2200 System

Section 6. Creating an Enterprise Application Development Model

Section 9. Creating Container-Managed Persistence Entity Beans

Section 10. Creating Web Client Servlets

Section 11. Creating Web Client JavaServer Pages

Section 12. Creating Web Services

Section 13. Creating Message-Driven Beans

Figures

Tables

Section 1
Getting Started

The Eclipse platform (www.eclipse.org/platform) is one of the most useful integrated development tools available to a developer. Its built-in functionality is generic, open, and extensible by plug-ins. With the appropriate plug-ins, it can be used to develop applications in Java and many other programming languages. Plug-ins are available that support development activities ranging from design, development, and debugging to deployment of both simple and multitiered applications.

This guide is based on the J2EE™ tutorial that is owned and copyrighted by TUSC Computer Systems Pty Ltd. (www.tusc.com.au).

Purpose

This guide describes how to build Java Platform, Enterprise Edition (Java EE) components, using the ClearPath OS 2200 IDE *for Eclipse™* package as the Integrated Development Environment (IDE). It contains instructions for building and testing each component, along with completed examples.

Audience

This guide is for Java developers who write Java EE applications using Web Tools and deploying to a JBoss Application Server™ (JBoss AS).

Prerequisites

This guide assumes that you are familiar with the OS 2200 IDE for Eclipse environment and application development concepts. You should have a working knowledge of Java technology, XML, and Java EE technology, along with some exposure to SQL, JDBC concepts, and attribute-oriented programming (XDoclet).

The exercises in this guide assume that you installed all the components that are identified in the *ClearPath OS 2200 IDE for Eclipse™ Installation Guide*.

Unzip the file eclipse-2200-appl-dev-guide.zip to get the following subfolders and files, besides this guide:

- Database Scripts. Contains scripts for populating the databases for the examples in this guide, including the Hypersonic and Relational Database Server databases.
- Database Drivers. Contains drivers for viewing the Relational Database Server database with Data Source Explorer.
- Examples. Contains additional subfolders with complete Eclipse IDE workspaces for each section in this guide. Examples are numbered to match the sections to which they refer. To load these workspaces without errors, you might have to change project class path entries to match your Eclipse IDE environment.

Documentation Updates

This document contains all the information that was available at the time of publication. Changes identified after release of this document are included in problem list entry (PLE) 18837421. To obtain a copy of the PLE, contact your Unisys representative or access the current PLE from the Unisys Product Support Web site:

http://www.support.unisys.com/all/ple/18837421

Note: *If you are not logged into the Product Support site, you will be asked to do so.*

Notation Conventions

This guide uses the following notation conventions:

Convention	Description
Italic font	Used for names of variables to which values must be assigned.
Bold font	Used to • Emphasize items such as the names of objects on windows and dialog boxes, key names, and commands that are identified in text. • Identify code that users must replace in some examples.
`Monospace font`	Used for examples and system output, such as prompt signs and responses to commands.
[ ]	Used to enclose optional fields or subfields.
>	Represents command line prompt.

Accessing Menu Options

Throughout this guide, the instructions direct you to right-click selected objects in the Java EE project window to reduce the number of mouse clicks for accessing menu options. You can also access options using the main menu.

Right-click refers to the right button on a right-handed mouse. If you are using a left-handed mouse, use the left button of the mouse.

You can access menu options from other perspectives, such as the Java perspective. However, the menu items documented in this guide are different from those of other perspectives. Typically, you must navigate through more menu options if you are not in the Java EE perspective.

1.1. Office Supply Store Overview

The Office Supply Store is a Java project that provides the examples in this guide. As you learn how to create the various Java EE components, you build them into the project to perform specific functions. As each component is developed and integrated, the project becomes more complex and useful.

The Office Supply Store can serve as a prototype as you develop projects to suit your own business needs.

1.1.1. Database Schema

The procedures in the components use a database schema, called OSupplyStore, that consists of the following tables:

- UserAccess authenticates all customers, suppliers, and managers for online access to Office Supply Store applications.
- StoreCustomer records details of customers who bought an item at least once.
- StoreManager records details of managers who run the Office Supply Store; currently, this table contains only one manager.
- StoreSupplier records details of suppliers who sell items to the Office Supply Store in response to requests sent to these suppliers from an Office Supply Store manager as the need arises.
- StoreInventory maintains an inventory of store items.

Table 1–1 describes the database schema.

Table 1–1. OSupplyStore Database Schema

Table Name	Column Name	Size	Other
UserAccess	UserName	VARCHAR(20)	
	Password	VARCHAR(8)	
	StoreAccessID	VARCHAR(8)	NOT NULL PRIMARY KEY
StoreCustomer	CustomerID	VARCHAR(8)	NOT NULL PRIMARY KEY
	LName	VARCHAR(18)	
	FName	VARCHAR(18)	
	StreetAddress	VARCHAR(28)	
	City	VARCHAR(28)	
	State	VARCHAR(2)	
	ZipCode	VARCHAR(5)	
	Notes	VARCHAR(40)	
StoreManager	ManagerID	VARCHAR(8)	NOT NULL PRIMARY KEY
	LName	VARCHAR(18)	
	FName	VARCHAR(18)	
	StreetAddress	VARCHAR(28)	
	City	VARCHAR(28)	
	State	VARCHAR(2)	
	ZipCode	VARCHAR(5)	
	Notes	VARCHAR(40)	
StoreSupplier	SupplierID	VARCHAR(8)	NOT NULL PRIMARY KEY
	CompanyName	VARCHAR(18)	
	StreetAddress	VARCHAR(28)	
	City	VARCHAR(28)	
	State	VARCHAR(2)	
	ZipCode	VARCHAR(5)	
	Notes	VARCHAR(40)	
StoreInventory	ItemID	VARCHAR(10)	NOT NULL PRIMARY KEY
	SupplierID	VARCHAR(8)	
	Description	VARCHAR(40)	
	QtyOnHand	INTEGER	
	Price	DECIMAL(12,2)	

1.1.2. J2EE Components in Office Supply Store

To access data from the database and perform business operations, the Office Supply Store example uses various Java 2 Platform, Enterprise Edition (J2EE) components, including session, entity, and message-driven Enterprise JavaBeans (EJB) components, along with Web clients that use servlets and JavaServer Pages (JSP).

Figure 1–1 illustrates the Office Supply Store application architecture. Pieces of this architecture are illustrated in each section that describes a component.

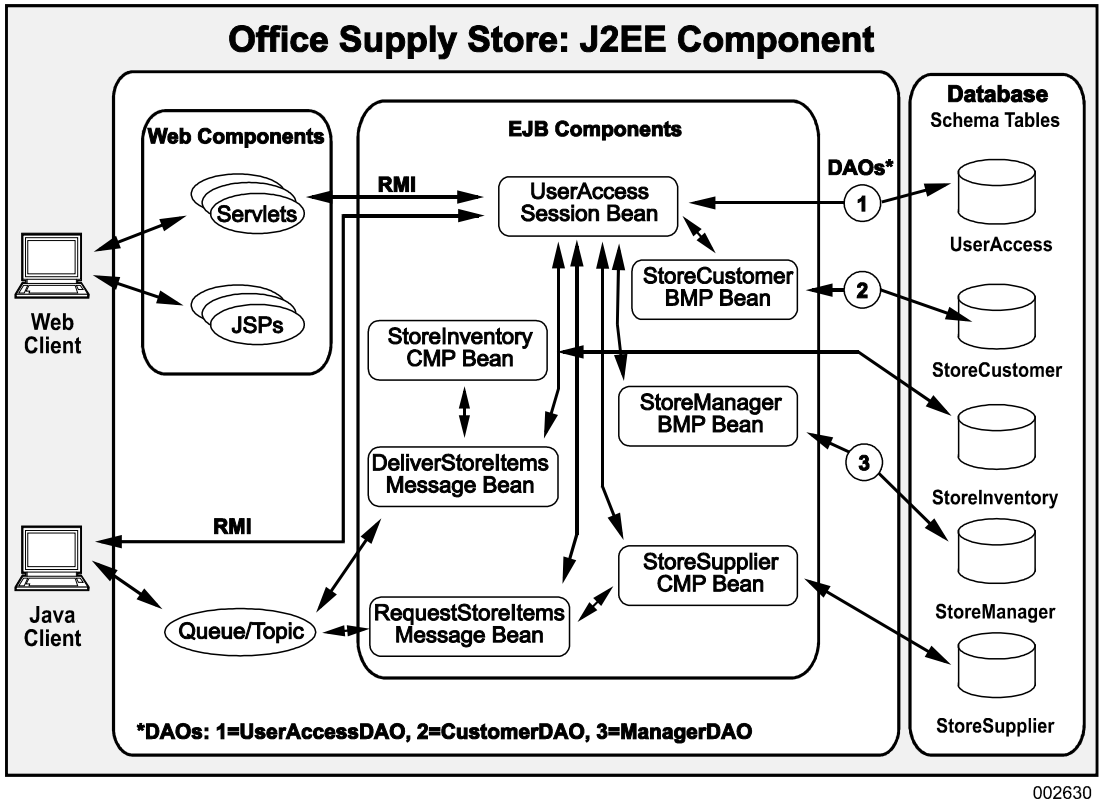

Figure 1–1. Office Supply Store Application Architecture

In the Office Supply Store application

- For Web clients, requests are sent by servlets or JavaServer Pages (JSP) to beans using Remote Method Invocation (RMI).
- For Java clients, requests are made to entity beans using Java middleware technology: RMI or Java Message Service (JMS).
- Stateless and bean-managed persistence (BMP) entity beans access data from the database using data access objects (DAO), which are wrappers for Java Database Connectivity (JDBC) code.
- Container-managed persistence (CMP) entity beans do not require a DAO, because the container manages communication between the beans and the database. This feature is very powerful.

1.1.3. Beans in Office Supply Store

Beans are independent Java program modules that are called by an application in the Java environment (refer to 6.6). JavaBeans are used primarily for developing user interfaces in the client. Entity beans and message-driven beans are Enterprise JavaBeans (EJB) components on the server.

The Office Supply Store implements the following beans:

- UserAccess is a stateless session bean. A session bean exposes its interface to the presentation tier while encapsulating complex business interactions with the other beans. UserAccess forms the backbone of the application and enables the Office Supply Store customers, suppliers, and managers to log in to the system.
- StoreCustomer, StoreManager, StoreInventory, and StoreSupplier are entity beans, which process data. Once authenticated, users request information about Office Supply Store customers, managers, inventory, and suppliers using the various interfaces in UserAccess. These interfaces invoke methods on the entity beans, which supply the requested information.
- RequestStoreItems and DeliverStoreItems are message-driven beans, which pass requests and results between UserAccess and end users. RequestStoreItems and DeliverStoreItems are message-driven beans that listen for messages from a JMS producer and transfer the messages to appropriate beans.

Table 1–2 identifies all beans that are implemented in the Office Supply Store, together with their public behaviors (type) and methods.

Table 1–2. Beans for Office Supply Store

Bean Name	Type	Methods
UserAccessBean	Session	verifyUser() getStoreCustomerData() getStoreManagerData() getStoreSupplierData() getStoreInventoryData() getAllItems() getOutOfStockInventory() getStoreInventoryBySupplier() ejbCreate() setSessionContext( ) unsetSessionContext( )
StoreCustomerBean	Entity	getStoreCustomerData() ejbFindByPrimaryKey()
StoreManagerBean	Entity	getStoreManagerData() ejbFindByPrimaryKey()

Table 1–2. Beans for Office Supply Store

Bean Name	Type	Methods
StoreInventoryBean	Entity	ejbCreate() getStoreInventoryData() addToInventory()
StoreSupplierBean	Entity	ejbCreate() getStoreSupplierData() requestInventoryItem( ) setEntityContext( ) unsetEntityContext( )
RequestStoreItemsBean	Message	onMessage()
DeliverStoreItemsBean	Message	onMessage()

The exercises in this guide develop these beans, build them into the Office Supply Store application, and test the results.

Section 2
Configuring Eclipse IDE to Use JBoss Application Server

A simple enterprise bean (EJB) is used to deploy the components of the Office Supply Store. Before constructing the enterprise bean, ensure that the JBoss Application Server (JBoss AS) is installed and configured as described in the *ClearPath OS 2200 IDE for Eclipse™ Installation Guide*.

2.1. Opening a Java EE Perspective

To open a Java EE perspective

1. Start Eclipse IDE. (If you create a new workspace, you must reconfigure WTP; refer to the *ClearPath OS 2200 IDE for Eclipse™ Installation Guide* for configuring WTP to use JBoss AS and XDoclet.)
2. On the **Window** menu, point to **Open Perspective** and click **Other**.

   The **Open Perspective** dialog box appears.
3. Click **Java EE (default)** and then click **OK**.

   A Java EE perspective window appears.

Close unnecessary windows, such as Welcome and Outline, to allow more viewing room.

Figure 2–1 illustrates the user interface (UI) for an open Java EE perspective. The procedures in this guide refer to the various parts of the UI as needed.

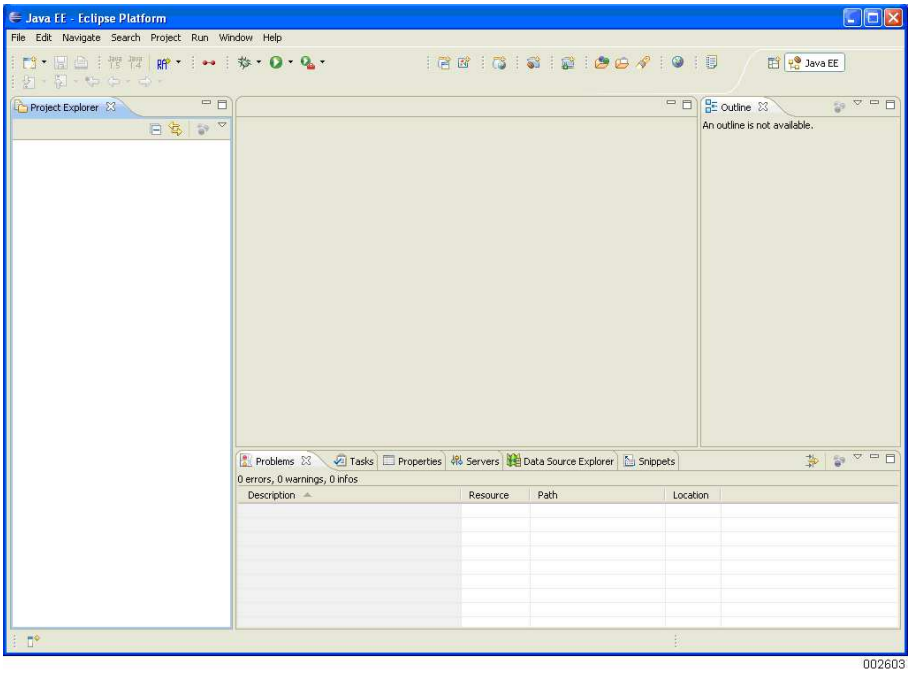

Figure 2–1. Java EE Perspective

2.2. Updating Eclipse 2200 with JBoss Community and RedHat JBoss Enterprise Middleware

To update Eclipse 2200 with JBoss community and RedHat JBoss Enterprise middleware

1. On the Eclipse **Window** menu, click **Preferences**.

   The **Preferences** dialog box appears.

2. Double-click the **Server** node in the left side of the pane to expand the **Server** node and select **Runtime Environments**.

3. Click **Add** in the **Preferences** dialog box.

   The **New Server Runtime Environment** dialog box appears.

4. Click **Download addtitional server adapters** hyperlink.

   The **Install New Extension** dialog box appears and starts populating with the available server adapters.

5. Select **JBossAS Tools (JBoss by Red Hat)** and click **Next**.

6. Accept the license agreement in the Review License page and click **Finish**.

7. Click **OK** in the **Install Extension** dialog box to confirm JBossAS download.

   The **Preferences** dialog box appears. It might take a while to download the update. Click **OK** in the Security Warning window that might appear during the update process.

8. Click **Restart Now** in the Software Updates window to restart the Eclipse.

9. To load the **Server** view from Eclipse, click the **Windows** menu, select **Show View**, and then select **Other**.

   The **Show View** dialog box appears.

10. Type **Servers** in the **Show View** dialog box.

    The tree view is reduced to the **Servers** selection.

11. Select **Servers** and click **OK**.

    The **Servers** view is displayed in the Java EE perspective.

12. Right-click in the **Servers** view, click **New**, and then select **Server**.

    The **New Server** dialog box appears. In addition to the default JBoss server type that comes with Eclipse, the JBoss Community and JBoss Enterprise Middleware server types are also available.

13. Expand either **JBoss Community** or **JBoss Enterprise Middleware** and select the appropriate JBoss server type. **JBoss Enterprise Middleware** JBoss 4.3 and JBoss 5.x are equivalent to the recent OS 2200 IDE for Eclipse JBoss releases.

    The server host name options are: localhost, 127.0.0.1, the DNS name or IP address, or 0.0.0.0. Refer to the "Glossary" for more information about IP addresses. Refer to the *JBoss Application Server™ for ClearPath OS 2200 Installation, Administration, and Programming Guide* for information on JBoss bind address considerations.

14. Click **Next**.

15. Click **Browse** (next to the Home Directory) and navigate to the location where JBoss is installed. If this location is the local installation of JBoss Application Server for ClearPath OS 2200 IDE for Eclipse, navigate to the **jboss511u** or **jboss430GAu** level.

16. Click **OK**.

17. In the **New Server** dialog box, the **Configuration** list is populated with the list of JBoss server instances, that the selected JBoss Home Directory provides. Select the required JBoss server configuration from the list.

18. Click **JRE** to select a version of JRE and select the required version of JRE from the **Installed JRE** list, and click **OK**.

    The available **JRE** options are based on the **JBoss Runtime Name** you selected. If the JBoss Home Directory is a local installation of the JBoss Application Server for ClearPath OS 2200 IDE for Eclipse, refer to the *JBoss Application Server™ for ClearPath OS 2200* for information on required Java levels.

19. Click **Next** in the **New Server** dialog box.

Note: *If you are maintaining JBoss server outside the Eclipse then select the* ***Server is externally managed. Assume server is started*** *checkbox. This option available only with Indigo Release.*

20. Select one of the following:
    - To deploy the application on a local system, select **Local** from the **Server Behavior** list.
    - To deploy the application to a remote system, select **Remote System deployment** from the **Server Behavior** list.

      This enables the **Host**, **Remote Server Home** and **Remote Server Configuration** boxes. Enter the required information in the respective boxes.

    The JBoss server adapter supports remote deployment through Secure Shell (SSH) and Secure Copy (SCP), and the server adapter publishes individual files as well as individual folders. A new JBoss Tools Runtimes preference page is available that enables you to configure any server run time found in a list of directories.
21. Click **Next**.

    The **New Server** dialog box appears.
22. To add one or more resources to the project, select the required resources from the **Available** list and click **Add**. The selected resources are moved to the **Configured** list.
23. Click **Finish**.

2.3. Creating a Java EE Project

A Java EE project is a collection of Java projects that compose the various tiers of an application. You can create some or all projects, at once or as needed, using a Java EE project wizard. Refer to the "Glossary" for more information about Java EE.

To create a Java EE project using the wizard

1. On the **File** menu, point to **New** and click **Project**.

   The New Project wizard appears.
2. Expand **Java EE** and click **Enterprise Application Project**, and then click **Next**.

   The New EAR Application Project wizard appears.
3. Type the project name (MySample in this example) in the **Project Name** box and select one of the following:
   - If the **Target runtime** list is not empty, select the appropriate Target runtime server.
   - If the **Target runtime** list is empty, click **New runtime** next to the list and select the appropriate JBoss version. Select the appropriate **JRE** and **Application Server Directory** from the **New Server Runtime Environment** dialog box and click **Finish**.

4. Verify that EAR version 5.0 is selected in the New EAR Application Project wizard and click **Next**.

   Note: *By default, the* ***Default Configuration*** *is selected. You might need to change the* ***Default Configuration*** *to a predefined configuration. To use a predefined configuration for your project, select a configuration from the* ***Configuration*** *list, and click* ***Modify*** *to do the following:*

   - *To customize the project facets, select the required check boxes (next to the facets) and select a version number for each facet. For more information about the facet, select the required facet in the* ***Details*** *tab. You can also choose a preset combination of facets from the* ***Configurations*** *list.*
   - *To limit your project to be compatible with one or more runtimes, click the* ***Show Runtimes*** *and select the runtimes that you want the project to be compatible with.*

5. Select the **Generate Application.xml deployment descriptor** check box and click **New Module**.

   The New Java EE Module wizard appears.

6. Clear the **Web module** and **Connector module** check boxes and click **Finish**, leaving the **Create default modules**, **Application client module**, and **EJB module** check boxes selected.

   Following a short pause for the wizard to create the projects, the names of the client and EJB projects are populated.

   Note: *The Web module and connector module are not required to create these EJB and application client module.*

7. Click **Finish**.

It is common for the empty EJB project to be in error. The error disappears when an enterprise bean is created.

2.4. Creating Enterprise JavaBeans

Enterprise JavaBeans (EJB) is a server-side component architecture that simplifies the process of building an enterprise-class distributed component application in Java. Using EJB, you can write scalable, reliable, and secure applications without writing complex distributed component framework. Stock trading systems, banking systems, and customer call centers are some examples that use EJB. Refer to Section 6, "Business Components," for information on EJB.

To create the enterprise javabean in the Java EE project

1. Right-click **MySampleEJB** on the **Project Explorer** tab, point to **New**, and click **Other**.

   The **New** dialog box for selecting a wizard appears.

2. Expand **EJB**, click **Session Bean (EJB3.x)**, and then click **Next**.

   The Create EJB 3.x Session Bean wizard appears.

3. Type **us.com.unisys** in the **Java package** box. (Refer to 2.4.1 for information about typing versus browsing to values.)
4. Type **MySampleBean** in the **Class Name** box. (Refer to 2.4.2 for information about naming beans.)
5. Select the **Remote** check box under **Create business interface** (the **Local** check box is selected by default).
6. Click **Next** and then click **Finish**.

   Note: *If "Permgen (out of memory)" error occurs, perform the following steps:*

   1. *On the **Run** menu, click **Run Configuration**.*

      The Run Configurations wizard appears.

   2. *On the **Arguments** tab, change the VM arguments parameters value to "-Xms128m -Xmx512m".*

Figure 2–2 illustrates the EJB project, all the classes, and interfaces generated from the single session bean.

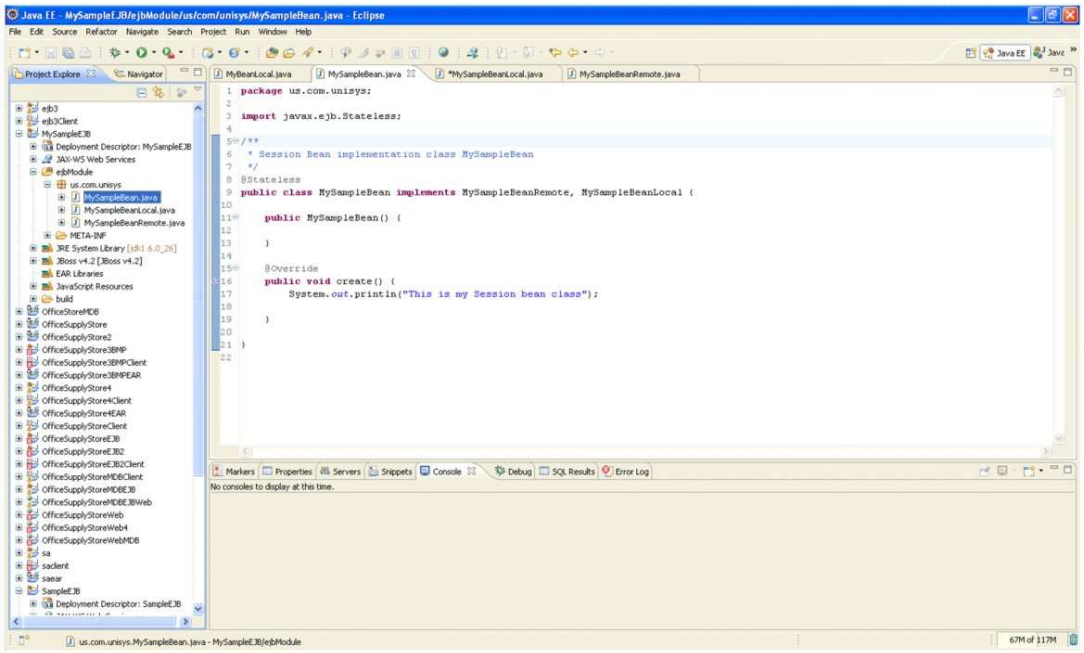

Figure 2–2. EJB Project

2.4.1. Typing Versus Browsing to Names

In many procedures throughout this guide, you cannot browse to a name because it is not yet defined. Browsing requires creating the item previously in a separate step. Once an item is created, its name is defined in a list for browsing.

2.4.2. Naming Beans

The Create EnterpriseJavaBean wizard forces the bean class name to end with Bean. Other generated classes also have specific suffixes, as specified throughout this guide.

To avoid confusion with generated classes, do not use any of the following suffixes as the name of your bean: Bean, EJB, Home, Local, or Session.

2.5. Creating Business Method for the EJB Project

To create a business method for the EJB project

1. Open MySampleBeanlocal or MySampleBeanRemote interface and declare a method called create() as follows:

```
public void create();
```

Figure 2–3 illustrates the Create method in the Interface Module for the MySampleEJB Project.

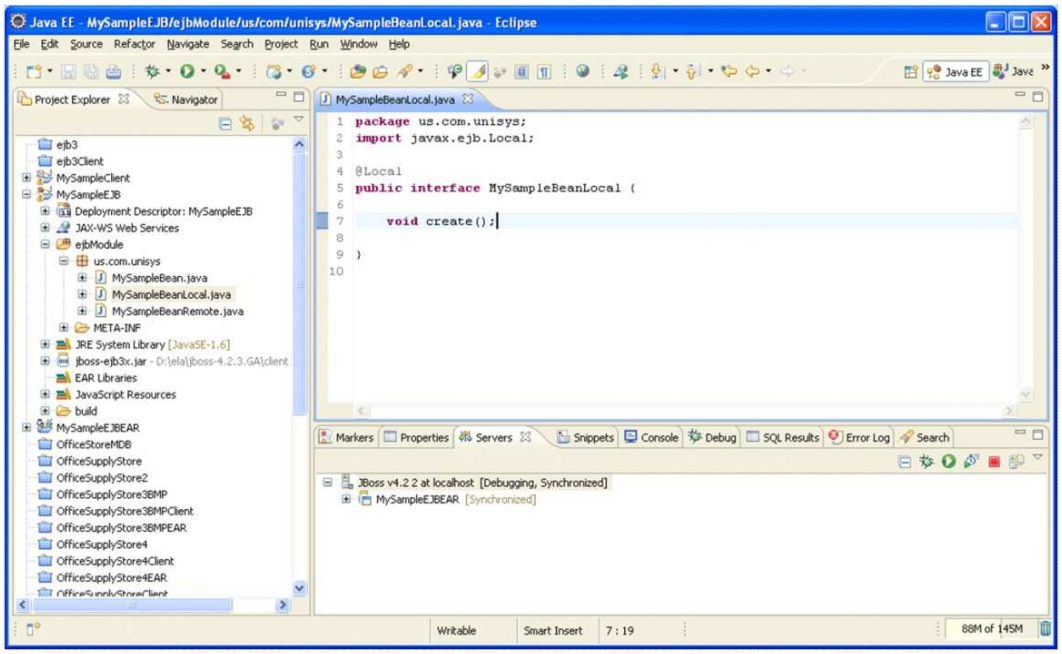

Figure 2–3. Interface Module for the MySampleEJB Project

2. Open MySampleBean class and write the implementation for create() method as follows:

```
public void create()
{
      System.out.println("EJB project created");
}
```

Figure 2–4 illustrates the Create method in the Remote and Local modules of the MySampleEJB Project.

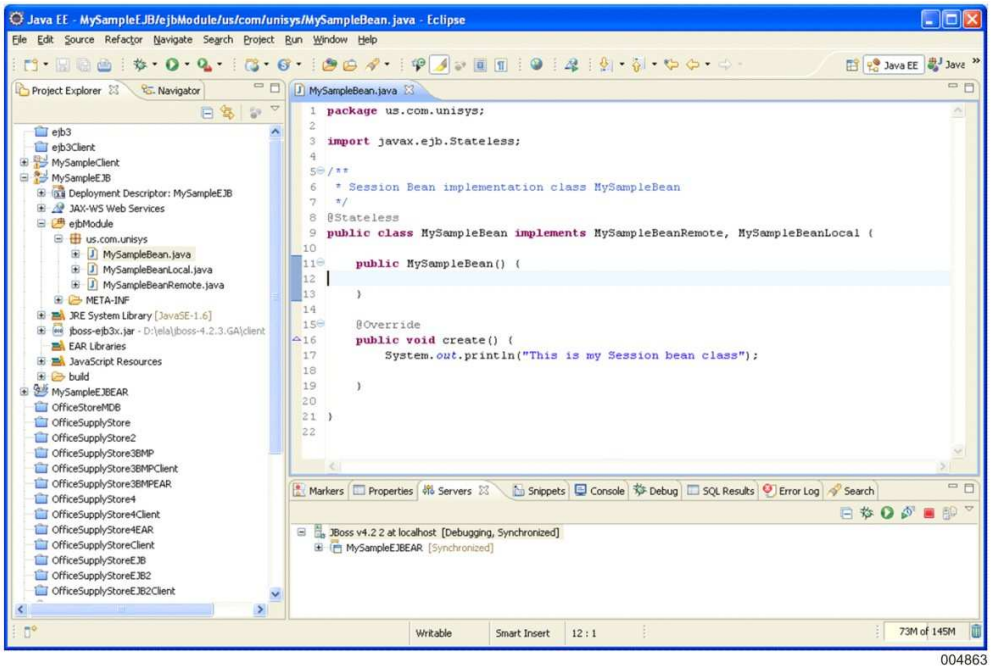

Figure 2–4. Create method in the Remote and Local Modules of the MySampleEJB Project

2.6. Configuring JBoss AS as the Deployment Server

Note: *Before configuring JBoss AS as the deployment server, ensure that the Server view is active in your Java EE perspective. To load the Server view in Eclipse, click the* ***Window*** *menu, point to* ***Show View****, and click* ***Servers.***

To configure JBoss AS as the application server for deploying the bean, perform the following steps:

- If there are any instances of a server configured in the project, perform the following steps:
    a. Right-click the required server and click **Add and Remove**.

       The **Add and Remove** dialog box appears.

       – To add one or more resources to the project, select the required resources from the **Available** list and click **Add**. The selected resources are moved to the **Configured** list.

       – To remove one or more resouces from the JBoss deployment, select the required resources from the **Configured** list and click **Remove**. The selected resources are moved to the **Available** list.

    b. Click **Finish**.
- If there are no instances of server configured in the project, follow the steps 9 through 21 of section 2.2.

2.7. Running JBoss AS

You can start JBoss AS from either Eclipse IDE or a command prompt (refer to 3.3). Starting from Eclipse IDE is recommended because debugging and deploying the application is easier.

2.7.1. Starting JBoss AS

To start JBoss AS from Eclipse IDE, right-click the **JBoss** root node on the **Servers** tab and click **Start**.

The startup process displays the **Console** tab. JBoss AS starts and then displays the **Servers** tab when JBoss AS is running.

Note: *If the "unable to run JBoss within 50 sec" error occurs, open Server (JBoss v4.2 at local host), click* ***timeouts****, and increase the start and stop time values.*

2.7.2. Cleaning the Project

Examine the **Console** tab for deployment errors. JBoss AS can start successfully even if the bean fails to deploy. If the bean fails to deploy, clean the project, as follows:

1. Stop JBoss AS.
2. Click **Clean** on the **Project** menu.
3. Start JBoss AS.

Refer to Appendix A if the bean still fails to deploy properly.

2.7.3. Stopping JBoss AS

To stop JBoss AS from Eclipse IDE, right-click the **JBoss** root node on the **Servers** tab and click **Stop**.

2.8. Building a Test Client

To build a test client

1. Right-click **MySampleClient** on the **Project Explorer** tab, point to **New**, and click **Class**.
2. Type **test** in the **Package** box.
3. Type **TestClient** in the **Name** box.
4. Be sure **public** is selected in the **Modifiers** list.
5. Select **public static void main(String[] args)** method stub.

   Inherited abstract methods is already selected.
6. Click **Finish**.

   TestClient.java code file is created.
7. Add the following code in the TestClient.java code file:

```
package test;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import us.com.unisys.MySampleBeanRemote;
public class TestClient {
    public static void main(String[] args) {
        try {
            Properties props = new Properties();
                        props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
            props.setProperty("java.naming.factory.url.pkgs", "org.jboss.
naming");
```

```
                props.setProperty("java.naming.provider.url", "127.0.0.1:1099
");
                InitialContext ctx = new InitialContext(props);
                MySampleBeanRemote bean = (MySampleBeanRemote)
                ctx.lookup("MySampleEJB/MySampleBean/remote");
                 bean.create();
            } catch (NamingException e) {
                e.printStackTrace();
            }
        }
    }
```

The client contains compilation errors because the client project depends on the EJB project.

Figure 2–5 illustrates the compilation errors in Test client.

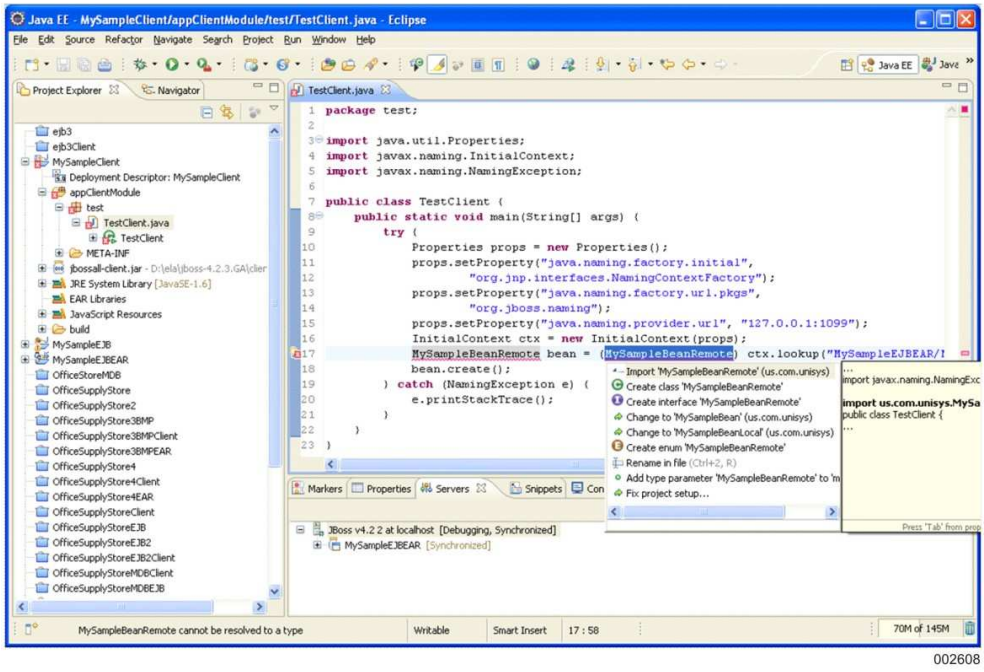

Figure 2–5. Compilation Errors in Test Client

2.9. Resolving Compilation Errors

To add an EJB project dependency and resolve compilation errors in the client project

1. On the **Project Explorer** tab, right-click the **MySampleClient** and click **Properties**.

   The **Properties for MySampleClient** dialog box opens.
2. Click **Java Build Path**.
3. Click **Add** on the **Projects** tab.

   The **Required Project Selection** dialog box opens.
4. Select **MySampleEJB** to add a project dependency and click **OK**.
5. Click **OK** to close the **Properties for MySampleClient** dialog box.
6. If the TestClient code contains an error at the MySampleBeanRemote line within the **try** block, click the cross mark for that line and double-click **Fix Project Setup**.

   The Project Setup Fixes wizard appears.
7. Click **OK**.

The test client should now compile cleanly.

2.10. Testing the Project

To start JBoss from Eclipse IDE, right-click the **JBoss** root node on the **Servers** tab and click **Start**.

The startup process displays the Console tab. JBoss starts and then displays the Servers tab when JBoss is running.

To run the test, right-click **TestClient.java** on the **Project Explorer** tab, point to **Run as**, and click **Java Application**. Figure 2–6 shows the output window.

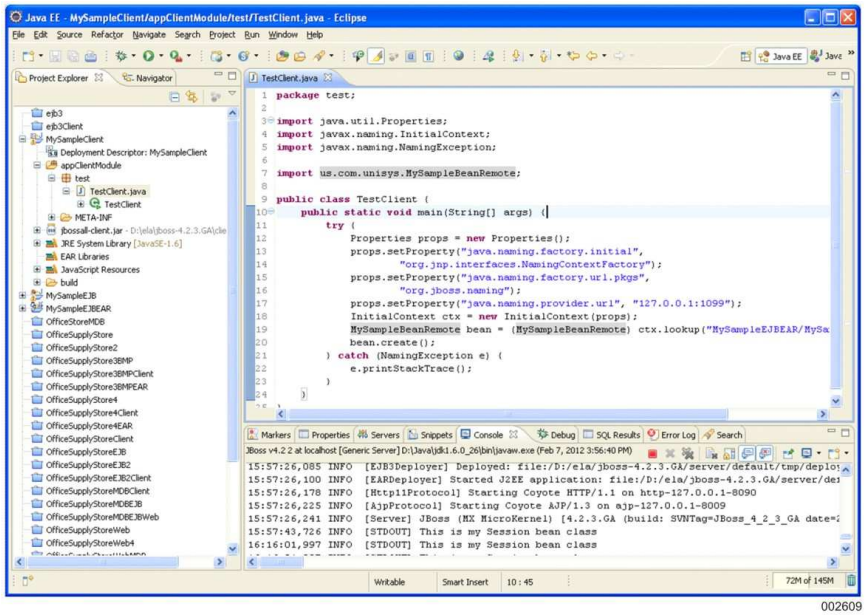

Figure 2–6. Running the Test Client

2.11. Troubleshooting the Project

The test client might fail as some of the files might not have been compiled properly or it might contain old class files. As the first step, you must clean all the projects.

To clean all the projects

1. On the **Project** menu, click **Clean...**.

   The Clean window appears.

2. Select **Clean all projects** and click **OK**.

See Appendix A for best practices and Appendix B for additional troubleshooting tips.

2.12. Hot-Deploying the Project

Hot-deployment means deploying while JBoss AS is running.

To hot-deploy the project, modify the bean by editing the code so that you know you are executing the revised bean and save the file.

Figure 2–7 illustrates the updated project state.

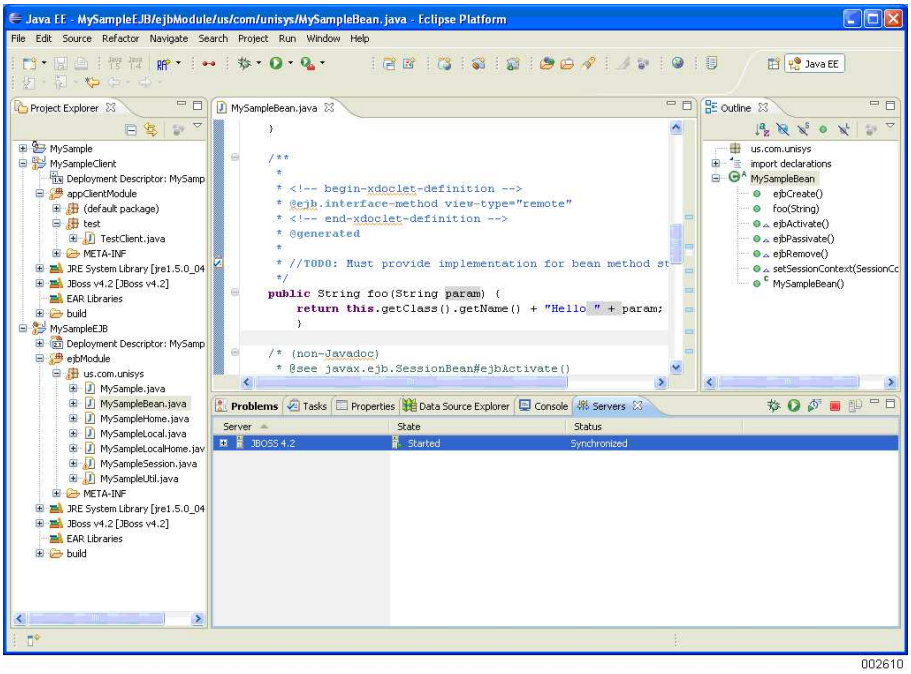

Figure 2–7. Hot-Deploying the Project

Run the test client again and note the change in the trace output, showing that you are exercising the modified bean.

Section 3
Loading the Hypersonic Database

The schema for the Office Supply Store inventory consists of five tables. Load the schema and data into the embedded Hypersonic database using the following scripts:

- OSStoreSchema.script. Creates the database schema.
- OSStoreInsertData.script. Populates the tables with data.
- OSStoreDropTables.script. Drops and deletes all tables in this example.

Note: *The Hypersonic database is used for demonstration purposes only and is not considered to be of enterprise quality.*

3.1. Scripts for Loading and Populating the Database

The OSStoreSchema and OSStoreInsertData scripts (refer to Figure 3–1 and Figure 3–2) load and populate the embedded Hypersonic database. Refer to 3.4 to run the scripts.

OSStoreSchema.script

```
CREATE TABLE USERACCESS(STOREACCESSID VARCHAR(8) NOT NULL PRIMARY KEY,
USERNAME VARCHAR(20),PASSWORD VARCHAR(8))

CREATE TABLE STORECUSTOMER(CUSTOMERID VARCHAR(8) NOT NULL PRIMARY KEY,
FNAME VARCHAR(18),LNAME VARCHAR(18),STREETADDRESS VARCHAR(28),CITY VARCHAR(28),
STATE VARCHAR(2),ZIPCODE VARCHAR(5),NOTES VARCHAR(40))

CREATE TABLE STORESUPPLIER(SUPPLIERID VARCHAR(8) NOT NULL PRIMARY KEY,
COMPANYNAME VARCHAR(18),STREETADDRESS VARCHAR(28),CITY VARCHAR(28),STATE VARCHAR(2),
ZIPCODE VARCHAR(5),NOTES VARCHAR(40))

CREATE TABLE STOREMANAGER(MANAGERID VARCHAR(8) NOT NULL PRIMARY KEY,
FNAME VARCHAR(18),LNAME VARCHAR(18),STREETADDRESS VARCHAR(28),CITY VARCHAR(28),
STATE VARCHAR(2),ZIPCODE VARCHAR(5),NOTES VARCHAR(40))

CREATE TABLE STOREINVENTORY(ITEMNUM VARCHAR(10) NOT NULL PRIMARY KEY,
SUPPLIERID VARCHAR(8),DESCRIPTION VARCHAR(40),QTYONHAND INTEGER,PRICE DECIMAL(12,2) N
OT NULL)
```

Figure 3–1. Script to Load Hypersonic Database Schema

OSStoreInsertData.script

```
INSERT INTO USERACCESS(STOREACCESSID,USERNAME,PASSWORD)
 VALUES('USER1','FRANK1','PASSWD1')
INSERT INTO USERACCESS(STOREACCESSID,USERNAME,PASSWORD)
 VALUES('USER2','ANN123','PASSWD2')
INSERT INTO USERACCESS(STOREACCESSID,USERNAME,PASSWORD)
 VALUES('USER3','PAUL12','PASSWD3')
INSERT INTO USERACCESS(STOREACCESSID,USERNAME,PASSWORD)
 VALUES('USER4','SAND12','PASSWD4')
INSERT INTO USERACCESS(STOREACCESSID,USERNAME,PASSWORD)
 VALUES('USER5','SUE123','PASSWD5')
INSERT INTO USERACCESS(STOREACCESSID,USERNAME,PASSWORD)
 VALUES('USER6','DAN123','PASSWD6')

INSERT INTO STORECUSTOMER(CUSTOMERID,FNAME,LNAME,STREETADDRESS,CITY,STATE,ZIPCODE,NOT
ES)
 VALUES('CUST1','Frank','Jones','123 Main St.','Minneapolis','MN',54101,'Likes Sony products.')
INSERT INTO STORECUSTOMER(CUSTOMERID,FNAME,LNAME,STREETADDRESS,CITY,STATE,ZIPCODE,NOT
ES)
 VALUES('CUST2','Ann','Thomas','3456 1st St.','Minneapolis','MN','54112','Big customer, be nice!')
INSERT INTO STORECUSTOMER(CUSTOMERID,FNAME,LNAME,STREETADDRESS,CITY,STATE,ZIPCODE,NOT
ES)
 VALUES('CUST3','Paul','Stephens','9876 Pine Rd.','Anoka','MN','53111','New client.')
INSERT INTO STORESUPPLIER(SUPPLIERID,COMPANYNAME,STREETADDRESS,CITY,STATE,ZIPCODE,NOT
ES)
```

```
 VALUES('SUPL1','Anderson Supply','101 Front St.','Roseville','MN','55113','Electronics')
INSERT INTO STORESUPPLIER(SUPPLIERID,COMPANYNAME,STREETADDRESS,CITY,STATE,ZIPCODE,NOT
ES)
 VALUES('SUPL2','Peters Papers','456 Brown Ave.','St. Paul','MN','55101','Paper Supplies')

INSERT INTO STOREMANAGER(MANAGERID,FNAME,LNAME,STREETADDRESS,CITY,STATE,ZIPCODE,NOTES
)
 VALUES('MANAGE1','Daniel','Nelson','3 Dodge Rd.','Minneapolis','MN','54111','The Boss.')

INSERT INTO STOREINVENTORY(ITEMNUM,SUPPLIERID,DESCRIPTION,QTYONHAND,PRICE)
 VALUES('ITEM1','SUPL1','SAMSUNG PDA',0,245.95)
INSERT INTO STOREINVENTORY(ITEMNUM,SUPPLIERID,DESCRIPTION,QTYONHAND,PRICE)
 VALUES('ITEM2','SUPL1','HP SCANNER',190,110.50)
INSERT INTO STOREINVENTORY(ITEMNUM,SUPPLIERID,DESCRIPTION,QTYONHAND,PRICE)
 VALUES('ITEM3','SUPL2','EPSON PRINTER',90,200.10)
INSERT INTO STOREINVENTORY(ITEMNUM,SUPPLIERID,DESCRIPTION,QTYONHAND,PRICE)
 VALUES('ITEM4','SUPL1','KODAK CAMERA',0,345.00)
```

Figure 3–2. Script to Populate Hypersonic Database

3.2. Script for Purging the Database

The OSStoreDropTables script (refer to Figure 3–3) deletes tables in the database. Run this script when you want to purge the database and start again.

OSStoreDropTables.script

```
DROP TABLE STOREINVENTORY
DROP TABLE STORESUPPLIER
DROP TABLE STOREMANAGER
DROP TABLE STORECUSTOMER
DROP TABLE USERACCESS
```

Figure 3–3. Script to Delete Hypersonic Database Tables

3.3. Opening the Console

The HSQL Database Manager window is the console for the embedded Hypersonic database. The console is used to access schemas and do other database management operations.

To open the console

1. Start JBoss AS from the command prompt window. Using command prompt, open the drive in which JBoss is present and enter the following commands:

   ```
   C:\>cd\jboss\bin

   C:\jboss\bin>run
   ```

   JBoss AS starts.

2. Refer to Appendix A for troubleshooting tips.
3. Enter the following URL in your Web browser:

   ```
   http://localhost:8080/jmx-console
   ```

   The JBoss JMX Management Console window appears.
4. Under the **jboss** subheading, click **database=localDB,service=Hypersonic**.

   The JMX MBean View page appears on the MBean Inspector window.
5. Scroll to **startDatabaseManager** (an MBean operation near the bottom of the page) and click **Invoke**.

   Close the **DataBaseManagerSwing Font Selection Dialog** dialog box if it appears.

   The HSQL Database Manager window (console) appears. (If you do not see the console, check the taskbar. The console might be hidden behind the browser window.)

3.4. Creating and Populating the Database

If you perform these procedures more than once, you must first remove the previously created tables by running the **OSStoreDropTables.script** script (refer to 3.1).

To create and populate the database

1. Click **Open Script** on the **File** menu of the HSQL Database Manager console.

   The **Open Script** dialog box opens.
2. Select the directory containing the script files in the **Look in** list, select **OSStoreSchema.script**, and click **Open**.
3. Click **Execute SQL Statement** on the console.
4. Click **Refresh Tree** on the **View** menu to see the results from the script.
5. Click **Open Script** on the **File** menu.

   The **Open Script** dialog box opens.
6. Select the directory containing the script files in the **Look in** list, select **OSStoreInsertData.script**, and click **Open**.
7. Click **Execute SQL Statement** on the console.
8. Click **Refresh Tree** on the **View** menu to see the results from the script.
9. Click **Commit** on the **Options** menu to commit the changes.

   If **Commit** is not visible, select the **Autocommit mode** check box from the **Options** menu.

3.5. Verifying the Database

To verify the database by running a database query

1. Click **Clear SQL Statement** on the console.
2. Enter the following query in the box:

   ```
   SELECT * FROM USERACCESS
   ```

3. Click **Execute SQL Statement** on the console.

Figure 3–4 illustrates the console with data.

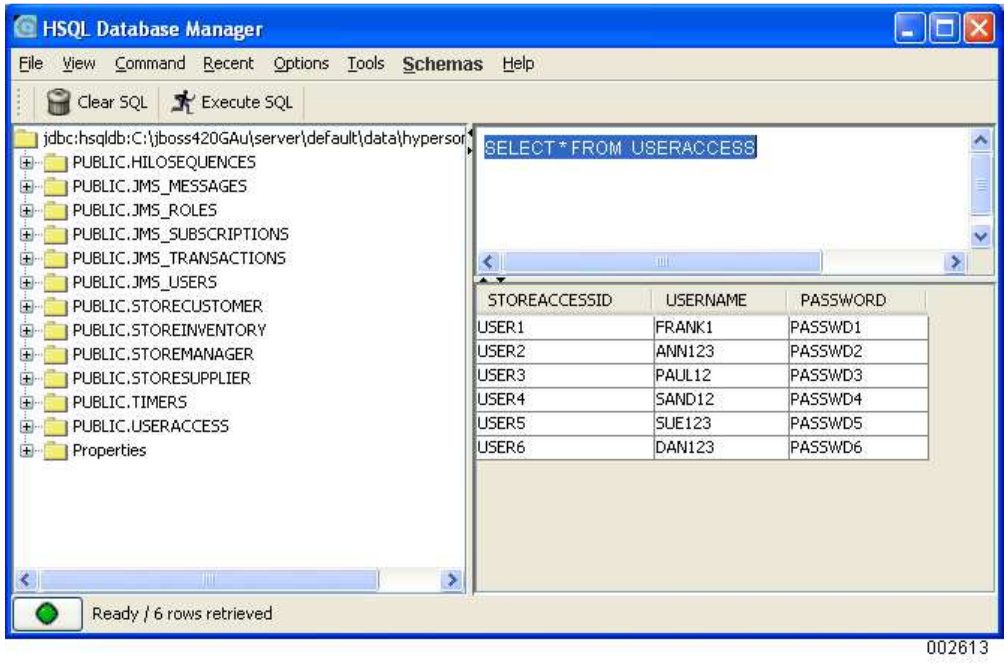

Figure 3–4. HSQL Database Manager with Data

3.6. Viewing the Database Using Data Source Explorer

JBoss AS must not be running during the following procedure. If JBoss AS is running, an error appears when you try to connect to the Hypersonic database.

To view the Office Supply Store database with Data Source Explorer

1. Open a **Java EE** perspective in Eclipse IDE.

   Refer to 2.1 for the steps.
2. Right-click **Databases** in the **Data Source Explorer** pane and click **New**.

   The **New Connection Profile** dialog box for selecting a wizard appears.
3. Select **HSQLDB under Connection Profile types**.
4. Type **OfficeSupplyStore-Hypersonic** in the **Name** box and click **Next**.
5. Click the button next to the **driver** list.
6. Perform the following in the **New Driver Definition** dialog box that is displayed:
   a. Select the **HSQLDB JDBC DRIVER** in the **Name/Type** tab.

      HSQLDB JDBC DRIVER appears in the **Driver Name** box.
   b. Click **Jar List** tab, click **Add Jar/Zip**, browse to **hsqldb.jar**, and click **Open**.

      If hsqldb.jar is already present under **Driver File(s)**, click **Edit Jar/Zip** to modify the driver file.
   c. Modify the **Connection URL** to **jdbc:hsqldb:C:\jboss-4.2.3.GA\server\default\data\hypersonic\localDB**.
   d. Specify localDB for the database in the **Database Name** box and click **OK**.
   e. Select **HSQLDB JDBC Driver** in the **Drivers** dialog box and click **OK**.
7. Type **sa** in the **User name** box (under the properties pane in **General** tab), leaving the **Password** box blank and click **Next**.

   Notes:

   - *Click **Test Connection**, if you want to verify whether you are able to connect to the server.*
   - *Click **OK**, if the **Ping Failed** dialog box is displayed. From the command prompt window, enter Ctrl+C to stop JBoss server. The connection is made successfully.*
8. Click **Finish**.

   The new connection, **OfficeSupplyStore-Hypersonic**, is added under **Databases**.
9. Right-click **OfficeSupplyStore-Hypersonic** and click **Connect**.

10. Expand **localDB, Catalogs, PUBLIC, Schemas**, **Tables**, and then **STORECUSTOMER**.

    The structure of the Store Customer table is displayed.

11. Right-click **STORECUSTOMER**, point to **Data** and click **Sample Contents**.

    The **SQL Results** pane appears with the SQL select statement generated in the **Status** tab.

12. Click the **Result1** tab to view the data of the Store Customer table.

Figure 3–5 illustrates the data for the Store Customer table in the SQL Results pane.

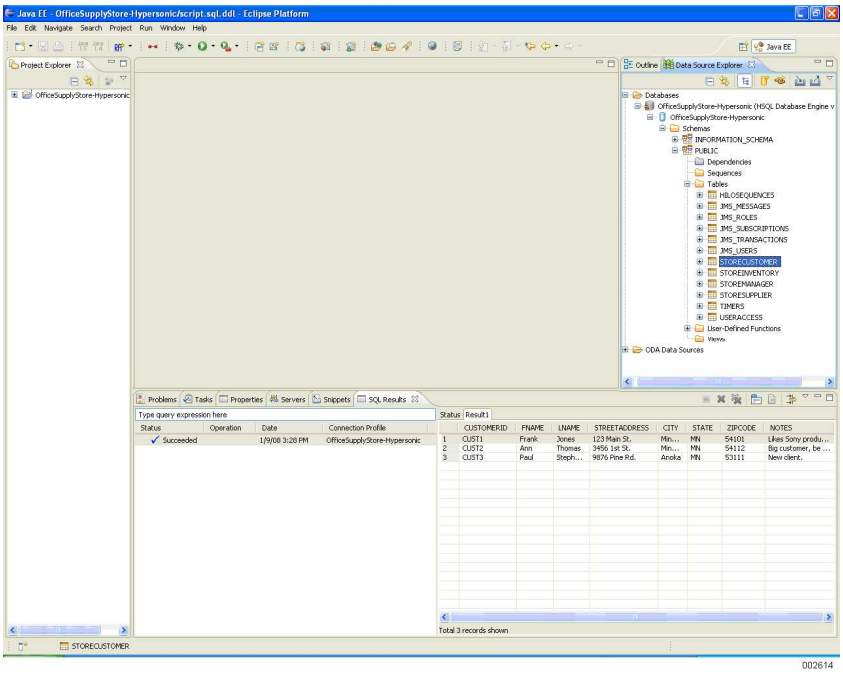

Figure 3–5. Hypersonic Database Contents

3.7. Generating DDL Scripts

You can generate customized DDL scripts for the tables in the database. To generate a DDL script for the Store Customer table, perform the following steps:

1. Right-click **STORECUSTOMER** in the Data Source Explorer pane and click **Generate DDL**.
2. Select the model elements and objects that you want to include in the DDL script, and click **Next**.
3. Click **Browse** to select an existing Java project (MySample in this example).

   The corresponding .sql file name is populated in the **File name** box.

   Note: *If a Java project is not available, refer to 5.3.2 for the procedure to create a Java project.*
4. Click **Next** and then click **Finish**.

   The DDL script is generated in the .sql file you specified.
5. Double-click the .sql file on the **Project Explorer** tab.

   The file opens in the Connection profile view.
6. Select **HSQLDB_1.8** from the **Type** list, **OfficeSupplyStore-Hypersonic** from the **Name** list, and *database name* from the **Database** list, where *database name* is the name you specified for the database in 3.6.
7. Select the query you want to execute, right-click, and click **Execute Current Text**.

   The query is executed and the result is displayed in the **SQL Results** view.

Section 4
Loading the Relational Database Server Database

For demonstration purposes, this section uses OS 2200 application group 3 and an administrator user-id of jvmtst in all procedures. Make appropriate changes if you use a different application group or have a different administrator user-id. Refer to 4.2 for the items that you might have to change.

4.1. Testing the Database

The Office Supply Store schema and database must be loaded only once. To determine if the Relational Database Server database is already populated, log on to the OS 2200 host using Telnet or a UTS emulator and enter the following commands:

```
>@dd,e ,,udssrc
>Report schema osupplystore.
```

If the output displays the name of the schema and user-id of the administrator, skip the steps that populate the database. If the following message appears, perform all steps:

```
*WARNING UREP1715: UREP cannot find the report entity.
```

4.2. Creating and Populating the Database

4.2.1. Modifying the Add Stream

An add stream is provided to create the Office Supply Store schema and populate the database. The add stream is located in *folder*/file: database scripts/setup2200. In the following copy, the items you might have to modify appear in bold font.

WARNING

End-of-line terminators in OS 2200 source files must be LF (line feed) characters. If you edit the file with Eclipse IDE or on the OS 2200 system, end-of-line terminators are correct. Windows editors, such as Notepad or Wordpad, insert CR (carriage return) and LF (line feed) characters at the end of each line, causing the add stream to work incorrectly on the OS 2200 system.

```
@ .
@ . DELETE ALL STORAGE areas, tables, schemas
@ . ----------------------------------------
@DD,E
process STORAGE-AREA StoreSupplier      FOR SCHEMA OSupplyStore delete.
process STORAGE-AREA StoreCustomer      FOR SCHEMA OSupplyStore delete.
process STORAGE-AREA StoreManager       FOR SCHEMA OSupplyStore delete.
process STORAGE-AREA StoreInventory     FOR SCHEMA OSupplyStore delete.
process STORAGE-AREA UserAccess         FOR SCHEMA OSupplyStore delete.
exit.
@xqt,e sys$lib$*rsa.rsac-coivp
drop table        OSupplyStore.StoreSupplier ;
drop table        OSupplyStore.StoreCustomer ;
drop table        OSupplyStore.StoreManager ;
drop table        OSupplyStore.StoreInventory ;
drop table        OSupplyStore.UserAccess ;
COMMIT thread;
Exit;
@DD,E
delete  schema OSupplyStore .
exit.
@ .
@ . Create Schema
@ . -------------
@XQT SYS$LIB$*RSA.RSAC-COIVP
ECHO ON;
BEGIN THREAD FOR UDSSRC UPDATE;
CREATE SCHEMA OSupplyStore AUTHORIZATION jvmtst
  CREATE TABLE StoreSupplier
     ( SupplierID character(8) NOT NULL PRIMARY KEY,
       CompanyName  character(18),
       StreetAddress character(28),
       City  character(28),
       State character(2 ),
       ZipCode character(5),
       Notes character(40) )
  GRANT ALL PRIVILEGES ON StoreSupplier TO PUBLIC
  CREATE TABLE StoreCustomer
     ( CustomerID character(8) NOT NULL PRIMARY KEY,
       FName character(18),
       LName character(18),
       StreetAddress character(28),
       City  character(28),
       State character(2 ),
       ZipCode character(5),
       Notes character(40) )
  GRANT ALL PRIVILEGES ON StoreCustomer TO PUBLIC
```

Figure 4–1. Add Stream to Populate Relational Database Server Database (cont.)

```
  CREATE TABLE StoreManager
     ( ManagerID character(8) NOT NULL PRIMARY KEY,
       FName character(18),
       LName character(18),
       StreetAddress character(28),
       City  character(28),
       State character(2 ),
       ZipCode character(5),
       Notes character(40) )
  GRANT ALL PRIVILEGES ON StoreManager TO PUBLIC
  CREATE TABLE StoreInventory
     ( ItemID character(10) NOT NULL PRIMARY KEY,
       SupplierID character(8),
       Description character(40),
       QtyOnHand INTEGER,
       Price DECIMAL(12,2) NOT NULL )
  GRANT ALL PRIVILEGES ON StoreInventory TO PUBLIC
  CREATE TABLE UserAccess
     ( UserName character(20),
       Password character(8),
       StoreAccessID character(8) NOT NULL PRIMARY KEY )
  GRANT ALL PRIVILEGES ON UserAccess TO PUBLIC ;
COMMIT;
EXIT;
@ .
@ . Insert some data
@ . ----------------
@xqt sys$lib$*rsa.rsac-coivp
echo on;
begin thread for UDSSRC update;
INSERT INTO OSupplyStore.UserAccess VALUES ('FRANK1','PASSWD1','USER1');
INSERT INTO OSupplyStore.UserAccess VALUES ('ANN123','PASSWD2','USER2');
INSERT INTO OSupplyStore.UserAccess VALUES ('PAUL12','PASSWD3','USER3');
INSERT INTO OSupplyStore.UserAccess VALUES ('SAND12','PASSWD4','USER4');
INSERT INTO OSupplyStore.UserAccess VALUES ('SUE123','PASSWD5','USER5');
INSERT INTO OSupplyStore.UserAccess VALUES ('DAN123','PASSWD6','USER6');
INSERT INTO OSupplyStore.StoreCustomer VALUES ('CUST1'
  ,'Frank','Jones','123 Main St.','Minneapolis','MN','54101'
  ,'Likes Sony products.');
INSERT INTO OSupplyStore.StoreCustomer VALUES ('CUST2'
  ,'Ann','Thomas','3456 1st St.','Minneapolis','MN','54112'
  ,'Big customer, be nice!');
INSERT INTO OSupplyStore.StoreCustomer VALUES ('CUST3'
  ,'Paul','Stephens','9876 Pine Rd.','Anoka','MN','53111'
  ,'New client.');
INSERT INTO OSupplyStore.StoreSupplier VALUES ('SUPL1','Anderson Supply'
  ,'101 Front St.','Roseville','MN','55113','Electronics');
```

Figure 4–1. Add Stream to Populate Relational Database Server Database (cont.)

```
INSERT INTO OSupplyStore.StoreSupplier VALUES ('SUPL2','Peters Papers'
  ,'456 Brown Ave.','St. Paul','MN','55101','Paper Supplies');
INSERT INTO OSupplyStore.StoreManager VALUES ('MANAGE1'
  ,'Daniel','Nelson','3 Dodge Rd.','Minneapolis','MN','54111','The Boss');
INSERT INTO OSupplyStore.StoreInventory VALUES ('ITEM1','SUPL1'
  ,'SAMSUMG PDA',0,245.95);
INSERT INTO OSupplyStore.StoreInventory VALUES ('ITEM2','SUPL2'
  ,'HP SCANNER',18,110.50);
INSERT INTO OSupplyStore.StoreInventory VALUES ('ITEM3','SUPL2'
  ,'EPSON PRINTER',32,200.10);
INSERT INTO OSupplyStore.StoreInventory VALUES ('ITEM4','SUPL1'
  ,'KODAK CAMERA',0,345.55);
commit;
end thread;
exit;
@end
```

Figure 4–1. Add Stream to Populate Relational Database Server Database

4.2.2. Uploading the Add Stream

Upload the add stream to the OS 2200 system using FTP or CIFS/SMB. If the OS 2200 file system is not mounted as a drive using CIFS/SMB, use FTP. Open a command prompt window and perform the following, using your log-in, file source, and destination:

```
U:\>ftp rs02
Connected to rs02.rsvl.unisys.com.
220 1100JD1100 Service ready for new user.
User (rs02.rsvl.unisys.com:(none)): dps
331 User name okay, need password.
Password: *******
332 Need account for login.
Account: 164153
230 User logged in, proceed.
ftp> put "d:\app guide 3.1\database scripts\setup2200" dps*temp.setup2200
200 Command okay.
150 File status okay; about to open data connection.
226 Closing data connection; requested file action successful.
ftp: 4437 bytes sent in 0.00Seconds 4437000.00Kbytes/sec.
```

Figure 4–2. Uploading the Add Stream

4.3. Accessing Relational Database Server

Open a Telnet session to your OS 2200 site, logon using a user-id that has access to the Relational Database Server application environment that you wish to use (this example uses the user-id jvmtst, which is the administrator of application group 3).

Enter the following command in the Telnet session (assuming you uploaded the file to the same location as in 4.2.2). After the add stream completes, the schema and tables are created.

```
@add dps*temp.setup2200
```

Run the dde command to check whether the command is successful (refer to 4.1) and terminate the Telnet session.

4.4. Setting Up Drivers to Access the Database

To set up drivers to access the Relational Database Server, download the appropriate drivers from the OS 2200 system and, for convenience, place it in the C:\Database Drivers folder. In this example, download the drivers for application group 3 using CIFS/SMB, as follows:

a. Using Windows File Explorer, mount the share name **OS 2200** on the OS 2200 system that you are going to access.

b. Browse to the uds$$src/jdbc$client folder, where src is application group 3.

c. Copy the files rdmsdriver.jar and unisys-jca.jar to the Database Drivers folder.

Note: *It is important to retrieve the correct rdmsdriver.jar. Different OS 2200 systems and different application groups can require different rdmsdriver.jar files.*

For more information on setting up drivers, refer to the *Relational JDBC Driver for ClearPath OS 2200 User Guide*.

4.5. Setting Up Data Source Explorer to View the Database

To set up Data Source Explorer to view the Office Supply Store database

1. Open a **Java EE** perspective in the Eclipse IDE.

   Refer to 2.1 for the steps.

2. Right-click **Database connections** in the Data Source Explorer pane and click **New**.

   The **New Connection Profile** dialog box appears.

3. Select **Generic JDBC Connection** from the **New Connection Profile** dialog box.

4. Type **RDMS-OSupplyStore** in the **Name** box and click **Next**.

5. Click the button next to the **Drivers** list.

   The **New Driver Definition** dialog box appears.

   a. Select **Generic JDBC Driver** from the **Available Driver Templates** list.

      Generic JDBC Driver appears in the **Driver Name** box.

   b. Click **Jar List.**

   c. Browse to the files **unisys-jca.jar** and **rdmsdriver.jar** that you copied to c:\Database Drivers in 4.4, and then click **Open**.

   d. Type **jdbc:rdms:host=*name*; port=1544; varchar=varchar; schema=OSupplyStore** in the **Connection URL** box

      where

      name is the name of your OS 2200 host.

      1544 is the default port number for application group 3; your port number can differ if you are using an application group other than 3 or are not using the default port.

   e. Select **Driver Class**.

      A button appears in the right pane.

   f. Click the button.

      The **Available Classes from Jar List** dialog box appears.

   g. Click **Browse** to select the class paths. Select **com.unisys.os2200.rdms.jdbc.RdmsDriver** and click **OK**.

   h. Click **OK** to close the **Edit Driver Definition** and **Driver Definition** dialog boxes.

6. Select **Generic JDBC Driver** from the **Drivers** list.

7. Type your user name and password in the **User name** and **Password** boxes, and click **Next**.

   Note: *Click* ***Test Connection*** *if you want to verify whether you are able to connect to the server.*

8. Click **Finish**.

   The new connection, **RDMS-OSupplyStore**, is added under **Database connections**.

4.6. Viewing the Database Using Data Source Explorer

Before attempting to connect to the Relational Database Server using JDBC, ensure that the JDBC server for OS 2200 is running. Contact your system administrator for instructions on how to obtain this information.

To view the contents of the Office Supply Store database using Data Source Explorer

1. Expand **RDMS-OSupplyStore**.
2. Expand the database name, **Schemas**, **OSupplyStore, Tables**, and then **STORECUSTOMER**.

   The structure of the Store Customer table is displayed.
3. Right-click **STORECUSTOMER**, point to **Data**, and click **Sample Contents**.

   The **SQL Results** pane appears with the SQL select statement generated in the **Status** tab.
4. Click the **Results1** tab to view the data of the Store Customer table.

   Note: *The* ***Result1*** *tab is not visible if there is no data in the Customer table.*

Figure 4–3 illustrates the data for the Store Customer table in the SQL Results pane.

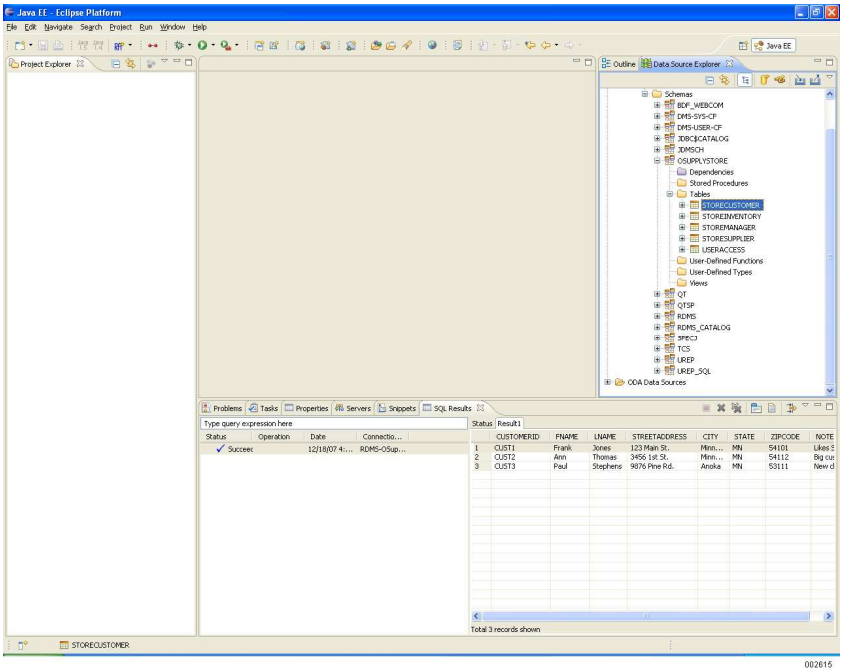

Figure 4–3. Relational Database Server Database Contents

You can also generate customized DDL scripts for the tables in the database. Refer to 3.7 for the procedure.

Section 5
Running Java Applications on the OS 2200 System

Using Eclipse IDE, you can set up Java applications on an OS 2200 system to run and debug them remotely.

5.1. Configuring Connections

To prepare for running Java applications on an OS 2200 IDE for Eclipse system, you must set up host accounts, connections, and log-in scripts.

5.1.1. Setting up Host Accounts

Host accounts are needed to log on to the OS 2200 IDE for Eclipse . To create a new host account

1. Start Eclipse IDE.
2. Click **Telnet** .

   The New Telnet connection window for managing host and connections appears.
3. Select **Configured** and click **New Connection**.

   The **Connections Settings** dialog box appears.
4. Type the host name, user-id, and password in the **Host**, **User ID**, and **Password** boxes.
5. Retype the password in the **Retype** box.
6. Optionally, select the **Save Password** check box to save the password on the system for future use of the host account.

   Saving the password causes it to persist between Eclipse IDE sessions. If you do not save the password, the password is retained only in memory and is discarded when Eclipse IDE closes. Local office policies determine whether you can save passwords on your disk. Passwords are saved in private file space, however, and another user with administrative privileges can see your password.
7. Select the **OS 2200** operating system option if it is not already selected.
8. Enter the required name in the **Connection Name** box.

   The name is displayed in the Telnet wizard.
9. Click **OK** and then click **Finish**.

10. Enter the password in the Host Prompt window that is displayed.
11. Click **OK**.

    You are now connected to the OS 2200 host.

5.1.2. Setting up Connections

Connections are associated with host accounts. Each host account can have any number of connections. To create a new connection

1. Click **Telnet** icon on the toolbar.

   The Telnet window for managing host and connections appears.
2. Select **Configured** and click **New Connection**.

   The **Connections Settings** dialog box appears.
3. Select the required account in the **Host Accounts** list.
4. Enter the required name for the account in the **Connection Name** box.

   The name is displayed in the Telnet wizard.
5. If necessary, change the default entries in the **Port Number** and **Prompt Character** boxes. (In most cases, you do not need to change these values. If the specified port is an SSL port, select the **SSL Port** check-box.)
6. If your OS 2200 IDE for Eclipse log-in has no prompts other than for user-id and password, click **OK**. If your log-in requires project-id, account, account index, and so forth, complete the procedure in 5.3 and then click **OK**.

5.1.3. Recording Log-In Scripts

Log-in scripts enable Telnet sessions to log in automatically. To record a log-in script for the host account and connection

1. Click **Record**.

   If you have an existing script, a warning is displayed asking if you are sure; if you click **OK**, existing scripts are deleted.
2. Click **OK**.

   If no scripts are needed, you receive a notice.
3. Click **OK**.
4. Answer each prompt as it appears and click **OK**.

   The prompts and your responses are recorded and attached to the connection as a script. They are also displayed in the **Telnet Connection Settings** dialog box.

5. If one of the host prompts needs to be edited manually (such as its timestamp)
   a. Select the prompt and click **Edit**.

      The host prompt is displayed in the editing area.

   b. Edit the host prompt and response as needed and click **Replace**.

      The Host prompt and response are updated with the new values.

   c. Clear the contents in the **Host Prompt** and **Response** boxes and then click **Record**.

      Otherwise, add the values present in these boxes and then click **Record**.

5.2. Starting a Telnet Session

5.2.1. Using Preconfigured Connections

To start a Telnet session using preconfigured connections (refer to 5.1) in the Telnet wizard

1. On the **File** menu, point to **New** and click **Other**.

   The **Select a wizard** dialog box appears.
2. Expand **Telnet** and click **Telnet Connection**.
3. Click **Next**.

   If you already configured a set of connections, the list of connections is displayed in **Configured Connections**.
4. Select **Configured** to use one of your preconfigured connections.
5. Select the desired connection and click **Finish**.

The resulting Telnet window automatically logs on to the system using the predefined user-id, password, and any log-in scripts that you defined. Some prompts can occur if

- You did not save the password.
- The password is expired.
- The log-in scripts are incorrect.

The Telnet wizard saves the most recent log-in entries and uses them in new Eclipse IDE sessions.

You can also create a new host account by clicking **New Connection**.

5.2.2. Creating Connections Manually

If no connections are configured or if you wish to connect to a system that is not listed

1. Launch the Telnet wizard as in 5.2.1. In step 4, leave the selection as **Manual**.
2. Type the host name.
3. Change the port number if your Telnet server uses a nonstandard port number. If the specified port is an SSL port, select the **SSL Port** check-box.
4. Select a character set from the **Character Conversion** list, if required.

   Note: *Telnet supports character set translation, which includes the Japanese character set and 7-bit character sets supported by ISO-646. When a character set is selected, any occurrences of the selected characters in Telnet traffic are translated to Unicode before being displayed.*
5. Click **Finish**.

   The **Telnet** dialog box appears.
6. Answer the prompts for user-id, password, and possibly other values, depending on the system to which you are connecting.

5.2.3. Switching between Command Line Modes

A connected Telnet session has two command line modes.

- The duplex command line mode, represented by the following double-arrow icon:

  The duplex command line sends each character to the host as it is typed. The character is then echoed by the host and placed on the screen. The only editing allowed in duplex mode is the use of backspace. The duplex command line mode is the default mode when connecting to a UNIX host and it is typically used when control characters need to be sent to the host.
- The buffered command line mode, represented by the following single-arrow icon:

  The buffered command line only transmits the command line when the Enter key is pressed. The line is buffered until the Enter key is encountered. Typical workstation editing, such as insert, delete, cut, and paste, is allowed in the middle of the line. The buffered command line mode is the default mode when connecting to an OS 2200 IDE for Eclipse host.

To access the command line modes, right-click an empty area on the Telnet console.

5.3. Preparing the OS 2200 IDE for Eclipse Java Project

5.3.1. Creating OS 2200 IDE for Eclipse Directories

To create a directory on the target OS 2200 IDE for Eclipse system using a Telnet session

1. Start a Telnet Session in Eclipse IDE (refer to 5.2).
2. On the OS 2200 IDE for Eclipse system, create a network-visible name for the CIFS directory using a CIFS utility script that is similar to the following:

   ```
   @cifsut
   cd /java
   mkdir <your-name>Workspace
   share /java/<your-name>Workspace  <your-name>WorkArea
   ```

   where <*your-name*> is your name or some other identifier that makes the folder unique on the system.
3. On your workstation, use Windows Explorer to map a network drive to the OS 2200 IDE for Eclipse shared name (<*your-name*>WorkArea) of the CIFS directory. This directory contains your Java program source code file and its class file. For example if the host name is HostA and the share name is TestWorkArea, enter the following path:

   ```
   \\HostA\TestWorkArea
   ```
4. Click **Connect using a different user name**.

   The **Connect As** dialog box appears.
5. Type your OS 2200 IDE for Eclipse user-id and password in the **User name** and **Password** boxes and click **OK**.
6. Click **Finish**.

   Note: *Every OS 2200 IDE for Eclipse project has a Telnet session associated with it, which is represented by the following icon:*

   

   The icon is enabled when an OS 2200 IDE for Eclipse project is selected.

5.3.2. Creating New Java Projects

To create a new Java project

1. On the **File** menu, point to **New** and click **Project**.

   The New Project wizard appears.
2. Select **Java Project** and click **Next**.
3. Type **DebugTest1** in the **Project name** box.
4. Select **Create project from existing source**.

5. Click **Browse**.

   The **Browse for Folder** dialog box appears.

6. Select the mapped drive for **TestWorkArea** from the list and click **OK**.

   The drive letter for the project appears in the **Directory** box.

7. Click **Finish**.

5.3.3. Creating Source Class Folders

By default, the Java source folder named src is created in the DebugTest1 project. If it is not created by default, perform the following steps to create a folder for the Java source:

1. From Eclipse IDE Package Explorer, select and right-click **DebugTest1**, point to **New**, and click **Source Folder**.

   The **New Source Folder** dialog box appears.

2. Type **src** in the **Folder name** dialog box and click **Finish**.

5.3.4. Creating Application Class Files

To create a new class file for the application

1. From Eclipse IDE Package Explorer, select and right-click **DebugTest1**, point to **New**, and click **Class**.

   The New Java Class wizard appears.

2. Type **DebugTest1** in the **Name** box.
3. Be sure **public** is selected in the **Modifiers** list.
4. Select the **public static void main(String[] args)** check box.
5. Click **Finish**.
6. Edit the main() method of the DebugTest1 class to add the lines in Figure 5–1.

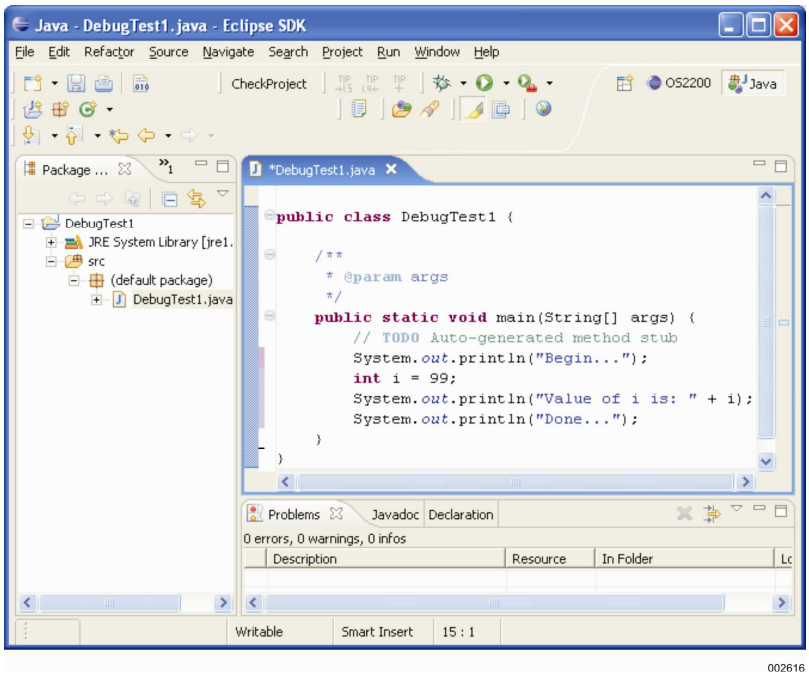

Figure 5–1. Creating a Class File

7. Save the updated file.
8. Double-click the left margin of the second println statement to add a breakpoint.

   The following symbol appears:

5.4. Running the Java Application

5.4.1. Running the Java Application Remotely on OS 2200 IDE for Eclipse

To run the Java application remotely on OS 2200 IDE for Eclipse, create the Java project in Eclipse and perform the following steps:

1. On the **File** menu, click **New** and then click **Project**.

   The New Project wizard appears.

2. Type **Java Project** in the **Wizards** box, select **Java Project** from the available options, and click **Next**.

   The New Java Project wizard appears.

3. Type the required project name in the **Project name** box. For example, **sampleProject**.

4. Click **Next** and then click **Finish**.

   The sampleProject is created.

5. Right-click the **src** folder of sampleProject you just created, click **New**, and then click **Other**.

   The New wizard appears.

6. Type **class** in the **Wizards** box, select **Class** from the Java node, and then click **Next**.

   The New Java Class wizard appears.

7. Type the required Java class name in the **Name** box (for example DebugTest1) and click **Finish**.

   The New Java class file (DebugTest1.java file in this example) is created.

8. Add the following code in the DebugTest1.java file which you created:

```
public class DebugTest1 {
    public static void main(String[] arg) {
        System.out.println("Begin Java ...");
        int i = 99;
        System.out.println("The value of i : is " + i);
        System.out.println("End of Java...");
    }
}
```

9. Right-click the sample project (created in step 3) and click on **Properties**.

   The Properties for sample Project wizard appears.

10. Browse to the location in your system under **Resources** in the wizard, open the **bin** folder, and copy the DebugTest1.class file.

11. Map the Network drive with the OS 2200 IDE for Eclipse server (RS02) and paste the DebugTest1.class file to the mapped drive.

12. Log on to the OS 2200 IDE for Eclipse server (RS02) using the Telnet icon on the toolbar.

    The RS02 pane is displayed at the bottom of the Eclipse window.

13. Type the following command to run the DebugTest1 program on RS02:

    ```
    @'java -cp file DebugTest1
    ```

    where –cp is used to set the classpath if the program is kept in a different directory.

The following output appears in the **Console** tab:

```
Begin Java ...
The value of i : is 99
End of Java..
```

5.4.2. Running the Java Application Remotely on OS 2200 IDE for Eclipse JProcessor

Perform one of the following steps to run the Java application remotely on OS 2200 IDE for Eclipse JProcessor:

- If the OS 2200 IDE for Eclipse server is mounted with JProcessor, run the following command through the Telnet window:

  ```
  @'jpjava -cp file DebugTest1
  ```

- If the OS 2200 IDE for Eclipse server is not mounted with JProcessor, run the following command through the Telnet window:

  ```
  @cifsut
  set CURRENTDIRECTORY=~/mnt/file
  @eof
  @'icmount /file /home/IC-your2200userid/mnt/file
  @'jpjava DebugTest1
  ```

  where

  - File is the directory name.
  - your2200userid is the IP address of the JProcessor.
  - DebugTest1 is the java program that you created in step 7 of Section 5.4.1 (this file is saved on the OS 2200 IDE for Eclipse server, RS02).

For more information on mounting the JProcessor, refer to the *ClearPath Specialty Engine for OS 2200 Installation and Servicing Guide*.

5.5. Debugging the Java Application

5.5.1. Debugging the Java Application Remotely from OS 2200 IDE for Eclipse Server

To debug the java application remotely on OS 2200 IDE for Eclipse server

1. Enter the following statement in the Telnet window:

   ```
   @'java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspen
   d=y -cp file DebugTest1
   ```

   where

   - -Xdebug enables debugging support in Virtual Machine
   - The Java Debug Wire Protocol (jdwp) defines the format of information and requests transfer between the process being debugged and the debugger.
   - address is the transport address for the connection
     - If server = n, attempt to attach to debugger application at this address.
     - If server = y, listen for a connection at this address.
     - If suspend = y, suspend this Java Virtual Machine before the main class loads.

     The port address must be unique (8000, in this example). If a port address is in use, a verbose ER ABORT$ message appears.

   - DebugTest1 is the Java program that you created in step 7 of Section 5.4.1

   The output window should display the following:

   ```
   Listening for transport dt_socket at address: 8000
   ```

2. Open the DebugTest1.java file under sampleProject from Eclipse and verify its code. The code should be similar to the following:

   ```
   public class DebugTest1 {
       public static void main(String[] arg) {
           System.out.println("Begin Debug ...");
           int i = 99;
           System.out.println("The value of i : is " + i);
           System.out.println("End of Debug...");
       }
   }
   ```

3. Right-click the left margin of the DebugTest1.java file code and click **Toggle Breakpoint** to add a breakpoint.
4. On the **Run** menu, click **Debug Configurations**.

   The Debug Configurations window appears.

5. Double-click the **Remote Java Application** node in the left pane and then select **Connection Type Standard (Socket Attach)** in the right side of the pane.
6. Type the host system name in the **Host** box.

   If your OS 2200 IDE for Eclipse system uses JProcessor, type the IP address of JProcessor system instead of the OS 2200 IDE for Eclipse system.
7. Type **8000** in the **Port** box, the same port address as specified in step 1.
8. Click **Apply** and then click **Debug**.
9. Click **Yes** in the Confirm Perspective Switch window.

   The Eclipse IDE switches to the Debug perspective.

After you enter the debugging information, you can reuse the remote debug configuration to restart the debugging. Debug as necessary and click the following icon on the toolbar or press F8 to resume:

The debug output appears on the **Console** tab, as shown in Figure 5–2.

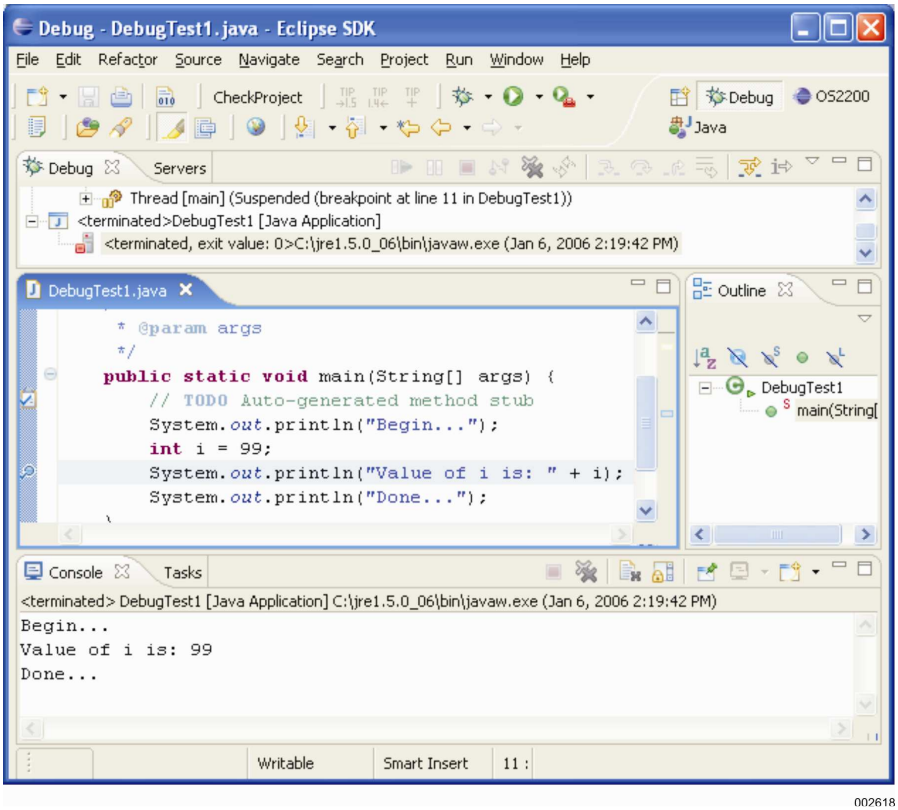

Figure 5–2. Eclipse IDE Debug Output

5.5.2. Debugging the Java Application Remotely from OS 2200 JProcessor

Before you begin debugging, determine the IP address of the JProcessor that is connected to the OS 2200 IDE for Eclipse server. Enter the following statement in the Telnet or INFOConnect window:

```
@icadmin
```

The window will switch to the Interconnect ICADMIN Command Choices. Enter the following command to determine the IP Address:

```
list_processor
```

The output window displays the IP address of the JProcessor connected to the OS 2200 IDE for Eclipse server as illustrated in Figure 5–3:

```
 Type         Num     IC-Mode     host:port                  Qualifier\J-name
JPROCESSOR     1      DEFAULT     192.61.252.148:22719       JPR1
JPROCESSOR     3      DEFAULT     192.61.229.93:22719        JPR3
JPROCESSOR     4      DEFAULT     192.61.229.94:22719        JPR4
JPROCESSOR     5      DEFAULT     192.61.213.5:22719         JPR5
```

Figure 5–3. JProcessor IP Address

To debug the java application remotely on OS 2200 IDE for Eclipse JProcessor

1. Enter the following statement in the Telnet window:

   ```
   @'jpjava -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,susp
   end=y -cp file DebugTest1
   ```

2. Follow steps 2 through 5 of section 5.5.1 and type your OS 2200 IDE for Eclipse system JProcessor IP address (that you determined before debugging) in the **Host** box.
3. Type **8000** in the **Port** box, the same port address as specified in step 1.
5. Click **Apply** and then click **Debug**.
6. Click **Yes** in the Confirm Perspective Switch window.

   The Eclipse IDE switches to the Debug perspective.

After you enter the debugging information, you can reuse the remote debug configuration to restart the debugging. Debug as necessary and click the following icon on the toolbar or press F8 to resume:

The following debug output appears on the **Console** tab:

```
Listening for transport dt_socket at address: 8000
   Begin Debug ...
   The value of i : is 99
    End of Debug...
```

5.6. Troubleshooting Errors

The following errors can occur when running Java applications remotely on an OS 2200 IDE for Eclipse system:

- A launching error message appears stating that the connection to the remote VM failed. Possible reasons and solutions are
  - The remote application might not be ready to accept the connection from the Eclipse IDE debugger. Wait a few seconds and try again.
  - You might not have configured the OS 2200 or Eclipse IDE correctly. Correct the Eclipse IDE debugging parameters and check the start parameters for the JVM on the OS 2200 site.
- The program is not waiting on the port you specified.
- The JVM is not configured for the host IP address that you are attempting to use.

Section 6
Creating an Enterprise Application Development Model

An Enterprise application is a business application which is developed to satisfy the business needs of the enterprise. It can be deployed on a variety of platforms across networks. Java EE technology provides a framework for creating a simple development model.

6.1. Overview of J2EE Technology and Concepts

A simple development model for enterprise applications that uses J2EE technology is a component-based application model. The components use services that are provided by the container. Without the container, these services typically need to be incorporated in the application code.

J2EE technology is multitiered architecture for implementing enterprise-class applications such as

- Web applications
- Client/server applications

The J2EE model is not ideal for all scenarios. For example, a light-weight Java technology solution, such as servlets or JavaServer Pages (JSP), can be a better solution for a small scale application.

6.2. J2EE Components

J2EE applications consist of different components. A J2EE component is a self-contained functional software unit that

- Is assembled into a J2EE application with its helper classes and files
- Communicates with other components in the application

The J2EE specification defines the following primary J2EE components:

- Application clients and Web clients, including applets, are components that run on the client.
- Java servlet and JavaServer Pages (JSP) technology components are Web components that run on the Web server.
- Enterprise JavaBeans (EJB) components (also known as enterprise beans) are business components that run on the application server.

Figure 6–1 illustrates these components and environments.

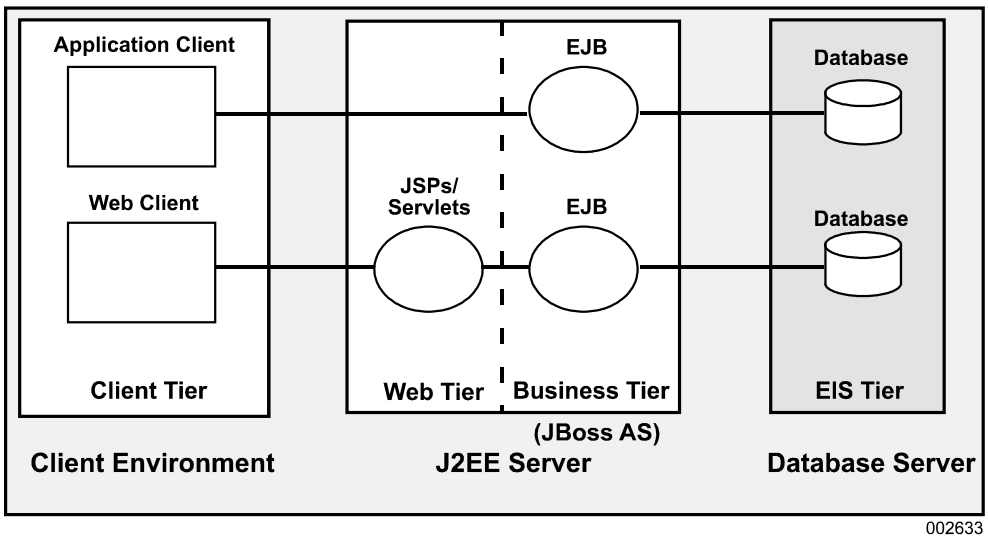

Figure 6–1. J2EE Components

J2EE components are assembled into a J2EE application, verified to be well formed and in compliance with the J2EE specification, and deployed to production, where they are run and managed by a J2EE application server, such as JBoss AS.

6.3. J2EE Services and Supporting Technologies

In addition to the primary components, other standard services and supporting technologies are

- Java Database Connectivity (JDBC) technology. Provides access to relational database systems.
- Java Transaction API (JTA) and Java Transaction Service (JTS). Provide transaction support for Java EE components.
- Java Message Service (JMS). Provides asynchronous communication between J2EE components.
- Java Naming and Directory Interface (JNDI). Provides naming and directory access.

6.4. J2EE Distributed Architecture

All J2EE applications implement a distributed architecture in which

- An object is associated with a logical name.
- A naming service accepts advertisements from logically named EJB components.
- Client components request references to service components using the name.

Figure 6–2 illustrates these relationships, with the process flow described in 6.11.1.

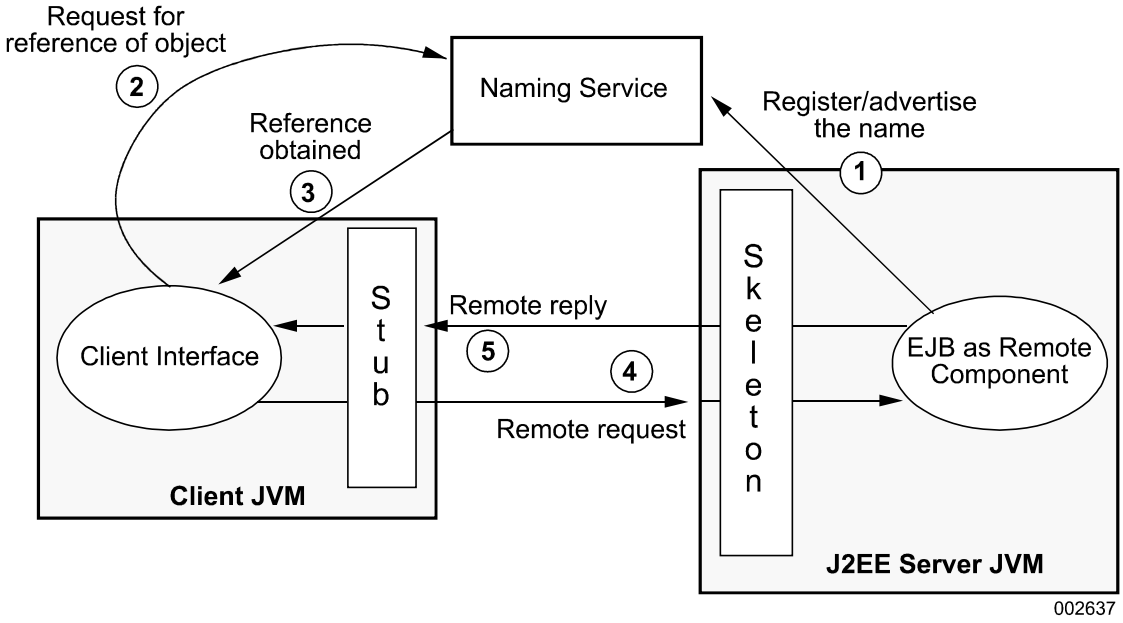

Figure 6–2. J2EE Distributed Architecture

6.4.1. Process Flow

The process flow of a distributed architecture is

1. A remote object advertises its availability with the naming service using a logical name.
2. A client requests the object references by looking for an object by its advertised name.
3. The naming service translates the name to the physical location of the object in the J2EE environment.
4. When the client gets the reference to a remote component, the client performs the necessary operations (sends requests to) on that object.
5. The client processes the reply, as appropriate.

The run-time system handles the distributed communication between the remote objects, which includes serialization and deserialization of parameters. (In RPC terminology, serialization is the same as marshalling, and deserialization is the same as unmarshalling.)

6.4.2. Naming Services

Some of the naming services that are used in distributed systems are

- Remote Method Invocation (RMI). For Java-only implementations; JBoss AS uses RMI as its naming service.
- Common Object Request Broker Architecture (CORBA)
- Lightweight Directory Access Protocol (LDAP)
- Domain Name System
- Network Information Services (NIS)

6.4.3. Java Naming and Directory Interface Architecture

Java EE uses the Java Naming and Directory Interface (JNDI) API to generically access naming and directory services using Java technology. The JNDI API resides between an application and a naming service and makes the underlying naming service implementation transparent to application components.

A client can look up references to EJB components and other resources using a naming service. The client code remains unchanged, regardless of which naming service is used or on what technology it is based.

6.5. Java EE Components

A Java EE application is packaged into one or more standard units for deployment to any Java EE platform-compliant system. Each unit contains a functional component or components (enterprise bean, JSP page, servlet, applet, and so on).

An optional deployment descriptor that describes its content once a Java EE unit has been produced is ready to be deployed. Deployment typically involves using a platform's deployment tool to specify location-specific information, such as a list of local users that can access it and the name of the local database. Once you deploy an application on a local platform, you can run the application.

A Java EE application is delivered in an Enterprise Archive (EAR) file– a standard Java Archive (JAR) file with an .ear extension. You can use the EAR files and modules to assemble a number of different Java EE applications with the help of some of the JEE components.

An EAR file contains Java EE modules and deployment descriptors. A deployment descriptor is an XML document with an .xml extension that describes the deployment settings of an application, a module, or a component. Since deployment descriptor information is declarative, it can be changed without the need to modify the source code. At run time, the Java EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.

Figure 6–3 illustrates the JAVA EE components.

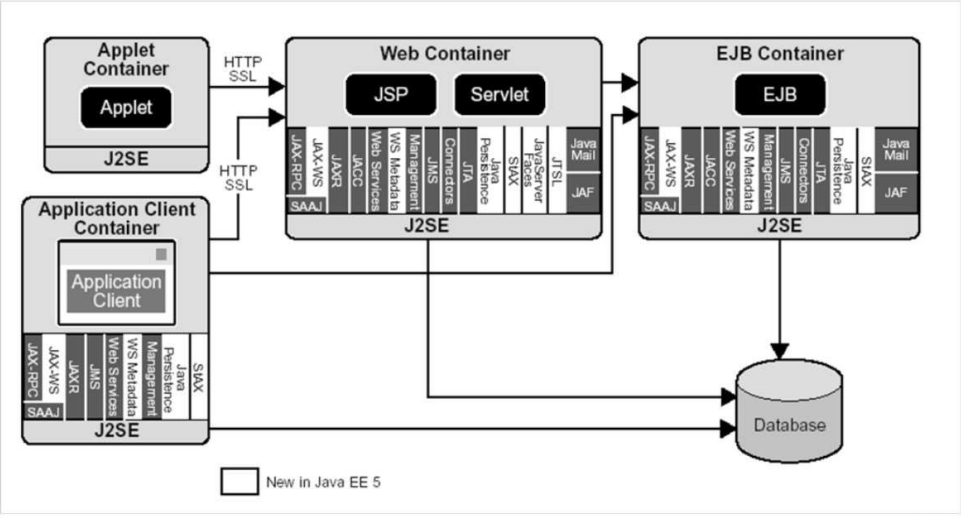

Figure 6–3. Java EE Components

A Java EE module consists of one or more Java EE components for the same container type and one component deployment descriptor of that type. An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean. A Java EE module without an application deployment descriptor can be deployed as a stand-alone module.

The four types of Java EE modules are as follows:

- **EJB modules** contain class files for enterprise beans and an EJB deployment descriptor. EJB modules are packaged as JAR files with a .jar extension.
- **Web modules** contain servlet class files, JSP files, supporting class files, GIF and HTML files, and a Web application deployment descriptor. Web modules are packaged as JAR files with a .war (Web ARchive) extension.
- **Application client modules** contain class files and an application client deployment descriptor. Application client modules are packaged as JAR files with a .jar extension.
- **Resource adapter modules** contain all Java interfaces, classes, native libraries, and other documentation, along with the resource adapter deployment descriptor. Together, these implement the Connector architecture (J2EE Connector Architecture) for a particular EIS. Resource adapter modules are packaged as JAR files with a .rar (resource adapter archive) extension.

The Java EE architecture provides configurable services, application components within the same Java EE application can perform differently based on where they are deployed. For example, an enterprise bean can have security settings that allow the bean a certain level of access to database data in one production environment and another level of database access in another production environment.

The container also manages nonconfigurable services such as enterprise bean and servlet lifecycles, database connection resource pooling, data persistence, and access to the Java EE platform APIs.

6.5.1. Java EE APIs

Java EE includes the following API specifications:

- Enterprise JavaBeans Technology
- Java Servlet Technology
- JavaServer Pages Technology
- JavaServer Pages Standard Tag Library
- JavaServer Faces
- Java Message Service API
- Java Transaction API
- JavaMail API
- JavaBeans Activation Framework
- Java API for XML Processing
- Java API for XML Web Services (JAX-WS)
- Java Architecture for XML Binding (JAXB)
- SOAP with Attachments API for Java
- Java API for XML Registries
- Java Database Connectivity (JDBC) API
- Java Persistence API (JPA)
- Java Naming and Directory Interface (JNDI)
- Java Authentication and Authorization Service

6.5.2. Java EE Communication Technologies

Communication technologies provide mechanisms for communication between clients and servers and between collaborating objects hosted by different servers.

The Java EE specification requires support for the following types of communication technologies:

- Internet protocols
- Remote Method Invocation Protocols
- Object Management Group Protocols
- Messaging technologies

  It provides a way to asynchronously send and receive messages. The Java Message Service API provides an interface for handling asynchronous requests, reports, or events that are consumed by enterprise applications.

6.6. Java EE Clients

Java EE clients are Web clients or application clients.

6.6.1. Web Clients

A Web client consists of

- Dynamic Web pages that contain various types of markup language (such as HTML or XML), which are generated by Web components running in the Web tier
- A Web browser that renders the pages received from the server

Thin Clients

Web clients are often designed as thin clients. Thin clients pass input and requests from users to a server and return results. Thin clients do not perform heavyweight operations, such as querying databases, executing complex business rules, or connecting to legacy applications. These operations are performed by enterprise beans that execute on the Java EE server, where they can make use of the security, speed, services, and reliability of Java EE server-side technologies.

Embedded Applets

Web pages that are received from the Web tier can include embedded applets. An applet is a small client application that is written in the Java programming language and executes in the Java virtual machine (JVM) that is installed in the Web browser.

To execute applets successfully in the Web browser, client systems need the Java plug-in and possibly a security policy file.

Web Components

Web components (servlets and JSPs) are often preferred over applets for creating Web clients because no plug-ins or security policy files are needed on the client systems. Web components provide a means to separate application logic from Web page design, allowing a modular application design. Refer to 6.5 for more information.

6.6.2. Application Clients

An application client runs on a client machine and provides a way to perform tasks that require a richer user interface than a markup language can provide. An application client

- Normally has a graphical user interface (GUI) that is created using the Swing or Abstract Window Toolkit (AWT) APIs
- Can have a command-line interface
- Can directly access enterprise beans that are running in the business tier

If a Web client is needed, an application client can open an HTTP connection to establish communication with a servlet that is running in the Web tier.

6.7. Web Components

Java EE Web components are servlets or JavaServer Pages (JSP), which are defined as follows:

- Servlets are Java programming language classes that dynamically process requests and construct responses.
- JSPs are text-based documents that execute like servlets but allow a natural language approach to creating static content.

The Web tier can include JavaBeans components to manage the user input and send that input to enterprise beans that are running in the business tier for processing. HTML pages, applets, and server-side utility classes are bundled with Web components during application assembly, but the Java EE specification does not consider them to be Web components.

Figure 6–4 illustrates Web components and communication between them.

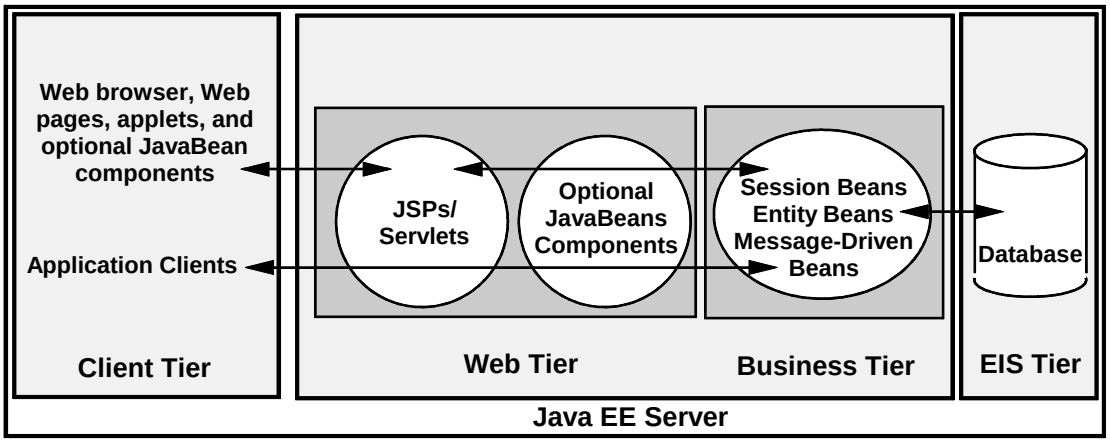

Figure 6–4. Web Components and Communication

6.8. Business Components

The enterprise beans that run in the business tier handle business operations for a particular business domain, such as banking or finance. Some enterprise beans

1. Receive data from client programs.
2. Process the data (if necessary).
3. Send the data to the enterprise information system (EIS) tier, which stores it in a database.

Other enterprise beans

1. Retrieve data from the database.
2. Process the data (if necessary).
3. Send the data to client programs.

6.8.1. Types of Enterprise Beans

Enterprise beans can be

- Session beans (stateless and stateful). A transient conversation with a client. When the client finishes executing, the session bean and its data are gone.
- Entity beans (bean-managed and container-managed). Persistent data stored in one row of a database relation or table. If the client terminates or the server shuts down, the underlying services ensure that the entity bean data is saved.
- Message-driven beans. Combination of features of a session bean and a Java Message Service (JMS) message listener, which allows a business component to receive JMS messages asynchronously.

The procedures in this guide create and use all bean types in the Office Supply Store (refer to Table 1–2).

6.8.2. Enterprise Beans Versus JavaBeans

The Java EE specification does not include JavaBeans as Java EE components because JavaBeans are different from enterprise beans. Both server and client tiers can use JavaBeans component architecture to manage the communication between an application client or applet and components running on the Java EE server or between server components and a database. JavaBeans have instance variables and accessor and mutator methods to access properties of beans.

Enterprise beans (EJB components) are used only in the business tier as part of the server tier.

6.9. Enterprise Information System Tier

The enterprise information system (EIS) tier handles enterprise information system software and includes enterprise infrastructure systems such as

- Enterprise resource planning (ERP)
- Mainframe transaction processing
- Database systems
- Legacy information systems

Java EE application components can access enterprise information systems for database connectivity.

6.10. Java EE Containers

Java EE containers provide access to the underlying services of the Java EE server environment. In a traditional environment, application developers write code to perform services such as transaction processing, state management, multithreading, and resource pooling. Java EE containers provide these services, thereby allowing developers to concentrate on solving business problems.

Containers are the interface between components and the low-level platform-specific functionality that supports the components. Before Web clients, enterprise beans, or application client components can be executed, components must be assembled into a Java EE application and deployed into a container. The assembly process involves specifying container settings for each component in the Java EE application and for the Java EE application itself. Container settings customize the underlying support provided by the Java EE server, which includes services such as Java Naming and Directory Interface (JNDI), security, and transaction management.

Some of the main points are as follows:

- The Java EE security model lets you configure a Web component or enterprise bean so that system resources are accessed only by authorized users.
- The Java EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.
- The JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access these services.
- The Java EE remote connectivity model manages low-level communications between clients and enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

The Java EE server provides Enterprise JavaBeans (EJB) containers and Web containers, as follows:

- EJB containers manage the execution of enterprise beans for Java EE applications.
- Web containers manage the execution of JSP and servlet components for Java EE applications.

The following additional containers reside on the client machine and are not part of the Java EE server:

- Application client containers (typically, Java Runtime Environment) manage the execution of application client components.
- Applet containers (typically, a Java-enabled Web browser) manage the execution of applets.

Figure 6–5 illustrates where the containers reside.

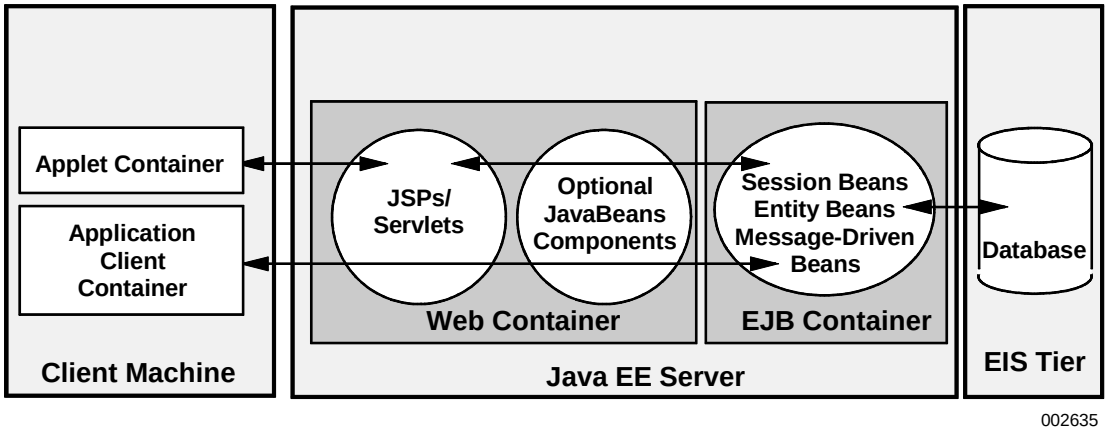

Figure 6–5. Java EE Containers and Additional Containers

6.11. Packaging for Deployment

To deploy a Java EE application after its components are developed, it is packaged into special archive files that contain

- Relevant class files.
- XML deployment descriptors. Information specific to each bundled component; a mechanism for configuring application behavior at assembly or deployment time.

XML deployment descriptors are bundled in different archive types for different component types, as follows:

- Web archive (war) files. Web components are archived in a war file, which contains servlets, JSPs, and static components, such as HTML and image files. The war file contains classes and files that are used in the Web tier, along with a Web component deployment descriptor.

- Java archive (jar) files.
  - Business components are archived in a jar file, which contains an EJB deployment descriptor, remote and object interface files, and helper files that are required by the EJB component.
  - Client-side class files and deployment descriptors are archived in a jar file, which makes up the client application.
- Enterprise archive (ear) files. A Java EE application is bundled in an ear file, which contains the whole application, along with a deployment descriptor that provides information about the application and its assembled components.

Figure 6–6 illustrates component packaging.

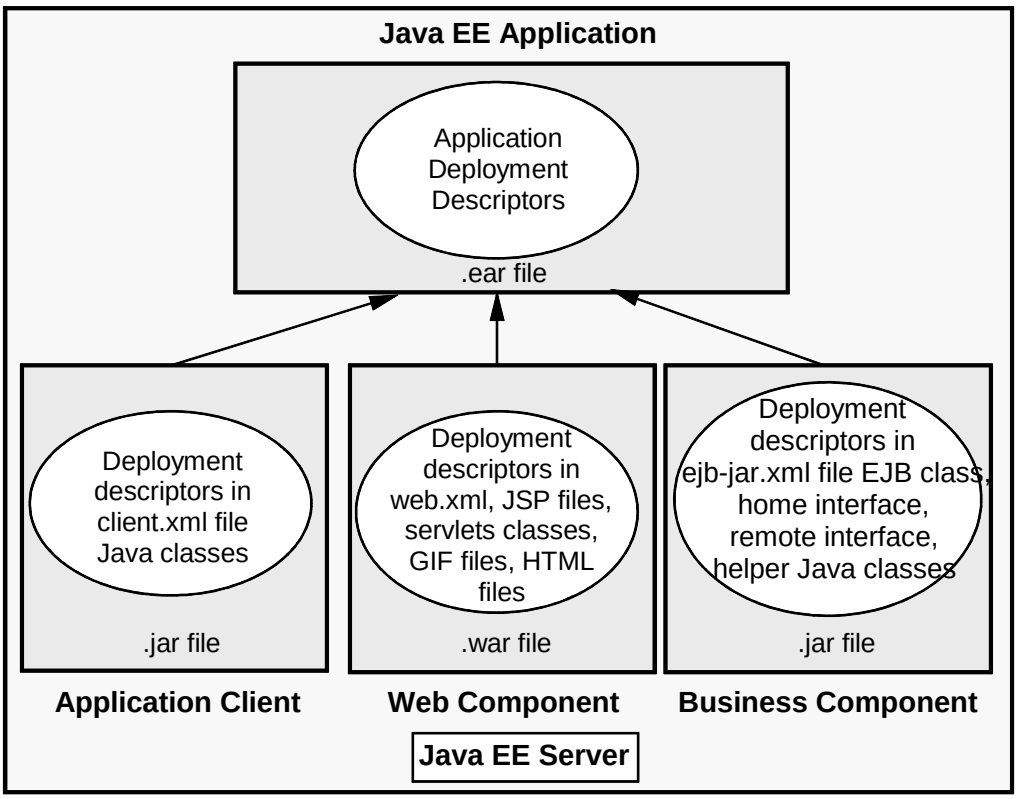

Figure 6–6. Java EE Component Packaging

6.12. Java EE Platform Roles

The process of building the different components of a Java EE application involves the following roles to develop, deploy, and manage an enterprise application:

- Application component provider. Develops the reusable components of a Java EE application (Web components, enterprise beans, applets, and application clients) for use in Java EE applications.
- Application assembler. Takes all building blocks from the application component; installs and deploys components in a Java EE environment or Java EE server.
- System administrator. Configures and administers computing systems in an enterprise.
- Tool provider. A vendor used to develop, package, and deploy Java EE applications.

Roles can be assigned to either a single person or an organization.

Section 7
Creating Stateless Session Beans

A session Enterprise JavaBean (EJB) component is a transient conversation with a client. A session bean and its data exist only while the client is executing. Transactions that use stateless beans must contain all the parameters required to carry out a transaction in a single call. Stateless session beans do not retain state information once they finish.

7.1. Accessing Office Supply Store

The Office Supply Store application implements UserAccess, a session bean (refer to 6.6.1). UserAccess oversees all processing for the application. UserAccess is a transient object through which clients access the underlying entity beans for specific services.

Figure 7–1 illustrates the flow of information and processing in the Office Supply Store application.

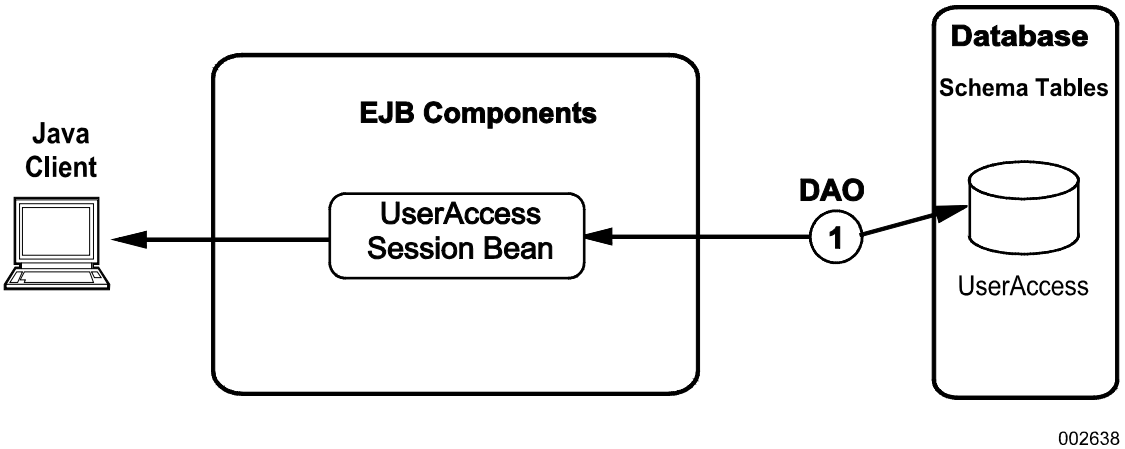

Figure 7–1. Stateless Session Bean in Office Supply Store

7.1.1. Session Facade Pattern

A session facade is an application pattern in which enterprise beans encapsulate business logic and business data and expose their interfaces. The session facade manages business objects and provides a uniform business service abstraction to clients in the presentation layer, while the business object implementation is hidden in the lower-level beans.

In many applications, a session bean provides access to the application business methods. A session bean encapsulates business logic and acts as an interface to the lower-level EJB components. In the Office Supply Store, UserAccess is a session bean that accesses entity beans and message-driven beans, which are developed in the following sections of this guide.

7.1.2. Authenticating Users

The Office Supply Store requires that all customers, suppliers, and managers have a user name and password to access the services of the application. Each client application must sign on and be verified for the system.

UserAccess authenticates the user by communicating with the database using a data access object (DAO) that encapsulates some Java Database Connectivity (JDBC) code. A DAO has all attributes (fields) and behaviors (methods) that correspond to the bean for which it is being used.

UserAccess uses a method for authentication named verifyUser, which takes two string parameters (username and password) as input. verifyUser returns storeAccessID if authentication is successful; otherwise, it returns null. Client applications check the returned storeAccessID and continue only if storeAccessID is not null.

7.1.3. Bean Business Methods

The method verifyUser is a business method. A client application can see (invoke) only business methods on a bean.

7.1.4. Remote and Local Access to Beans

UserAccess supports a remote EJB interface, which means that clients can reside in different JVMs, possibly on different machines. Remote is the original type of EJB, so the name “Remote” does not appear in the interface names. Local interfaces have “Local” in the name.

7.1.5. Java Programming Using DAO

Components in this example use the data access object UserAccessDAO and, in another section, StoreCustomerDAO interfaces.

Implementation classes UserAccessDAOImpl and StoreCustomerDAOImpl for accessing the database are also created (refer to 7.2.4 and 8.2.2).

7.2. Tasks

Complete the following tasks to create a stateless session bean EJB component. You can do the identified modifications in any order, but the bean does not work correctly until they are all complete.

1. Create a Java EE project.
2. Create a stateless session bean structure named UserAccess.
3. Add a business method named verifyUser in UserAccessSessionBean.
4. Create the DAO interface for verifyUser named verifyThisUser.
5. Implement the verifyThisUser method in the DAO implementation class, UserAccessDAOImpl.
6. Create a test client named SessionTestClient.
7. Run the test client application to test the UserAccess bean.

Refer to the following subsections for detailed procedures for each task.

7.2.1. Creating a Java EE Project

Refer to the instructions in 2.2 to create a Java EE project with a simple EJB and a test client, using the following information:

- Project name is OfficeSupplyStore.
- Application client module is OfficeSupplyStoreClient.
- EJB module is OfficeSupplyStoreEJB.

Open the Java EE perspective. It is normal for incomplete EJB projects to contain errors. As you develop more EJB components, the errors are resolved.

The Java perspective provides an alternate way to see the created project more clearly.

7.2.2. Creating Stateless Session Bean Structures

Refer to the instructions in 2.3 to create a stateless session bean, using the following information:

- Project name is OfficeSupplyStoreEJB.
- Folder is \OfficeSupplyStoreEJB\ejbModule.
- Java package is us.com.unisys.session.
- Class name is UserAccessBean.
- Superclass is java.lang.Object.

This creates the basic structure of a session bean.

The following paragraphs identify several modifications that are needed to complete the bean. A completed version of the code is at the following location:

```
...\examples\7\OfficeSupplyStoreEJB\ejbModule\us\com\unisys\session\
UserAccessBean.java
```

Class

The EJB creation wizard creates the UserAccessBean class, which implements the UserAccessRemote as illustrated in the following code:

```
public class UserAccessBean implements UserAccessRemote
```

Annotation in EJB 3.0

The following code illustrates the annotation in EJB 3.0.

```
@Stateless
@Remote
public class UserAccessBean implements UserAccessRemote {
Write the business code here..
}
```

7.2.3. Using Dependency Injection through Resource Name

Deployment Descriptors with Annotations

The deployment descriptors must be manually inserted to invoke annotation in the definition block. The following descriptor defines the data source details for the application server (JBoss AS).

```
* Resource (mappedName="DefaultDS") DataSource ds;
```

Adding Business Methods

The EJB wizard inserted a method called create as a placeholder for the location of various methods. Remove the create method and insert the business method verifyThisUser. This example does not need any logic here, but it does need to expose the interface as remote, which is done by adding the annotation name @Remote. This business method also needs to invoke PersistentContext, which communicates with the database.

```
/**
* @Remote
*
*/
public String verifyThisUser(String username, String password){
    System.out.println("Entering UserAccessBean.verifyUser()");
    return null;
}
```

7.2.4. Making a Stateless Session Bean as Java Persistence and Creating a POJO Class

PersistentContext is a set of entities such that for any persistent identity there is a unique entity instance. Entities are managed in persistent context. The EntityManager has the access to datastore resources and controls the lifecycle of entities.

To make a session bean as Java persistence

1. In the ejbModule of the OfficeSupplyStoreEJB project, right-click the **META-INF** folder, point to **New**, and click **other**.

   The New wizard appears.

2. Type **XML** in the **Wizards** box and select **XML File** from the XML node.
3. Click **Next**.

   The New XML File wizard appears.

4. Type **persistence.xml** in the **File Name** box and click **Next**.
5. Add the following content to the persistence.xml file:

```
<persistence-unit name="manager1" >
    <jta-data-source>java:DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.HSQ
LDialect"/>
    </properties>
  </persistence-unit>
</persistence>
```

6. From the **us.com.unisys.session** package, open the UserAccessBean.java file and add the following code:

```
@PersistenceContext(unitName = "manager1")
  EntityManager em;
   public Collection<USERACCESS> verifyThisUser(String USERNAME,String PA
SSWORD) {
           System.out.println("userName is *******"+USERNAME);
    storeList = em.createQuery("from USERACCESS p where USERNAME = ?1 AND
  PASSWORD=?2").setParameter(1, USERNAME).setParameter(2, PASSWORD)
                   .getResultList();
           return storeList;
       }
```

Creating the UserAccess POJO class

To create the UserAccess POJO class

1. Right-click the **us.com.unisys.session** package, click **New**, and then click **Other**.

   The New wizard appears.
2. Type **class** in the **Wizards** box and select **Class** from the Java node.
3. Click **Next**.
4. Type **UserAccess** in the **Class name** box.
5. Click **Finish**.
6. Add the following getter and setter methods for all the columns available in the UserAccess table in the **UserAccess** java file:

```
@Entity
    @Table(name = "USERACCESS")
    @EntityListeners(USERACCESS.class)
    public class USERACCESS implements Serializable {
    private static final long serialVersionUID = 1L;
    String STOREACCESSID;
    String USERNAME;
    public String getSTOREACCESSID() {
        return STOREACCESSID;
    }
    public void setSTOREACCESSID(String sTOREACCESSID) {
        STOREACCESSID = sTOREACCESSID;
    }
    public String getUSERNAME() {
        return USERNAME;
    }
    public void setUSERNAME(String uSERNAME) {
        USERNAME = uSERNAME;
    }
    public String getPASSWORD() {
        return PASSWORD;
    }
    public void setPASSWORD(String pASSWORD) {
        PASSWORD = pASSWORD;
    }
    String PASSWORD;
    long id;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
```

```
    }
    public USERACCESS() {
        super();
    }
    public USERACCESS(String STOREACCESSID, String USERNAME,
            String PASSWORD) {
        this.STOREACCESSID = STOREACCESSID;
        this.USERNAME = USERNAME;
        this.PASSWORD = PASSWORD;
    }
}
```

Generated Classes

EJB interfaces and helper classes are generated under **us.com.unisys.session** package. The following files are generated:

- UserAccessRemote.java is the remote object interface.
- UserAccessBean is the bean class for EJB.
- UserAccessLocal.java is the local object interface.

7.2.5. Creating a Test Client

To create a test client you need to modify the OfficeSupplyStoreClient project to include the EJB project. To modify the OfficeSupplyStoreClient project

1. Right-click the OfficeSupplyStoreClient project and alter the build path properties to include OfficeSupplyStoreEJB project.
2. Follow the instructions in 2.7 to create a new public class named **SessionTestClient** in **OfficeSupplyStoreClient/appClientModule** in the package **us.com.unisys.client**.

The following paragraphs identify how to finish implementing the class. A completed version of the code is at the following location:

```
...\examples\7\OfficeSupplyStoreClient\appClientModule\us\com\unisys\client\
SessionTestClient.java
```

The following imports are needed:

```
import java.sql.SQLException;
import java.util.Collection;
import java.util.Iterator;
import java.util.Properties;
import javax.ejb.Stateless;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
import us.com.unisys.session.USERACCESS;
import us.com.unisys.session.UserAccessRemote;
```

The following code is used for looking up the bean at the specified address and port number. To access a server other than the desktop, change the "127.0.0.1" and possibly the port number.

```
@Stateless
public class SessionTestClient {
    private static UserAccessRemote userAccessRemote;
    // @EJB static UserAccessRemote userAccessRemote;
    static DataSource ds = null;
    static InitialContext ctx = null;
    public static void main(String[] args) throws SQLException {
        try {
            Properties props = new Properties();
            props.setProperty("java.naming.factory.initial",
                  "org.jnp.interfaces.NamingContextFactory");
            props.setProperty("java.naming.factory.url.pkgs",
                  "org.jboss.naming");
        props.setProperty("java.naming.provider.url", "127.0.0.1:1099");
            ctx = new InitialContext(props);
            userAccessRemote = (UserAccessRemote) ctx
                  .lookup("OfficeSupplyStore/UserAccessBean/remote");
            Collection<USERACCESS> store = userAccessRemote.verifyThisUser(
                  "sa", "sa");
            for (Iterator i = store.iterator(); i.hasNext();) {
                USERACCESS e = (USERACCESS) i.next();
                System.out.println("storeaccessid is ****** "
                      + e.getSTOREACCESSID());
            }
        } catch (NamingException e) {
        e.printStackTrace();
        }
    }
}
```

7.2.6. Creating the hsql-ds.xml File

Create the hsql-ds.xml file for HSQL database and place it in the **deploy** folder of JBoss with the OfficeSupplyStore.ear file. The following example illustrates the hsql-ds.xml file for HSQL and OS 2200 RDMS databases:

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
<local-tx-datasource>
<jndi-name>DefaultDS</jndi-name>
<use-java-context>false</use-java-context>
<connection-url>jdbc:hsqldb:${jboss.server.data.dir}${/}hypersonic${/}localD
B</connection-url>
<driver-class>org.hsqldb.jdbcDriver</driver-class>
<user-name>sa</user-name>
<password></password>
```

```
<min-pool-size>5</min-pool-size>
<max-pool-size>20</max-pool-size>
<idle-timeout-minutes>0</idle-timeout-minutes>
<track-statements/>
<security-domain>HsqlDbRealm</security-domain>
<prepared-statement-cache-size>32</prepared-statement-cache-size>
<metadata>
<type-mapping>Hypersonic SQL</type-mapping>
</metadata>
<depends>jboss:service=Hypersonic,database=localDB</depends>
</local-tx-datasource>
<mbean code="org.jboss.jdbc.HypersonicDatabase" name="jboss:service=Hyperson
ic,database=localDB">
<attribute name="Database">localDB</attribute>
<attribute name="InProcessMode">true</attribute>
</mbean>
<local-tx-datasource>
   <jndi-name>jdbcunisysos2200rdms</jndi-name>
<connection-url>jdbc:rdms:host=rs02.rsvl.unisys.com;port=1544;schema=OSupply
Store;storagearea=RDMS_AUTO_3;varchar=varchar
</connection-url>
        <driver-class>com.unisys.os2200.rdms.jdbc.RdmsDriver</driver-class>
       <user-name>elango</user-name>
        <password>unisys</password>
       <max-pool-size>5</max-pool-size>
        <min-pool-size>1</min-pool-size>
        <prepared-statement-cache-size>200</prepared-statement-cache-size>
        <idle-timeout-minutes>1000</idle-timeout-minutes>
       <metadata>
            <type-mapping>RDMSOS2200</type-mapping>
        </metadata>
    </local-tx-datasource>
</datasources>
```

7.2.7. Testing the Bean

Before running the test client to test the bean

1. Define JBoss AS in the Servers window (refer to 2.5).
2. Deploy the OfficeSupplyStore project to the server (refer to 2.11).
3. Start JBoss AS (refer to 2.6).

To run SessionTestClient as a Java application; right-click **SessionTestClient**, point to **Run As**, and click **Java Application**.

If errors appear indicating that SupplyStoreDAOimpl cannot be found, try restarting the Eclipse IDE (and restart JBoss AS).

Trace lines from the server appear in the **Console** pane. To see the client trace output, open an additional console pane.

The two panes are similar to Figure 7–2 and Figure 7–3.

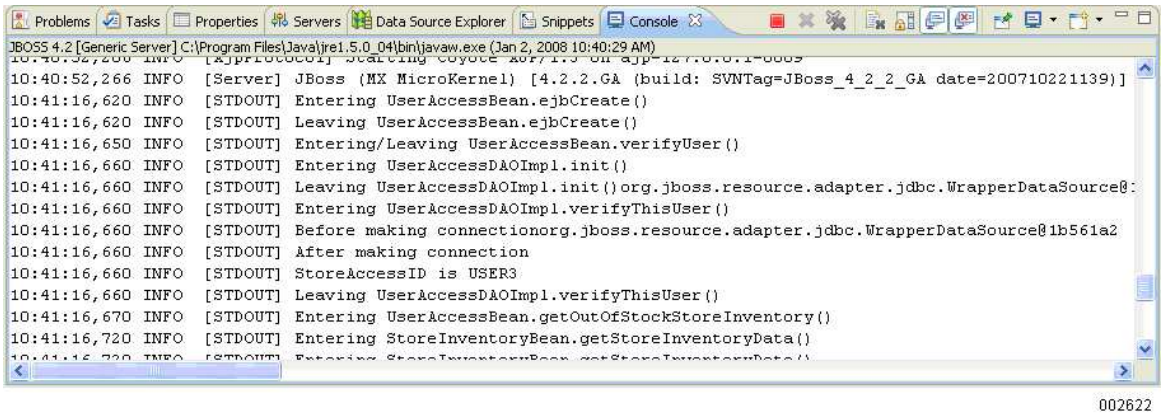

Figure 7–2. Server Trace Lines

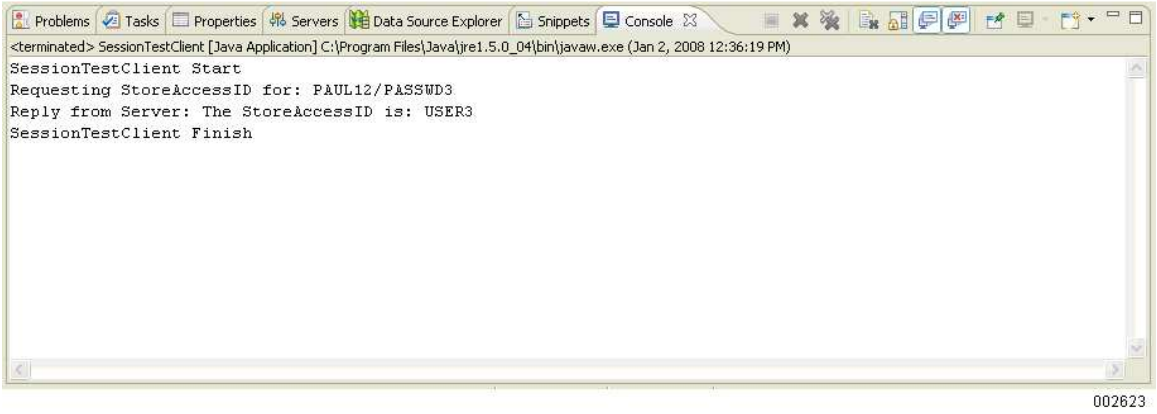

Figure 7–3. Client Trace Output

Section 8
Creating Bean-Managed Persistence Entity Beans

A bean-managed persistence (BMP) entity bean is an EJB component in which database access is controlled manually. The developer explicitly codes database calls in the bean itself, which enhances flexibility in how data is read and written.

8.1. Accessing Office Supply Store

The Office Supply Store application uses two BMP entity beans, as illustrated in Figure 8–1. The StoreCustomer bean stores the details of customers of the Office Supply Store. The optional StoreManager bean stores the details about managers of the Office Supply Store.

Both StoreCustomer and StoreManager are entity beans, a class that corresponds to a database table or entity (refer to 6.6.1). An instance of StoreCustomer or StoreManager corresponds to a single database row or record.

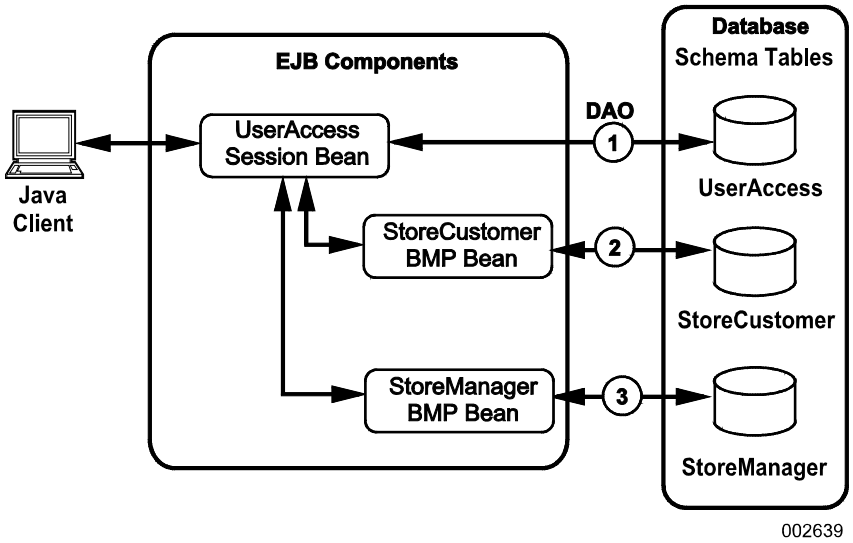

Figure 8–1. BMP Beans in Office Supply Store

8.1.1. Unique Identifiers

To create an Id, add the @Id tag to the field name in the StoreInventory java class of the OfficeSupplyStore project.

8.1.2. Local Access

StoreInventoryBean support a local EJB interface, which means that interface clients are guaranteed to reside in the same JVM. This is assured because they are part of the internal Office Supply Store module implementation and are not exposed to true outside clients.

8.1.3. Session Facade Pattern

In the session facade pattern (refer to Section 7), a session bean provides access to the business methods of BMP beans. In this example, StoreInventory are accessible through StoreInventoryBean.

8.1.4. UserAccess Methods

Clients access StoreInventoryBean through the StoreInventoryLocal interface. StoreInventoryLocal defines the StoreInventoryBean business methods such as verifyThisUser(), addInventory(), and getAllInventory().

StoreInventoryBean communicate with tables in the database using the data access objects StoreInventoryDAO, as illustrated in Figure 8–1.

8.2. Tasks

To create a bean-managed persistence (BMP) entity bean, complete the following task:

1. Create a bean structure named StoreInventory.
2. Modify the code to become a BMP entity bean.
3. Create a DAO class named StoreInventoryDAOImpl.
4. Create a test client named SessionBMPTestClient.
5. Deploy the StoreInventory bean.
6. Run the client and test the bean.

8.2.1. Creating a BMP Entity Bean Structure

To create a BMP entity bean structure named StoreCustomer, perform the following steps:

1. In the Java EE perspective, right-click the **OfficeSupplyStoreEJB** project, point to **New**, and click **Other**.

   The **New** dialog box for selecting a wizard appears.

2. Expand **EJB** and click **SessionBean (EJB 3.x)** and then **Next**.

   The Create EJB 3.x Session Bean wizard appears.

EntityBean Workaround

The EJB creation wizard does not provide an option for creating entity beans in Web Tools 1.1. As a workaround, if EntityBean is not an option in the EJB creation wizard, create a session bean and modify it to be a BMP entity bean, using the following procedure and the modifications in 8.2.2:

1. Create a session bean (refer to 7.2.2).
2. Type **us.com.unisys.bmp** in the **Java package** box (do not try to browse for this because it is not defined yet).
3. Type **StoreInventoryBean** in the **Class name** box and check the Remote and local checkboxes.
4. Click **Next** and then **Finish**.
5. Modify the session bean to be a BMP entity bean by completing all modifications as given in 8.2.2 and its subsections. The modifications include the following:
   a. Changing portions of the code as identified by bold font in the listing
   b. Modifying getter and setter methods
   c. Adding a local business method

8.2.2. Modifying the Code to Create a BMP Bean

The wizard generates code for a session bean. To change the code to BMP entity bean code, complete all modifications in the following subsections, turn on automatic XDoclet generation, and continue with 8.2.3.

The following paragraphs identify several modifications that are needed to complete the bean. A completed version of the code is at the following location:

```
...\examples\8\OfficeSupplyStoreEJB\ejbModule\us\com\unisys\bmp\
StoreCustomerBean.java
```

Modifications to Bean Code

Add the following code to the StoreCustomerBean.java code file:

```
@Stateless
@Remote(StoreInventoryRemote.class)
@TransactionManagement(TransactionManagementType.BEAN)
public class StoreInventoryBean implements StoreInventoryRemote, Serializabl
e {
    @PersistenceUnit(unitName = "manager1")
    EntityManagerFactory emf;
    @Resource
    UserTransaction utx;
    @Resource
    EJBContext etx;
    EntityManager em;
    Connection conn = null;
    protected StoreInventory4 si;
    protected Collection<StoreInventory4> storeList;
    public StoreInventoryBean() {
    }
    private static StoreInventoryDAO dao = new StoreInventoryDAOImpl();
    public String verifyThisUser(String username, String password) {
        return dao.verifyThisUser(username, password);
    }
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void addInventory(String itemnum, String supplierid,

    String description, String qtyonhand, float price) throws Exception {
        System.out.println("before begin inside addInventory");
        utx = etx.getUserTransaction();
        try {
            utx.begin();
            em = emf.createEntityManager();
            if (si == null)
                si = new StoreInventory4(itemnum, supplierid, description,
                      qtyonhand, price);
            em.merge(si);
            utx.commit();
        } catch (Exception e) {
            if (utx != null)
                utx.rollback();
            throw e;
        }
    }
    public Collection<StoreInventory4> getAllInventory() {
        storeList = em.createQuery("from StoreInventory4 b").getResultList();
        return storeList;
    }
```

```
    public StoreInventory4 findById(long id) {
        return ((StoreInventory4) em.find(StoreInventory4.class, id));
    }
}
```

Call StoreInventoryDAOImpl From Bean Class

Call the StoreInventoryDAOImpl from the Bean class, which extends the StoreInventoryDAO interface.

```
private static StoreInventoryDAO dao = new StoreInventoryDAOImpl();
    public String verifyThisUser(String username, String password) {
        return dao.verifyThisUser(username, password);
    }
```

Creating the Entity Bean Class

In the us.com.unisys.bmp package, create a bean class with all the fields that are mapped to the database.

```
String itemnum;
String supplierid;
String description;
String qtyonhand;
float price;
```

To create the getters and setters

1. Select the items in the editor.
2. Right-click, point to **Source**, and click **Generate Getters and Setters**.

Modifications to Getter and Setter Methods

After creating the getter and setter methods, modify the methods to add the following annotations:

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
public long getId() {
return id;
}
```

Tracer Method

Add the following method to the StoreInventoryBean class for tracing purposes:

```
public StoreInventory4 findById(long id) {
return ((StoreInventory4) em.find(StoreInventory4.class, id));
}
```

Local Business Method

Add a local business method in the StoreInventoryBean. The method is local because the method is called from StoreInventoryRemote, which is executing in the same JVM. The method is not visible remotely.

```
public void addInventory(String itemnum, String supplierid,
String description, String qtyonhand, float price) throws Exception;
```

8.2.3. Creating DAO Implementation Classes

The implementation class StoreInventoryDAOImpl and the StoreInventoryDAO interface need to be generated manually, as follows:

1. Right-click the **us.com.unisys.dao** package name, point to **New**, and click **Class**.

   The **New Java Class** dialog box opens.
2. Type **StoreCustomerDAOImpl** in the **Name** box. Be sure the following values appear:
   - Source folder is \OfficeSupplyStoreEJB\ejbModule.
   - Package is us.com.unisys.bmp.
   - Superclass is java.lang.Object.
3. Click **Add** and type **StoreCustomerDAO** (not the fully qualified name), and click **OK**.
4. Be sure the **Inherited abstract methods** check box is selected.
5. Click **Finish**.

Add a code to the method stubs in StoreInventoryDAOImpl, as described in the following paragraphs. A complete version of the code is at the following location:

```
...\examples\8\OfficeSupplyStoreEJB\ejbModule\us\com\unisys\dao\
StoreCustomerDAOImpl.java
```

The following imports are needed:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
```

findByPrimaryKey() Method

The findByPrimaryKey() method does the following:

1. Gets a connection to the database from the ServiceLocator class.
2. Creates an SQL statement that searches for customer-id in the table STORECUSTOMER, where customer-id is the primary key.
3. Returns the primary key.

Add the following code for findByPrimaryKey():

```
public String findByPrimaryKey(String customerid) {
       PreparedStatement ps = null;
       ResultSet rs = null;
   String customerId = null;
   System.out.println("Entering UserAccessDAOImpl.init()");
   conn = ServiceLocator.getDataSource(JNDI_NAME_HYPERSONIC);
   System.out.println(conn);
       if (conn != null) {
           try {
               // conn=ds.getConnection();
   String queryString = "select customerid from STORECUSTOMER where customer
id = ? ";
           ps = conn.prepareStatement(queryString);
           ps.setString(1, customerid);
           rs = ps.executeQuery();
           boolean result = rs.next();
           System.out.println(result);
               if (result) {
                 customerId = rs.getString("customerid");
                 System.out.println("StoreAccessID is " + customerId);
               }
           } catch (SQLException e) {
               e.printStackTrace();
   System.out.println("SQL Exception Inside UserAccessDAOImpl.verifyThisUser
() + e");
           } finally {
               try {
                 rs.close();
                 ps.close();
                 conn.close();
               } catch (Exception e) {
               }
           }
           System.out.println("Leaving UserAccessDAOImpl.verifyThisUser()");
       }
       return customerId;
   }
```

ServiceLocator Class

The ServiceLocator class establishes the connection to the database. To create the class, perform the following steps:

1. Right-click **ejbModule** of the EJB project, point to **New**, and click **Package**.

   The New Java Package window appears.

2. Type **us.com.unisys.service** in the **Name** box, and click **Finish**.

3. Right-click the package that you created, point to **New**, and click **class**.

   The New Java Class window appears.

4. Type **ServiceLocator** in the **Name** box (for the class name).

5. Click **Finish**.

```
package us.com.unisys.service;
import java.sql.Connection;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import javax.annotation.Resource;
import javax.ejb.EJB;
import javax.ejb.Stateless;
public class ServiceLocator {
private static DataSource ds = null;
InitialContext c = null;
static Connection conn = null;
private ServiceLocator() {
}
public static Connection getDataSource(String jndiName) {
try {
Properties props = new Properties();
props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
props.setProperty("java.naming.factory.url.pkgs",
"org.jboss.naming");
props.setProperty("java.naming.provider.url", "127.0.0.1:1099");
InitialContext ctx = new InitialContext(props);
ds = (DataSource) ctx.lookup(jndiName);
conn = ds.getConnection();
} catch (Exception e) {
e.printStackTrace();
}
return conn;
}
```

AddInventory() Method

AddInventory() method is used to load the values to the table that is mapped through the EntityBean class by configuring the persistence.xml file.

```
@Stateless
    @Remote(StoreInventoryRemote.class)
    @TransactionManagement(TransactionManagementType.BEAN)
    public class StoreInventoryBean implements StoreInventoryRemote, Serializ
able {
        @PersistenceUnit(unitName = "manager1")
    EntityManagerFactory emf;
        @ResourceUserTransaction utx;
        @ResourceEJBContext etx;
        EntityManager em;
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void addInventory(String itemnum, String supplierid,
    String description, String qtyonhand, float price) throws Exception {
        System.out.println("before begin inside addInventory");
        utx = etx.getUserTransaction();
        try {
            utx.begin();
            em = emf.createEntityManager();
            if (si == null)
                si = new StoreInventory4(itemnum, supplierid, description,
                       qtyonhand, price);
            em.merge(si);
            utx.commit();
        } catch (Exception e) {
            if (utx != null)
                utx.rollback();
            throw e;
        }
    }
```

Creating the persistence.xml File

To create the persistence.xml file for the Entity BMP bean, perform the following steps:

1. In the ejbModule of EJB project, right-click the **META-INF** folder, point to **New** and click **other**.
2. Click **XML file**.
3. Click **Next**.

   The New XML File window appears.
4. Enter **persistence.xml** in the file name box.
5. Click **Next**.

   The New XML File window appears.
6. Click **Next**.
7. Click **Finish**.

Add the following contents to the persistence.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence">
<persistence-unit name="manager1" >
<jta-data-source>DefaultDS</jta-data-source>
<properties>
<property name="hibernate.hbm2ddl.auto" value="create-drop"/>
<property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
</properties>
</persistence-unit>
</persistence>
```

8.2.4. Modifying StoreInventoryBean

The following paragraphs identify several modifications that are needed for the StoreInventory bean created in the previous section. A completed version of the code is at the following location:

```
...\examples\8\OfficeSupplyStoreEJB\ejbModule\us\com\unisys\session\
UserAccessBean.java
```

The following imports are needed:

```
import java.io.Serializable;
import java.sql.Connection;
import java.util.Collection;
import javax.annotation.Resource;
import javax.ejb.EJBContext;
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import javax.transaction.UserTransaction;
import us.com.unisys.dao.StoreInventoryDAO;
import us.com.unisys.dao.StoreInventoryDAOImpl
```

Deployment Descriptors

Create the hsqlds-ds.xml deployment file in the deploy folder of JBoss server and enter the following contents in the file.

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
<local-tx-datasource>
<jndi-name>DefaultDS</jndi-name>
<use-java-context>false</use-java-context>
<connection-url>jdbc:hsqldb:${jboss.server.data.dir}${/}hypersonic${/}localD
B</connection-url>
<driver-class>org.hsqldb.jdbcDriver</driver-class>
<user-name>sa</user-name>
<password></password>
<min-pool-size>5</min-pool-size>
<max-pool-size>20</max-pool-size>
<idle-timeout-minutes>0</idle-timeout-minutes>
<track-statements/>
<security-domain>HsqlDbRealm</security-domain>
<prepared-statement-cache-size>32</prepared-statement-cache-size>
<metadata>
<type-mapping>Hypersonic SQL</type-mapping>
</metadata>
<depends>jboss:service=Hypersonic,database=localDB</depends>
</local-tx-datasource>
<mbean code="org.jboss.jdbc.HypersonicDatabase" name="jboss:service=Hyperson
ic,database=localDB">
<attribute name="Database">localDB</attribute>
<attribute name="InProcessMode">true</attribute>
```

```
</mbean>
<local-tx-datasource>
   <jndi-name>jdbcunisysos2200rdms</jndi-name>
<connection-url>jdbc:rdms:host=rs02.rsvl.unisys.com;port=1544;schema=OSupply
Store;storagearea=RDMS_AUTO_3;varchar=varchar
</connection-url>
      <driver-class>com.unisys.os2200.rdms.jdbc.RdmsDriver</driver-class>
       <user-name>elango</user-name>
        <password>unisys</password>
        <max-pool-size>5</max-pool-size>
        <min-pool-size>1</min-pool-size>
        <prepared-statement-cache-size>200</prepared-statement-cache-size>
        <idle-timeout-minutes>1000</idle-timeout-minutes>
       <metadata>
            <type-mapping>RDMSOS2200</type-mapping>
        </metadata>
    </local-tx-datasource>
</datasources>
```

Data Retrieval Method

Create a method to retrieve the Inventory data that is visible to a remote client as the remote interface. This method retrieves the entire inventory from the StoreInventory table.

```
public Collection<StoreInventory4> getAllInventory() {
       storeList = em.createQuery("from StoreInventory4 b").getResultList();
       return storeList;
}
```

8.2.5. Creating a Test Client

To create a test client

1. Right-click the **us.com.unisys.client** package name, point to **New**, and click **Class**.

   The **New Java Class** dialog box opens.

2. Type or browse to the following values in the boxes:
   - Source folder is OfficeSupplyStoreClient/appClientModule.
   - Package is us.com.unisys.client (do not try to browse for this because it is not defined yet).
   - Name is SessionBMPTestClient.
   - Superclass is java.lang.Object.
3. Clear the **Inherited abstract methods** check box.
4. Click **Finish**.

The following paragraphs identify several modifications that are needed for the test client. A completed version of the code is at the following location:

...\examples\8\OfficeSupplyStoreClient\appClientModule\us\com\unisys\client\SessionBMPTestClient

```
import java.sql.SQLException;
import java.util.Collection;
import java.util.Iterator;
import java.util.Properties;
import javax.ejb.Stateless;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
import us.com.unisys.bmp.StoreInventory4;
import us.com.unisys.bmp.StoreInventoryRemote;

@Stateless
public class SessionTestClient {
private static StoreInventoryRemote userAccessRemote;
// @EJB static UserAccessRemote userAccessRemote;
static DataSource ds = null;
static Collection list;
static InitialContext ctx = null;
public static void main(String[] args) throws Exception {
try {
Properties props = new Properties();
props.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
props.setProperty("java.naming.factory.url.pkgs","org.jboss.naming");
props.setProperty("java.naming.provider.url", "127.0.0.1:1099");
ctx = new InitialContext(props);
userAccessRemote = (StoreInventoryRemote) ctx
.lookup("OfficeSupplyStore3BMPEAR/StoreInventoryBean/remote");
String customerId = userAccessRemote.findByPrimaryKey("CUST1");
System.out.println("storeAccessId is *****" + customerId);
userAccessRemote.addInventory("ITEM5", "SUPL3", "PHILIPS", "20",
(float) 120.30);
list = userAccessRemote.getAllInventory();
for (Iterator iter = list.iterator(); iter.hasNext();) {
StoreInventory4 element = (StoreInventory4) iter.next();
System.out.println(element.getId());
System.out.println(element.getItemnum());
System.out.println(element.getSupplierid());
System.out.println(element.getDescription());
System.out.println(element.getQtyonhand());
System.out.println(element.getPrice());
}
StoreInventory4 inven = userAccessRemote.findById(1);
```

```
System.out.println("findInventory is **************"
+ inven.getItemnum());
} catch (NamingException e) {
e.printStackTrace();
}
}
}
```

8.2.6. Testing the Bean

To run the client and test the bean

1. Start JBoss AS.
2. Right-click **SessionBMPTestClient**, point to **Run As**, and click **Java Application**.

The output appears in two console panes. Figure 8–2 illustrates the server output, and Figure 8–3 illustrates the client output. If both windows have the same output, close one and open another console pane.

```
15:25:31,490 INFO [STDOUT] Entering UserAccessDAOImpl.init()
15:25:31,490 INFO [STDOUT]
org.jboss.resource.adapter.jdbc.jdk6.WrappedConnectionJDK6@1ec6c08
15:25:31,490 INFO [STDOUT] true
15:25:31,490 INFO [STDOUT] StoreAccessID is CUST1
15:25:31,490 INFO [STDOUT] Leaving UserAccessDAOImpl.verifyThisUser()
15:25:31,506 INFO [STDOUT] before begin inside addInventory
```

Figure 8–2. Server Output from BMP Beans

```
storeAccessId is *****CUST1
1
ITEM5
SUPL3
PHILIPS
20
120.3
findInventory is **************ITEM5
```

Figure 8–3. Client Output from BMP Beans

8.3. Creating Another BMP Entity Bean

Implement the StoreManager bean as a BMP entity bean similar to StoreCustomer, with the same behaviors, using the following tasks. The StoreManagerBean, StoreManagerDAOImpl, modified StoreInventoryBean, and modified SessionBMPTestClient files are provided in the examples folder.

To implement the StoreManager bean

1. Create a BMP Bean named StoreManager under package us.com.unisys.bmp.
2. Create a DAO class named StoreManagerDAOImpl under package us.com.unisys.dao.
3. Add all attributes and properties in StoreManagerBean and getter and setter methods for each attribute.
4. Add a find method named ejbFindByPrimaryKey with signature.

   ```
   public StoreManagerPK ejbFindByPrimaryKey (MangerPK pk) throws FinderExcept
   ion.
   ```

5. Add a business method named getStoreManagerData with signature.

   ```
   public StoreManagerData getManagerData()
   ```

6. Implement methods in StoreManagerDAOImpl class. The lookup string required for JNDI API is

   ```
   java:/DefaultDS
   ```

7. Add a business method in UserAccess bean.

   ```
   public StoreMangerData getStoreManagerData(String managerID)
   ```

8. Test the StoreManager bean by running the test client created for StoreCustomer named SessionBMPTestClient.

Section 9
Creating Container-Managed Persistence Entity Beans

Java Persistence API (JPA) provides POJO (Plain Old Java Object) standard and object relational mapping (OR mapping) for data persistence. Persistence deals with the storing and retrieving of application data and can be programmed with Java Persistence API starting from EJB 3.0. It is an independent API and integrates with J2EE as well as J2SE applications.

9.1. Accessing Office Supply Store

The Office Supply Store application uses two Java Persistence entity beans, as illustrated in Figure 9–1. The StoreInventory bean stores the details of the inventory items, such as availability and prices, for the Office Supply Store. The StoreSupplier bean stores the details of suppliers to the Office Supply Store. Both beans interact with corresponding tables in the database.

An EntityManager instance is associated with a persistence context. A persistence context is a set of entity instances. The entity instances and their life cycles are managed within the persistence context. The EntityManager interface defines the methods that are used to interact with the persistence context. The EntityManager API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

The set of entities that can be managed by a given EntityManager instance is defined by a persistence unit. A persistence unit defines the set of all classes that are related to or grouped by the application, and which must be collocated in their mapping to a SINGLE database.

9.1.1 Unique Identifiers

All inventory items are assigned a unique itemNumber and all suppliers are assigned a unique supplier-id in Office Supply Store.

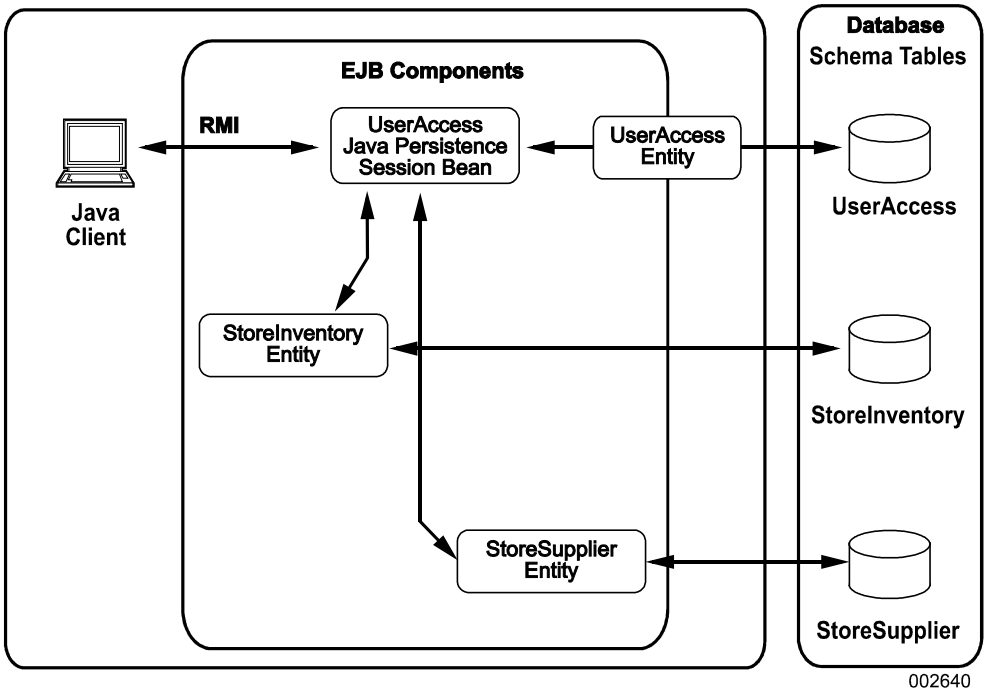

Figure 9–1. Java Persistence in Office Supply Store

9.2 Tasks

Creating a Java persistence entity bean requires the following tasks:

1. Create a bean structure named StoreInventory.
2. Modify the code to become a java persistence entity bean.
3. Implement a method named ejbCreate.
4. Add finder methods named findBySupplierID and findByOutOfStock.
5. Add the business method getStoreInventoryData to get inventory details.
6. Add callback methods to get and set bean context for the bean.
7. Modify the UserAccess bean to store the StoreInventory reference.
8. Add three business methods to UserAccess Bean: getStoreInventory, getOutOfStockInventory, and getStoreInventoryBySupplier.
9. Create a test client named SessionJPTestClient.
10. Run the client and test the bean.

Subsequently, you also need to create the StoreSupplier Java Persistence bean and test it with an updated SessionJPTestClient client (refer to Section 9.3).

9.2.1 Creating a Java Persistence Entity Bean Structure

To create a Java persistence entity bean structure named StoreInventory

1. In the Java EE perspective, right-click the **OfficeSupplyStoreEJB** project, point to **New**, and click **Other**.

   The **New** dialog box for selecting a wizard appears.

2. Expand **EJB** and click **SessionBean (EJB 3.x)**, and then click **Next**.

   The Create EJB 3.x Session Bean wizard appears.

3. Type **us.com.unisys.session** in the **Java package** box (do not try to browse for this because it is not defined yet).
4. Type **StoreInventoryBean** in the **Class name** box.
5. Select the **Remote** and **Local** check boxes.
6. Click **Next**.
7. Click **Finish**.

9.2.2 Creating the Java Persistence Bean Code and the POJO Class

StoreInventory Java Persistence Bean Code

The StoreInventory Java Persistence Bean code is as follows:

```
@Stateless
@Remote(StoreInventoryRemote.class)
@TransactionManagement(TransactionManagementType.CONTAINER)
public class StoreInventoryBean implements Serializable,StoreInventoryRemote
 {
    @PersistenceContext(unitName = "manager1")
    EntityManager em;
    // @Resource UserTransaction ut;
    protected StoreInventory2 si;
    protected Collection<StoreInventory2> storeList;
    public StoreInventoryBean() {
    }
    @Resource
    SessionContext ctx;
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void addInventory(String itemnum, String supplierid,

    String description, String qtyonhand, float price) throws Exception {
        // Initialize the form
        System.out.println("before begin inside addBook");
        try {
            if (si == null)
                si = new StoreInventory2(itemnum, supplierid, description,
```

```
                    qtyonhand, price);
            em.merge(si);
        } catch (Exception e) {
            ctx.setRollbackOnly();
            throw e;
        }
    }
    public Collection<StoreInventory2> getAllInventory() {
        storeList = em.createQuery("from StoreInventory2 b").getResultList();
        return storeList;
    }
}
```

Creating the POJO Class and adding Getter, Setter Methods

The following columns are defined in the StoreInventory table:

- ItemNumber, SupplierID, and Description are strings.
- QuantityOnHand is an integer.
- Price is a floating point number.

Right-click the **us.unisys.com.jp** package and create the StoreInventory class.

Add the following getter and setter signatures in the StoreInventory class:

```
package us.com.unisys.jp;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import java.util.Collection;
import javax.persistence.*;
import java.io.Serializable;
import javax.persistence.Version;
@Entity
@Table(name = "STOREINVENTORY2")
 public class StoreInventory2 implements Serializable {
    private static final long serialVersionUID = 1L;
    String itemnum;
    String supplierid;
    String description;
    String qtyonhand;
    float price;
    long id;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public long getId() {
        return id;
```

```
    }
    public void setId(long id) {
        this.id = id;
    }
    public StoreInventory2() {
        super();
    }
    public float getPrice() {
        return price;
    }
    public StoreInventory2(String itemnum, String supplierid,
            String description, String qtyonhand, float price) {
        this.itemnum = itemnum;
        this.supplierid = supplierid;
        this.description = description;
        this.qtyonhand = qtyonhand;
        this.price = price;
    }
    public String getItemnum() {
        return itemnum;
    }
    public void setItemnum(String itemnum) {
        this.itemnum = itemnum;
    }
    public String getSupplierid() {
        return supplierid;
    }
    public void setSupplierid(String supplierid) {
        this.supplierid = supplierid;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public String getQtyonhand() {
        return qtyonhand;
    }
    public void setQtyonhand(String qtyonhand) {
        this.qtyonhand = qtyonhand;
    }
    public void setPrice(float price) {
        this.price = price;
    }
}
```

In front of `getItemnum()` getter, add an annotation describing the access to the database, persistence, and the remote access.

For ItemNumber, the annotation is as follows:

```
/**
 * @ID
   @GeneratedValue(strategy = GenerationType.AUTO)
       public String getItemnum() {
       return itemnum;
           }
```

`@ID` is used to specify the generator for automatic key generation when new objects are created.

9.2.3 Adding Finder Methods

The home interface for an entity bean defines one or more finder methods to find an entity object or a collection of entity objects. The name of each finder method starts with the prefix "findBy," such as findByPrice, findBySupplierID or findByID.

The findById method defined in the StoreInventoryBean class is as follows:

```
public StoreInventory2 findById(long id) {
         return ((StoreInventory2)em.find(StoreInventory2.class, id));
         }
```

9.2.4 Adding Data Methods

To get inventory details, add the following getAllInventory business method to the StoreInventoryBean class:

```
public Collection<StoreInventory2> getAllInventory() {
       storeList = em.createQuery("from StoreInventory2 b").getResultList();
       return storeList;
   }
```

9.2.5 Callback Methods in EJB 3.0

It is not necessary to include callback methods in EJB 3.0 as in EJB 2.1. However, you can add listener methods using annotation to do the event.

You can add the following callback methods as an annotation:

- @PrePersist

  Executed before the entity manager persist operation is actually executed or cascaded. This callback method is synchronous with the persist operation.

- @PostPersist

  Executed after the entry manager persist operation is actually executed or cascaded. This callback method is invoked after the database INSERT is executed.

- @PreRemove

  Executed before the entity manager REMOVE operation is actually executed or cascaded. This callback method is synchronous with the REMOVE operation.

- @PostRemove

  Executed after the entity manager REMOVE operation is actually executed or cascaded. This callback method is synchronous with the REMOVE operation.

- @PreUpdate

  Executed before the entity manager database UPDATE operation is executed.

- @PostUpdate

  Executed after the entity manager database UPDATE operation is executed.

- @PostLoad

  Executed after an entity has been loaded into the current persistence context or an entity has been refreshed.

9.2.6 Creating the JPA persistence.xml File

To create the JPA persistence.xml file, perform the following steps:

1. In the OfficeSupplyStoreEJB project, right-click the **META-INF** folder, point to **New**, and click **other**.
2. Click **XML file**.
3. Click **Next**.

   The New XML File window appears.
4. Type **persistence.xml** in the **File name** box.
5. Click **Next**.

   The New XML File window appears.
6. Click **Next**.
7. Click **Finish**.

Add the following contents to the persistence.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence">
<persistence-unit name="manager1" >
    <jta-data-source>DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      <property name="hibernate.dialect"
```

```
        value="org.hibernate.dialect.HSQLDialect"/>
    </properties>
  </persistence-unit>
</persistence>
```

Create the hsqlds-ds.xml file as described in Section 8.2. Then, copy the hsqlds-ds.xml file in the **deploy** folder along with the .ear file.

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
<local-tx-datasource>
<jndi-name>DefaultDS</jndi-name>
<use-java-context>false</use-java-context>
<connection-url>jdbc:hsqldb:${jboss.server.data.dir}${/}hypersonic${/}localD
B</connection-url>
<driver-class>org.hsqldb.jdbcDriver</driver-class>
<user-name>sa</user-name>
<password></password>
<min-pool-size>5</min-pool-size>
<max-pool-size>20</max-pool-size>
<idle-timeout-minutes>0</idle-timeout-minutes>
<track-statements/>
<security-domain>HsqlDbRealm</security-domain>
<prepared-statement-cache-size>32</prepared-statement-cache-size>
<metadata>
<type-mapping>Hypersonic SQL</type-mapping>
</metadata>
<depends>jboss:service=Hypersonic,database=localDB</depends>
</local-tx-datasource>
<mbean code="org.jboss.jdbc.HypersonicDatabase" name="jboss:service=Hyperson
ic,database=localDB">
<attribute name="Database">localDB</attribute>
<attribute name="InProcessMode">true</attribute>
</mbean>
<local-tx-datasource>
   <jndi-name>jdbcunisysos2200rdms</jndi-name>
<connection-url>jdbc:rdms:host=rs02.rsvl.unisys.com;port=1544;schema=OSupply
Store;storagearea=RDMS_AUTO_3;varchar=varchar
</connection-url>
        <driver-class>com.unisys.os2200.rdms.jdbc.RdmsDriver</driver-class>
       <user-name>elango</user-name>
        <password>unisys</password>
        <max-pool-size>5</max-pool-size>
        <min-pool-size>1</min-pool-size>
        <prepared-statement-cache-size>200</prepared-statement-cache-size>
        <idle-timeout-minutes>1000</idle-timeout-minutes>
        <metadata>
            <type-mapping>RDMSOS2200</type-mapping>
        </metadata>
```

```
    </local-tx-datasource>
</datasources>
```

9.2.7 Adding Inventory Access Methods

Add other business methods to the StoreInventoryRemote to invoke on the StoreInventory bean. A user logs in to StoreInventory with username and password. Once validated, the user can retrieve details from StoreInventory by invoking methods on the StoreInventory bean to examine the inventory.

Inventory Data Method

Create a business method named findById to return the inventory. Add the method in StoreInventory Bean class as given:

```
public StoreInventory2 findById(long id)
{
      return ((StoreInventory2)em.find(StoreInventory2.class, id));
}
```

9.2.8 Creating a Test Client

Create a test client to access the beans and methods developed so far to invoke the following methods on Store inventory bean:

- addInventory
- getAllInventory
- findById

A completed version of the code is at the following location:

```
...\examples\9a\OfficeSupplyStoreClient\appClientModule\us\com\unisys\client\
SessionCMPTestClient.java
```

To create a test client

1. From the OfficeSupplyStoreClient project, right-click the **us.com.unisys.client** package name, point to **New**, and click **Class**.

   The **New Java Class** dialog box opens.
2. Type or browse to the following values in the boxes:
   - Source folder is OfficeSupplyStoreClient/appClientModule.
   - Package is us.com.unisys.client (do not try to browse for this because it is not defined yet).
   - Name is SessionJPTestClient.
   - Superclass is java.lang.Object.
3. Clear the **Inherited abstract methods** check box.
4. Click **Finish**.

Test Method

In the testBean method, invoke addInventory, getAllInventory, and findById.

The TestClient java code is as follows:

```
import java.util.Collection;
import java.util.Iterator;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.transaction.SystemException;
import us.com.unisys.cmp.StoreInventory2;
import us.com.unisys.cmp.StoreInventoryLocal;
import us.com.unisys.cmp.StoreInventoryRemote;

public class StoreInventory2Client {
    private static StoreInventoryRemote bci = null;
      String s1,s2,s3;
      static Collection list;
    public static void main(String[] args) throws NamingException, IllegalSta
teException, SecurityException, SystemException ,Exception{
       Properties props = new Properties();
        props.setProperty("java.naming.factory.initial","org.jnp.interfaces.
NamingContextFactory");
        props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming"
);
        props.setProperty("java.naming.provider.url", "127.0.0.1:1099");
        InitialContext ctx = new InitialContext(props);
    bci = ( StoreInventoryRemote) ctx.lookup("OfficeSupplyStore2/StoreInvento
ryBean/remote");
       bci.addInventory("ITEM5","SUPL3","SONY","20",(float)120.30);
       StoreInventory2 inven = bci.findById(3);
       System.out.println("findInventory is **************"+inven.getId());
              list=bci.getAllInventory();
         for (Iterator iter = list.iterator(); iter.hasNext();)
         {
         StoreInventory2 element = (StoreInventory2)iter.next();
         System.out.println(element.getId());
         System.out.println(element.getItemnum());
         System.out.println(element.getSupplierid());
         System.out.println(element.getDescription());
         System.out.println(element.getQtyonhand());
         System.out.println(element.getPrice());
         }
       }
     }
```

9.2.9 Testing the Bean

To run the client and test the bean

1. Start JBoss AS.
2. Right-click the **SessionJPTestClient** node, point to **Run As**, and click **Java Application**.

The output appears in two windows, one for the server and the other for the client.

The console pane should look like Figure 9–2 and Figure 9–3.

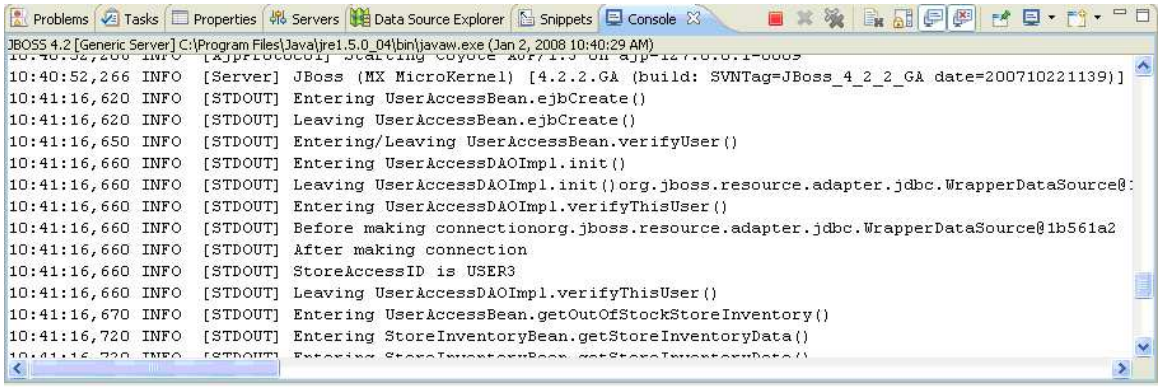

Figure 9–2. Server Output from Java Persistence Beans

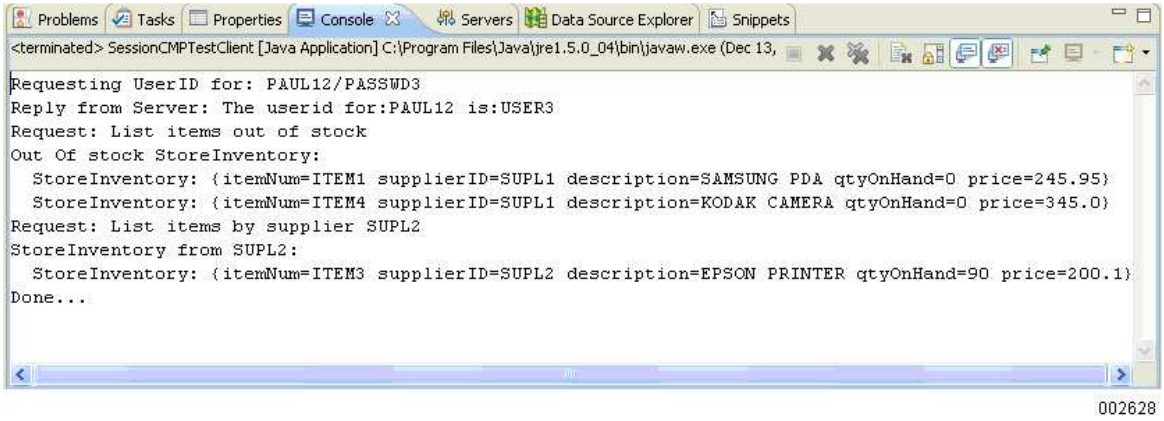

Figure 9–3. Client Output from Java Persistence Beans

Section 10
Creating Web Client Servlets

Web clients run in either the client tier or the presentation tier to access or otherwise communicate with the business tier.

10.1. Accessing Office Supply Store

The Office Supply Store application uses a Web client servlet, as illustrated in Figure 10–1. The servlet sends user requests to the UserAccess bean and returns the results.

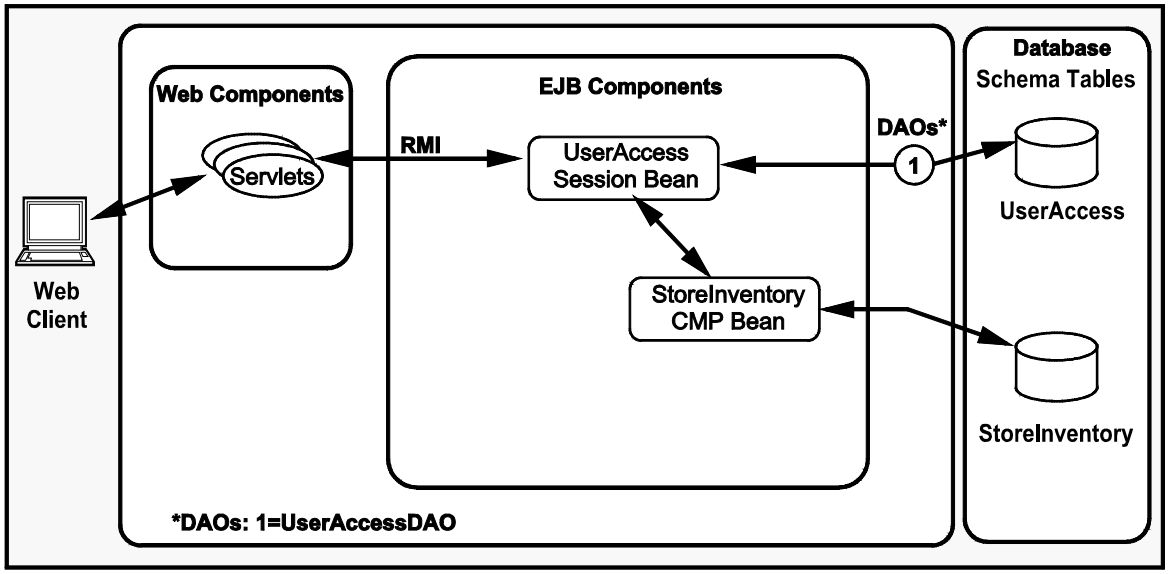

Figure 10–1. Web Client Servlets in Office Supply Store

10.1.1. Types of Web Clients

Web clients are either servlets or JavaServer Pages (JSP), which are defined as follows:

- Servlets are Java programming language classes that dynamically process requests and construct responses.
- JSPs are text-based documents that execute like servlets but allow a natural language approach to creating static content.

Refer to Section 11 for creating JSPs.

10.1.2. Web Client Pattern

A typical pattern for Web clients uses models, views, and controllers, where

- The model is the data.
- The JSPs act as the view.
- The servlet acts as the controller.

Figure 10–1 identifies these relationships in the Office Supply Store.

10.2. Tasks

Complete the following tasks to create a servlet:

1. Create a Web project named OfficeSupplyStoreWeb.
2. Create a servlet named UserAccessController.
3. Add a remote business method to UserAccess Bean named getStoreInventory().
4. Modify the servlet for the project.
5. Implement a method named processRequest and other helper methods.
6. Test the UserAccessController servlet.

10.2.1. Creating a Web Project

You can create a Web project when the Java EE project is created or when needed. To create a Web project

1. Right-click **OfficeSupplyStore** on the **Project Explorer** tab, point to **New**, and click **Dynamic Web Project**.

   The New Dynamic Web Project wizard appears.
2. Type **OfficeSupplyStoreWeb** in the **Project Name** box.
3. Be sure the **Use default** check box under **Project contents** is selected.
4. Select **JBoss 4.2** from the **Target runtime** list.

5. Select the **Add project to an EAR** check box.
6. Be sure **OfficeSupplyStoreWebEAR** is selected in the **EAR Project Name** list, and click **Next**.
7. Be sure **OfficeSupplyStoreWeb** is in the **Context Root** box and **WebContent** is in the **Content Directory** box.
8. Type **src** in the **Java Source Directory** box, and click **Finish**.

10.2.2. Creating a Servlet

To create a servlet named UserAccessController

1. Right-click the OfficeSupplyStoreWeb project, point to **New**, and click **Servlet**.

   The Create Servlet wizard appears with the following values in the boxes:

   - Project is OfficeSupplyStoreWeb.
   - Folder is \OfficeSupplyStoreWeb\src.
   - Superclass is javax.servlet.http.HttpServlet.
2. Type **us.com.unisys.servlet** in the **Java package** box (do not try to browse for this because it is not defined yet).
3. Type **UserAccessController** in the **Class name** box
4. Click **Next**.
5. Type **signon** in the **Name** box.

   The value under **URL Mappings** changes to **/signon**.
6. Click **Finish**.

10.2.3. Adding Remote Business Method

Before implementing the servlet, add another remote business method named getStoreInventory to UserAccessBean.java. This method returns all the items in Office Supply Store by invoking the finder method named findAll in StoreInventoryLocalHome. A completed version of this code is at the following location:

```
...\examples\10\OfficeSupplyStoreEJB\ejbModule\us\com\unisys\session\
UserAccessBean.java
```

Add the following method to UserAccessBean:

```
/**
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void addInventory(String itemnum, String supplierid,
    String description, String qtyonhand, float price) throws Exception {
            // Initialize the form
        System.out.println("before begin inside addBook");
        try {
```

```
        if (si == null)
            si = new StoreInventory(itemnum, supplierid, description,
                  qtyonhand, price);
            em.merge(si);
        } catch (Exception e) {
            ctx.setRollbackOnly();
            throw e;
        }
    }
    public Collection<StoreInventory> getStoreInventory() {
    storeList = em.createQuery("from StoreInventory b").getResultList();
        return storeList;
    }
    public StoreInventory findById(long id) {
          return ((StoreInventory)em.find(StoreInventory.class, id));
      }
```

10.2.4. Modifying the Servlet for the Project

The following paragraphs describe how to modify the servlet for the project. A completed version of this code is at the following location:

`...\examples\10\OfficeSupplyStoreWeb\JavaSource\us\com\unisys\servlet\UserAccessController.java`

Project Definitions

To modify the OfficeSupplyStoreWeb project properties to include definitions from the EJB project

1. In the Java EE perspective, right-click the **OfficeSupplyStoreWeb** project, and click **Properties**.

   The **Properties** dialog box opens.

2. Select **Java Build Path** on the left tree pane.

3. On the **Projects** tab, click **Add**, and select **OfficeSupplyStoreEJB** in the **Required projects on the build path** list.

   OfficeSupplyStoreWeb/build/classes appear in the **Default output folder** box on the **Source** tab.

4. Click **OK**.

Imported Files

Add the following to UserAccessController.java. Not all are needed initially, but they are needed eventually.

```
import java.io.IOException;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Properties;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
import javax.servlet.http.HttpSession;

import us.com.unisys.bmp.StoreInventory;
import us.com.unisys.bmp.UserAccessBeanRemote;
```

init Method

The init method is responsible for initializing servlets. It is invoked once when the servlet is first created. In the init method, initialize the references for UserAccess bean. All client interfaces available are exposed in UserAccess.

```
 private UserAccessBeanRemote userAccessHome=null;
 public void init(ServletConfig config) throws ServletException {
      super.init(config);
      initUserAccess();
   }
   private void initUserAccess() {
      System.out.println("Entering UserAccessController.initUserAccess()");
      try {
         userAccessHome =
    (UserAccessHome)getContext().lookup("OfficeSupplyStore4EAR/UserAccessBean
/remote ");
      } catch ( Exception e) {
         System.out.println("Error in UserAccessController.initUserAccess()"
 +                                          e);
      }
      System.out.println("Leaving UserAccessController.initUserAccess()");
```

```
}

private InitialContext getContext() throws NamingException {
   Properties props = new Properties();
   props.setProperty("java.naming.factory.initial",
             "org.jnp.interfaces.NamingContextFactory");
 props.setProperty("java.naming.factory.url.pkgs","org.jboss.naming");
   props.setProperty("java.naming.provider.url", "127.0.0.1:1099");
   InitialContext ctx = new InitialContext(props);
   Return ctx;
}
```

10.2.5. Implementing Helper Methods

To implement the doGet and doPost methods, create a helper method named processRequest to provide the functionality for both of them. Requests are delegated to processRequest, where all business logic processing takes place. Once processing is complete, processRequest dispatches the request to the appropriate view (JSP) for display.

The processRequest method acts as a controller for all requests and uses other helper methods to do its work, as identified in the following paragraphs.

processRequest Method

The processRequest helper method performs the following steps:

1. Checks for the parameter *useraction* in the request object.
2. If *useraction* is empty, builds a URL and generates a log-in window.
3. If *useraction* has some value, examines it for processing.
4. If the request is to display items, builds a URL and generates a window that displays all items.
5. If errors occur, generates a window that displays the errors.

Implement the following code to create the processRequest helper method:

```
private static String LOGIN_SCREEN = "/login";
private static String LOGIN_ERROR_SCREEN = "loginError";
private static String ITEMS_SCREEN = "/showStoreItems";

protected void processRequest(HttpServletRequest request,
      HttpServletResponse response)
           throws ServletException, IOException {
   System.out.println("Entering UserAccessController.processRequest()");
   String buildURL = null ;
   HttpSession session = request.getSession(true);
   String userAction = request.getParameter("useraction");
   if (userAction == null){
      buildURL = LOGIN_SCREEN;
```

```
      } else {
         if (userAction.equals("dovalidation")) {
            System.out.println("Verifying User");
            String username = request.getParameter("username");
            String password = request.getParameter("password");
            if ((username == null) || !(checkUser(username,password,session)
))              {
              System.out.println("Error: Invalid attempt for username:" +
                                        username);
               buildURL = LOGIN_ERROR_SCREEN;
            } else {
               buildURL = ITEMS_SCREEN;
            }
         }
      }
      if ( buildURL == ITEMS_SCREEN) {
         try {
            String userID = (String)session.getAttribute("userID");
            theStore = userAccessHome.getStoreInventory();
            displayAllItems(response, theStore);
            theStore.remove();
         } catch (Exception e) {
            System.out.println("Error in UserAccessController.processRequest
()"                                          + e);
         }
      } else {
         if ( buildURL == LOGIN_SCREEN ) {
            displayLoginScreen(response);
         } else if (buildURL == LOGIN_ERROR_SCREEN) {
            displayLoginErrorScreen(response);
         }
      }
      System.out.println("Leaving UserAccessController.processRequest()");
   }
```

doGet, doPost Methods

The doGet and doPost methods simply call the processRequest helper method.

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response)
                           throws ServletException, IOException {
      processRequest(request, response);
   }
```

displayLoginScreen Method

If *useraction* is empty, which it is initially, processRequest builds the URL for the log-in screen and generates the log-in screen by invoking the method displayLoginScreen.

```
private void displayLoginScreen(HttpServletResponse response) throws
                                                    IOException {
         System.out.println("Entering
                          UserAccessController.displayLoginScreen()");
         response.setContentType("text/html");
         PrintWriter out = response.getWriter();
         out.println("<html><title>The Office Supply Store Login</title>");
         out.println("<body><h2>Welcome to the Office Supply Store</h2></bod
y>");
         out.println("<form method=\"get\">");
         displayLoginDataFields(out);
         out.println("</form></html>");
         if ( out != null ) {
            out.close();
         }
        System.out.println("Leaving
                          UserAccessController.displayLoginScreen()");
     }
```

displayLoginDataFields Method

The method displayLoginScreen calls displayLoginDataFields to generate the data fields for the log-in screen, as follows:

```
private void displayLoginDataFields(PrintWriter out) {
         System.out.println("Entering
                          UserAccessController.displayLoginDataFields()");
         out.println("<h3>Please enter your username and password:");
         out.println("<table><tr><td>Username: <td><input name=\"username\"
                           type=\"test\"/>");
         out.println("<tr><td>Password: <td><input name=\password\"
                           type=\"password\"/>");
         out.println("</table>");
         out.println("<input type=\"submit\" value=\"login\"
                           name=\"loginButton\"/>");
         out.println("<input type=\"reset\" name=\"resetButton\"
                           value=\"reset\"/>");
         out.println("<input type=\"hidden\" name=\"useraction\"
                           value=\"dovalidation\"/>");
         System.out.println("Leaving
                          UserAccessController.displayLoginDataFields()");
      }
```

checkUser, verifyUser Methods

Once the log-in screen is submitted, processRequest checks to see whether the username and password are valid by invoking the checkUser method, which invokes the verifyUser method in the UserAccessBean, as follows:

```
private boolean checkUser(String username, String passwd, HttpSession
                                        session){
        System.out.println("Entering UserAccessController.checkUser()");
        String userID = null ;
        try {
           userID = userAccessHome.verifyThisUser(username, passwd);
           session.setAttribute("userID", userID);
        } catch (Exception e) {
           System.out.println("Error in UserAccessController.checkUser()" +
                                             e);
        }
        System.out.println("Leaving UserAccessController.checkUser()");
        return (userID != null);
     }
```

displayAllItems Method

The processRequest method builds URLs for displaying items and errors, using the displayAllItems method, as follows:

```
private void displayAllItems(HttpServletResponse response, ArrayList
                                          itemsList)
       throws IOException {
         System.out.println("Entering UserAccessController.displayAllItems()
");
         response.setContentType("text/html");
         PrintWriter out = response.getWriter();
         out.println("<html><title>List of Office Supply Store Inventory:</t
itle>");
         out.println("<form method=\"get\">");
         out.println("<body><h2>The Office Supply Store Inventory (servlet)
                                                  </h2></body>");
         if ( itemsList.isEmpty()){
            out.println("<p><h3><b>No Items are available!</b></h3>");
         } else {
            out.println("<p><table border=\"1\">");
            out.println("<th><b>ItemID</b>");
            out.println("<th><b>Description</b>");
            out.println("<th><b>Quantity</b>");
            out.println("<th><b>Price</b></tr>");
            Iterator items = itemsList.iterator();
            StoreInventory item = null ;
            while ( items.hasNext()) {
               item = (StoreInventoryData)items.next();
               out.println("<tr><td>");
               out.println(item.getItemNumber());
               out.println("</td><td>");
               out.println(item.getDescription());
               out.println("</td><td>");
               out.println(item.getQuantityOnHand());
               out.println("</td><td>");
               out.println(item.getPrice());
               out.println("</td></tr>");
            }
           out.println("</table>");
         }
         out.println("</html>");
         if ( out != null ) {
            out.close();
         }
         System.out.println("Leaving UserAccessController.displayAllItems()"
);
      }
```

displayLoginErrorScreen Method

The processRequest method displays errors using the displayLoginErrorScreen method, as follows:

```
private void displayLoginErrorScreen(HttpServletResponse response) throws
                        IOException {
        System.out.println("Entering

   UserAccessController.displayLoginErrorScreen()");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><title>OfficeSupplyStore Login Error</title>");
        out.println("<h3>Please try again, your login has failed!");
        out.println("<form method=\"get\">");
        displayLoginDataFields(out);
        out.println("</form></html>");
        if ( out != null ) {
           out.close();
        }
        System.out.println("Leaving

   UserAccessController.displayLoginErrorScreen()");
     }
```

10.2.6. Testing the Servlet

To test the servlet

1. From the browser, access the servlet using the following URL:

   http://localhost:8080/OfficeSupplyStoreWeb/signon

   where **signon** is the URL mapping that was assigned while creating the servlet using the servlet creation wizard, and **OfficeSupplyStoreWeb** is the Web module where this servlet UserAccessController resides.

   The log-in window appears.

   Notes:

   - *Log-in entries are case sensitive.*
   - *If the log-in window does not appear, try cleaning the project and restarting JBoss AS.*

2. Type **sa** in the **Username** box.
3. Leave the **Password** box blank.
4. Click **login**.

The Office Supply Store Inventory window appears, as illustrated in Figure 10–2.

Figure 10–2. Office Supply Store Inventory from the Servlet

Section 11
Creating Web Client JavaServer Pages

Web clients run in either the client tier or the presentation tier to access or otherwise communicate with the business tier. Refer to Section 10 for more information about types of Web clients, Web client patterns, and creating servlets.

11.1. Accessing Office Supply Store

The Office Supply Store application uses a Web client JSP, as illustrated in Figure 11–1. The JSP communicates through the servlet to send user requests to the UserAccess bean and display the results.

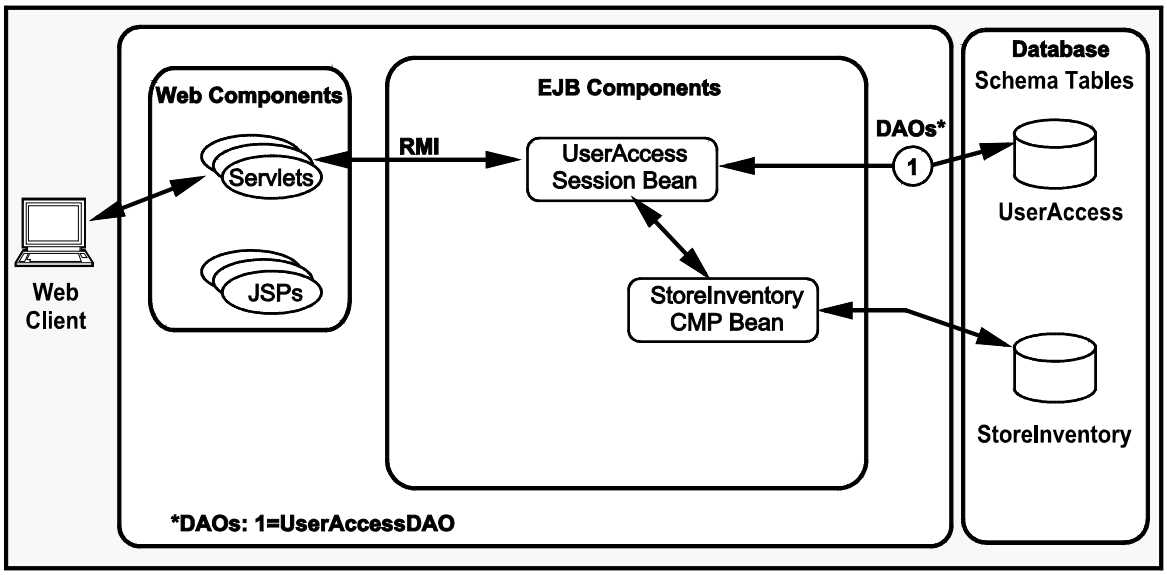

Figure 11–1. Web Client Servlets and JSPs in Office Supply Store

11.2. Tasks

Complete the following tasks to create a JSP:

1. Create a JSP named showStoreItems.
2. Modify the method processRequest in the UserAccessController servlet.
3. Add HTML and JSP tags to display a list of all inventory items in the Office Supply Store.
4. Deploy the module OfficeSupplyStoreWeb.
5. Test the showStoreItems JSP.

11.2.1. Creating JavaServer Pages

To create a JSP named showStoreItems

1. From the **Project Explorer**, right-click **OfficeSupplyStoreWeb**, point to **New**, and click **JSP**.

   The **New JavaServer Page** dialog box appears.
2. Expand **OfficeSupplyStoreWeb** and select **WebContent**.

   OfficeSupplyStoreWeb/WebContent appears in the box as the parent folder.
3. Type **showStoreItems.jsp** in the **File name** box.
4. Click **Finish**.

Writing a JSP is beyond the scope of this example. For demonstration purposes, copy a JSP from the following completed examples to a similarly named workspace location:

```
...\examples\11\OfficeSupplyStoreWeb\WebContent\showStoreItems.jsp
```

The completed JSP contains the following code:

```
<%@ page language="java" import="java.util.*"%>
<%@ page language="java" import="us.com.unisys.bmp.*"%>
<%-- <jsp:useBean id="inventoryData" scope="page" class="us.com.unisys.bmp.S
toreInventory">
    <jsp:setProperty name="inventoryData" property="*"/>
</jsp:useBean>
 --%><!DOCTYPE HTML PUBLIC "-w3cdtd html 4.0 transitionalen">
<html>
<head>
<title>Office Supply Store</title>
</head>
<body bgcolor="#FFFFFF">
<form method="get">
<b><h3>The Office Supply Store Inventory (jsp)</h3></b></br>
<table border="1">
```

```
<tr><td><B>ItemID</B></td>
<td><B>Description</B></td>
<td><B>Quantity</B></td>
<td><B>Price</B></td></tr>
<%
ArrayList arr = (ArrayList) session.getAttribute("InventoryList");
for ( int i = 0; i < arr.size(); i++ ) {
    StoreInventory inventoryList = (StoreInventory)arr.get(i);
    %>
<TR>
<TD><%=inventoryList.getItemnum()%></TD>
<TD><%=inventoryList.getDescription()%></TD>
<TD><%=inventoryList.getQtyonhand()%></TD>
<TD><%=inventoryList.getPrice()%></TD>
</TR>
 <%
}
%>
</table>
</form>
</body>
</html>
```

11.2.2. Modifying the Servlet for the JSP

To display Office Supply Store inventory items from the JSP, modifications to the processRequest method in the servlet code in UserAccessController.java are required. A completed version of the code is at the following location:

```
...\examples\11\OfficeSupplyStoreWeb\JavaSource\us\com\unisys\servlet\
UserAccessController.java
```

The complete servlet contains the following code:

```
package us.com.unisys.servlet;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

```
import javax.rmi.PortableRemoteObject;
import javax.servlet.http.HttpSession;
import us.com.unisys.bmp.StoreInventory;
import us.com.unisys.bmp.UserAccessBeanRemote;
/**
 * Servlet implementation class UserAccessController
 */
public class UserAccessController extends HttpServlet {
    private static final long serialVersionUID = 1L;
    /**
     * @see HttpServlet#HttpServlet()
     */
    public UserAccessController() {
        super();
        // TODO Auto-generated constructor stub
    }
    private UserAccessBeanRemote userAccessHome = null;
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        initUserAccess();
    }
    private void initUserAccess() {
        System.out.println("Entering UserAccessController.initUserAccess()");
        try {
            userAccessHome = (UserAccessBeanRemote) getContext().lookup(
                  "OfficeSupplyStore4EAR/UserAccessBean/remote");
        } catch (Exception e) {

    System.out.println("Error in UserAccessController.initUserAccess()"
                + e);
        }
        System.out.println("Leaving UserAccessController.initUserAccess()");
    }
    private InitialContext getContext() throws NamingException {
        Properties props = new Properties();
        props.setProperty("java.naming.factory.initial",
               "org.jnp.interfaces.NamingContextFactory");

    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");
        props.setProperty("java.naming.provider.url", "127.0.0.1:1099");
        InitialContext ctx = new InitialContext(props);
        return ctx;
    }
    private static String LOGIN_SCREEN = "login";
    private static String LOGIN_ERROR_SCREEN = "loginError";
    private static String ITEMS_SCREEN = "showStoreItems";
    ArrayList theStore;
    protected void processRequest(HttpServletRequest request,
```

```
    HttpServletResponse response) throws ServletException, IOException {
        System.out.println("Entering UserAccessController.processRequest()");
        String buildURL = null;
        HttpSession session = request.getSession(true);
        String userAction = request.getParameter("useraction");
        System.out.println(userAction);
        if (userAction == null) {
            buildURL = LOGIN_SCREEN;
        } else {
            if (userAction.equals("dovalidation")) {
                System.out.println("Verifying User");
                String username = request.getParameter("username");
                String password = request.getParameter("password");
                if ((username == null)
                      || !(checkUser(username, password, session))) {
                  System.out.println("Error: Invalid attempt for username:"
                         + username);
                  buildURL = LOGIN_ERROR_SCREEN;
                } else {
                  buildURL = ITEMS_SCREEN;
                }
            }
        }
        if (buildURL == ITEMS_SCREEN) {
            try {
                String userID = (String) session.getAttribute("userID");
                theStore = userAccessHome.getStoreInventory();
                System.out.println(theStore.size());
                //displayAllItems(response, theStore);
                request.setAttribute("InventoryList", theStore);
                session.setAttribute("InventoryList", theStore);
    RequestDispatcher rd =getServletContext().getRequestDispatcher("/showStor
eItems.jsp");
                rd.forward(request, response);
                theStore.remove(userID);
            } catch (Exception e) {
                System.out

    .println("Error in UserAccessController.processRequest()"
                           + e);
            }
        } else {
            if (buildURL == LOGIN_SCREEN) {
                displayLoginScreen(response);
            } else if (buildURL == LOGIN_ERROR_SCREEN) {
                displayLoginErrorScreen(response);
            }
        }
```

```
        System.out.println("Leaving UserAccessController.processRequest()");
    }
    protected void doGet(HttpServletRequest request,

    HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }
    private void displayLoginScreen(HttpServletResponse response)
           throws IOException {
        System.out

    .println("Entering UserAccessController.displayLoginScreen()");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("Hello World");
        out.println("<html><title>The Office Supply Store Login</title>");

    out.println("<body><h2>Welcome to the Office Supply Store<h2></body>");
        out.println("<form method=\"get\">");
        /*
         * out.println("<h3>Please enter your username and password:");
         * out.println
     * ("<table><tr><td>Username: <td><input name=\"username\"type=\"test\"/>
"
         * ); out.println(

     * "<tr><td>Password: <td><input name=\"password\"type=\"password\"/>");
         * out.println("</table>"); out.println(
         * "<input type=\"submit\" value=\"login\"name=\"loginButton\"/>");
         * out.println
         * ("<input type=\"reset\" name=\"resetButton\"value=\"reset\"/>");
         * out.println

     * ("<input type=\"hidden\" name=\"useraction\"value=\"dovalidation\"/>"
         * );
         */
        displayLoginDataFields(out);
        out.println("</form></html>");
        if (out != null) {
            out.close();
        }

    System.out.println("Leaving UserAccessController.displayLoginScreen()");
    }
    private void displayLoginDataFields(PrintWriter out) {

    System.out.println("Entering UserAccessController.displayLoginDataFields(
)");
        out.println("<h3>Please enter your username and password:");
```

```
    out.println("<table><tr><td>Username: <td><input name=\"username\"type=\"
test\"/>");

    out.println("<tr><td>Password: <td><input name=\"password\"type=\"passwor
d\"/>");
        out.println("</table>");
    out.println("<input type=\"submit\" value=\"login\"name=\"loginButton\"/>
");
    out.println("<input type=\"reset\" name=\"resetButton\"value=\"reset\"/>"
);
    out.println("<input type=\"hidden\" name=\"useraction\"value=\"dovalidati
on\"/>");
    System.out.println("Leaving    UserAccessController.displayLoginDataField
s()");
        if (out != null) {
            out.close();
        }
    }
    private boolean checkUser(String username, String passwd,
            HttpSession session) {
        System.out.println("Entering UserAccessController.checkUser()");
        String userID = null;
        try {
            userID = userAccessHome.verifyThisUser(username, passwd);
            System.out.println("userID is **********" + userID);
            session.setAttribute("userID", userID);
        } catch (Exception e) {

    System.out.println("Error in UserAccessController.checkUser()" + e);
        }
        System.out.println("Leaving UserAccessController.checkUser()");
        return (userID != null);
    }
    private void displayAllItems(HttpServletResponse response,
            ArrayList itemsList) throws IOException {

    System.out.println("Entering UserAccessController.displayAllItems()");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
    out.println("<html><title>List of Office Supply Store Inventory:</title>"
);
        out.println("<form method=\"get\">");
    out.println("<body><h2>The Office Supply Store Inventory (servlet) <h2></
body>");
        if (itemsList.isEmpty()) {
            out.println("<p><h3><b>No Items are available!<b><h3>");
        } else {
            out.println("<p><table border=\"1\">");
```

```
            out.println("<th><b>ItemID<b>");
            out.println("<th><b>Description<b>");
            out.println("<th><b>Quantity<b>");
            out.println("<th><b>Price<b><tr>");
            Iterator items = itemsList.iterator();
            StoreInventory item = null;
            while (items.hasNext()) {
                item = (StoreInventory) items.next();
                out.println("<tr><td>");
                out.println(item.getItemnum());
                out.println("</td><td>");
                out.println(item.getDescription());
                out.println("</td><td>");
                out.println(item.getQtyonhand());
                out.println("</td><td>");
                out.println(item.getPrice());
                out.println("</td></tr>");
            }
            out.println("</table>");
        }
        out.println("</form>");
        out.println("</html>");
        if (out != null) {
            out.close();
        }
        System.out.println("Leaving UserAccessController.displayAllItems()");
    }
    private void displayLoginErrorScreen(HttpServletResponse response)
            throws IOException {
    System.out.println("Entering UserAccessController.displayLoginErrorScreen
()");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><title>OfficeSupplyStore Login Error</title>");
        out.println("<h3>Please try again, your login has failed!");
        out.println("<form method=\"get\">");
        displayLoginDataFields(out);
        out.println("</form></html>");
        if (out != null) {
            out.close();
        }
        System.out

    .println("Leaving UserAccessController.displayLoginErrorScreen()");
    }
}
```

11.2.3. Displaying Inventory

Add HTML and JSP tags as follows to display the inventory, where buildURL has the link to the JSP:

```
protected void processRequest(HttpServletRequest request,
         HttpServletResponse response)
              throws ServletException, IOException {
            // servlet display code (module 6A)
            //displayAllItems(response, thestore);
            //
            // jsp display code
            session.setAttribute("InventoryList", thestore);
            RequestDispatcher rd = getServletContext().getRequestDispatcher("/
showStoreItems.jsp");
                rd.forward(request, response);
                //
```

11.2.4. Deploying Web Client Components

OfficeSupplyStoreWeb required a modification to UserAccessBean, which needs to be deployed to the server. Refer to the procedure in 2.11 to publish the EJB. The JBoss AS state changes from Republish to Synchronized.

11.2.5. Testing Web Client Components

To execute the Web Client, perform the following steps:

1. From the browser, access the servlet using the following URL

   http://localhost:8080/OfficeSupplyStoreWeb/signon

   where **signon** is the URL mapping that was assigned while creating the servlet using the servlet creation wizard, and **OfficeSupplyStoreWeb** is the Web module where this servlet UserAccessController resides.

   The log-in window appears.

   Notes:

   - *Log-in entries are case sensitive.*
   - *If the log-in window does not appear, try cleaning the project and restarting JBoss AS.*
2. Type **sa** in the **Username** box.
3. Leave the **Password** box blank.
4. Click **login**.

The Office Supply Store Inventory window appears, as illustrated in Figure 11–2.

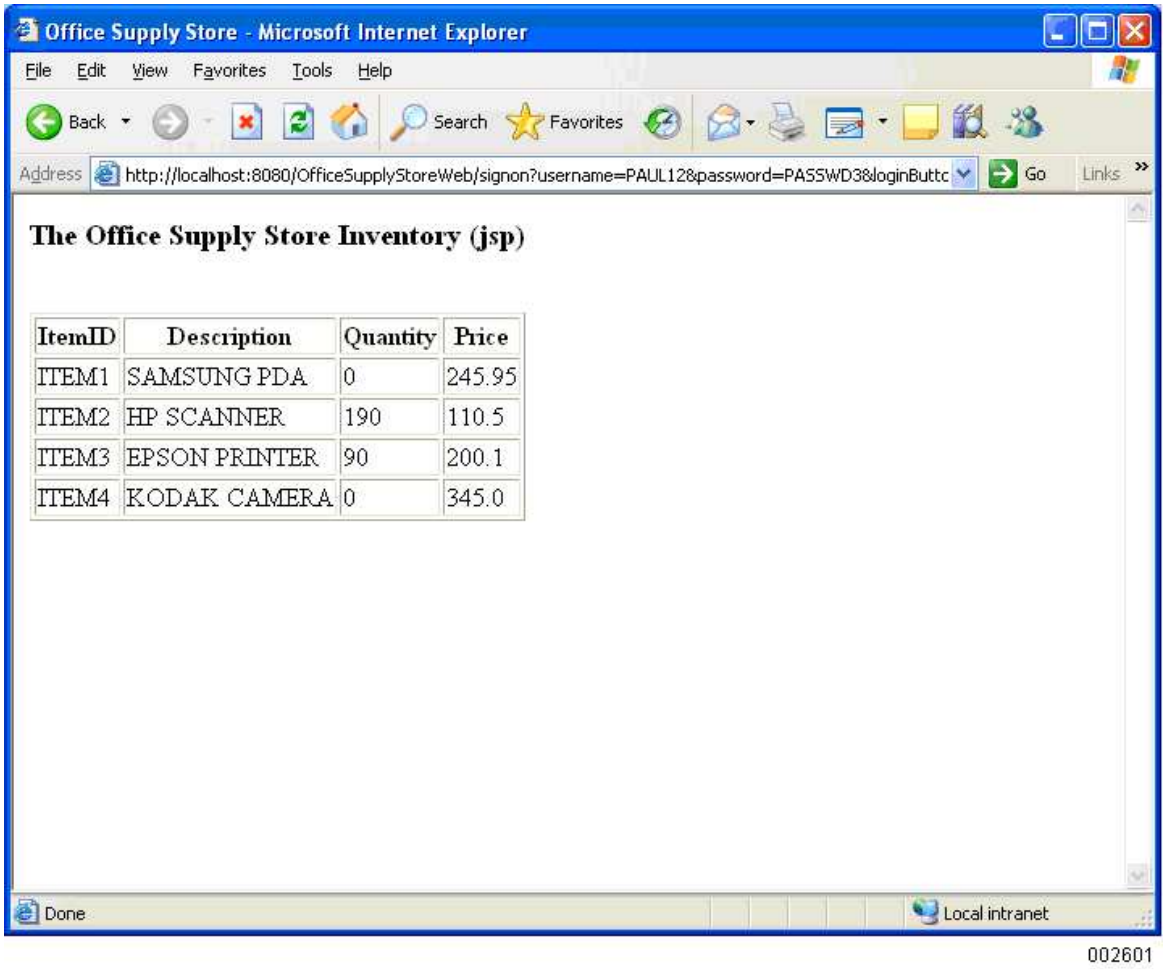

Figure 11–2. Office Supply Store Inventory from the JSP

Section 12
Creating Web Services

Web services technology enables integration among enterprise applications, including those from different vendors and different platforms, using XML to exchange data.

12.1. Web Services Overview

Historically, several initiatives to provide vendor-and-platform integration technology occurred, but none were successful. An example initiative would be getting an enterprise application that runs on a Windows operating system to communicate with an enterprise application that runs on a Unix operating system. While it can be done, it typically uses a technology that is not equally supported in both environments.

However, the development of Web services technology changed this situation. Simple Object Access Protocol (SOAP) and eXtensible Markup Language (XML) provide the necessary integration techniques. The basic mechanism of Web services uses XML to transport information between different applications, using the standard Hypertext Transfer Protocol (HTTP) of the Web environment.

Refer to Appendix C for information about Web Services standards.

Section 13
Creating Message-Driven Beans

A message-driven bean (MDB) is an EJB component that combines features of a session bean and a Java Message Service (JMS) message listener. The EJB container asynchronously invokes an MDB when it receives a message from the JMS queue.

13.1. Accessing Office Supply Store

The Office Supply Store application uses two MDBs (refer to 6.6.1), as illustrated in Figure 13–1. The RequestStoreItems bean sends requests to various suppliers to deliver items that are out of stock. The DeliverStoreItems bean replenishes the stock (quantity on hand) of an item.

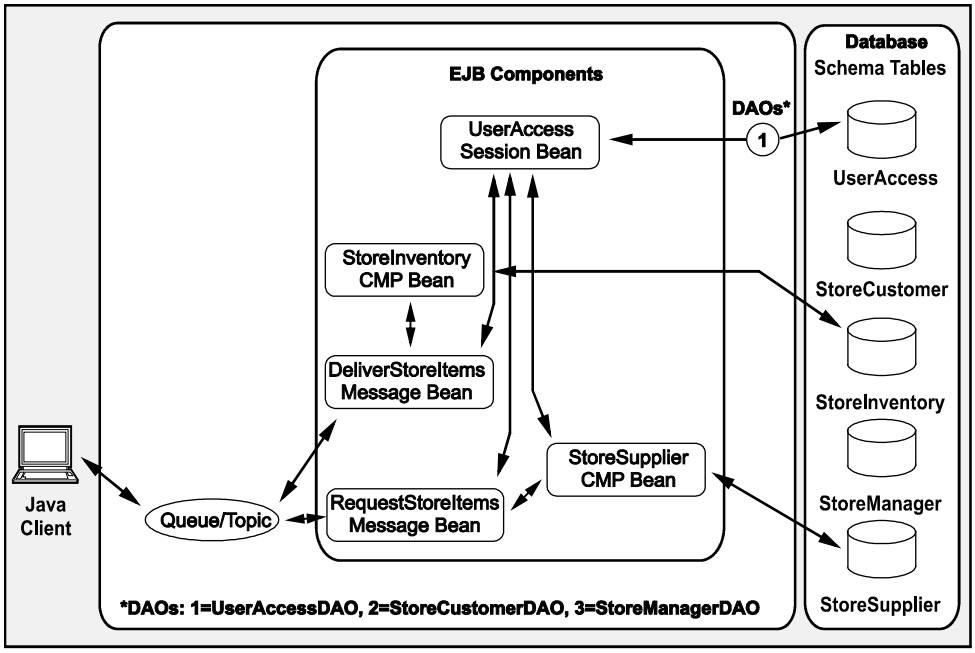

Figure 13–1. MDB Beans in Office Supply Store

13.1.1. Test Applications

Two test applications for the OfficeSupplyStore issue (send) messages to the message-driven beans, causing them to perform the stated actions.

13.1.2. Remote and Local Access to Beans

Both MDBs access the UserAccess bean through its remote interface, even though UserAccess is in the same JVM. This is because UserAccess is implemented as a remote bean, so it exposes only its remote interface. However, StoreInventory and StoreSupplier are in the same JVM as the MDBs and their local interfaces are exposed, so the MDBs can use their local interfaces when accessing StoreInventory and StoreSupplier.

13.2. Tasks

Complete the following tasks to create a message-driven bean (MDB). Before beginning these tasks, be sure that both the StoreInventory and StoreSupplier beans are generated (refer to Section 9). Alternatively, you can use a completed workspace from the examples folder.

1. Create a message-driven bean (MDB) structure named RequestStoreItems.
2. Create an immutable value object named RequestStoreItem.
3. Implement the onMessage method in RequestStoreItemsBean.java.
4. Create a test client named RequestMDBTestClient.
5. Run the client and test the RequestStoreItems bean.

Subsequently, you also need to create the DeliverStoreItems MDB and test client DeliverMDBTestClient (refer to 13.3).

13.2.1. Creating a Message-Driven Bean Structure

To create a message-driven bean (MDB) structure named RequestStoreItems

1. In the Java EE perspective Project Explorer, right-click the **OfficeSupplyStoreEJB** project, point to **New**, and click **Other**.

   The **New** dialog box for selecting a wizard appears.
2. Expand **EJB**, and click **Message-Drive Bean (EJB 3.x)**, and then click **Next**.

   The Create EJB 3.x Message Driven Bean window appears.
3. Type **us.com.unisys.mdb** in the **Java package** box (do not try to browse for this because it is not defined yet).
4. Type **RequestStoreItemsBean** in the **Class name** box.

5. Be sure the following values are in the boxes and click **Next**:
   - Project name is OfficeSupplyStoreEJB.
   - Folder is \OfficeSupplyStoreEJB\ejbModule.
   - Superclass is java.lang.Object.
6. Note that **RequestStoreItems** appears in the **Destination JNDI Name** box. (This name is used later.)
7. Click **Finish**.

Contents of MDB

The EJB creation wizard generates certain methods and tags but fewer classes for an MDB (refer to Figure 13–2).

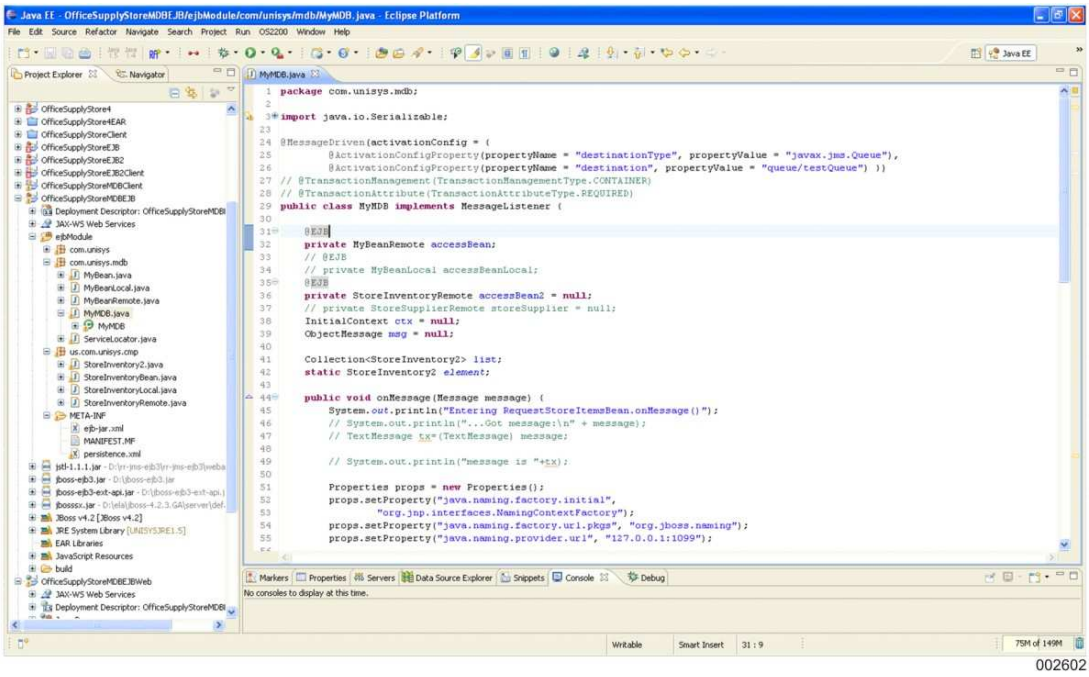

Figure 13–2. MDB Contents

Message-driven beans listen for messages from a JMS listener, which gets information from a producer (perhaps another bean) and transfers it to the relevant consumer bean. Because an MDB is responsible only for processing messages, it does not need helper classes, such as Remote and RemoteHome interfaces, Util classes, or a DAO class, as other types of beans do. The only helper classes needed are immutable value objects, which are responsible for holding the information that is extracted from messages and transferred to the beans.

@ejb.bean Tag

The EJB creation wizard creates one @ejb.bean tag that assigns the name, transaction type, destination type, and other properties, as follows:

```
/*
 * Created on Jun 22, 2004
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
 package us.com.unisys.mdb;
 import javax.ejb.MessageDrivenBean;
 import javax.jms.MessageListener;
```

The MDB has a method named onMessage for writing all business logic, as follows:

```
/**
 * This method implements the business logic for the EJB.
 *
 * <p>Make sure that the business logic accounts for asynchronous message
 * processing. For example, it cannot be assumed that the EJB receives
 * messages in the order they were sent by the client. Instance pooling
 * within the container means that messages are not received or processed in
 * a sequential order, although individual onMessage() calls to a given
 * message-driven bean instance are serialized.
 *
 * <p>The <code>onMessage()</code> method is required, and must take a
 * single parameter of type javax.jms.Message. The throws clause (if used)
 * must not include an application exception. Must not be declared as final
 * or static.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 */
 public void onMessage(javax.jms.Message message) {
   // begin-user-code
   System.out.println("Message-Driven Bean got message " + message);
   // TODO:  do business logic here
   // end-user-code
 }
```

13.2.2. Creating Immutable Value Objects

A message is received from a JMS producer as a message object. Data from the message is extracted, written to an immutable value object, and transferred to the relevant bean.

To create a class or immutable value object named RequestStoreItem for extracting information from the received message

1. Right-click the **us.com.unisys.mdb** package name, point to **New**, and click **Class**.

   The **New Java Class** dialog box opens with the following values in the boxes:

   - Source folder is \OfficeSupplyStoreEJB\ejbModule.
   - Package is us.com.unisys.mdb (do not try to browse for this because it is not defined yet).
   - Name is RequestStoreItem.
   - Superclass is java.lang.Object.
2. Click **Add** and type **Serializable** (not the fully qualified name).
3. Select the **Inherited abstract methods** check box.
4. Click **Finish**.

RequestStoreItem Class

To complete the RequestStoreItem class, copy a completed version from the following folder to a similarly named location in your workspace:

```
...\examples\13\OfficeSupplyStoreEJB\ejbModule\us\com\unisys\mdb\
RequestStoreItem.java
```

Attributes

Add the following attributes to the RequestStoreItem class:

```
public class RequestStoreItem implements Serializable {
private String username;
private String password;
private String itemNumRequested;
private int quantityRequested;
}
```

Getter, Setter Methods

Use a wizard to generate getter and setter methods for the attributes. From the Java perspective, right-click the RequestStoreItem file, point to **Source** and then **Generate Getters and Setters**, and click **Generate Getters and Setters for all attributes**.

Constructor

Use a wizard to add a constructor for the class. From the Java perspective, right-click the RequestStoreItem file, point to **Source** and then **Generate Constructor using Fields**, and click **Generate Constructor for all attributes**.

The constructor has four parameters with the same type as the attributes defined previously. The wizard generates code in the constructor to assign the parameter values to the attributes.

Complete the RequestStoreItem constructor, as follows:

```
public RequestStoreItem(String username, String password,
    String itemNumRequested, int quantityRequested) {
 super();
 this.username = username;
 this.password = password;
 this.itemNumRequested = itemNumRequested;
 this.quantityRequested = quantityRequested;
}
```

13.2.3. Implementing onMessage Method

Complete the RequestStoreItems bean as identified in the following paragraphs. You can copy a completed version of this class from the following file in the examples folder to a similarly named location in your workspace.

```
...\examples\13\OfficeSupplyStoreEJB\ejbModule\us\com\unisys\mdb\
RequestStoreItemsBean.java
```

Implement the onMessage method in the RequestStoreItemsBean class in the file RequestStoreItemsBean.java. This method extracts the information from the message and transfers it to the relevant bean. The Office Supply Store test program generates a request to a supplier by specifying an item (itemNumRequested) and a quantity (quantityRequested) required. This request arrives at the onMessage method.

The following imports are needed:

```
import java.io.Serializable;
import java.util.Collection;
import java.util.Iterator;
import java.util.Properties;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.EJB;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;
import javax.naming.InitialContext;
import us.com.unisys.cmp.StoreInventory2;
import us.com.unisys.cmp.StoreInventoryRemote;
import com.unisys.RequestStoreItem;
```

onMessage Method

The complete code of the onMessage method follows. In the method, data is extracted from the message and placed in the RequestStoreItem object. Individual fields from the immutable value object can be accessed and data passed to the appropriate beans to validate the user and get the inventory.

```
@EJB
private MyBeanRemote accessBean;
@EJB
private StoreInventoryRemote accessBean2 = null;
InitialContext ctx = null;
ObjectMessage msg = null;
Collection<StoreInventory2> list;
static StoreInventory2 element;
public void onMessage(Message message) {
    System.out.println("Entering RequestStoreItemsBean.onMessage()");
    Properties props = new Properties();
    props.setProperty("java.naming.factory.initial",
    "org.jnp.interfaces.NamingContextFactory");
    props.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming");
    props.setProperty("java.naming.provider.url", "127.0.0.1:1099");
    try {
        if (message instanceof ObjectMessage) {
        // msg = (ObjectMessage) message;
        Serializable seObj = ((ObjectMessage) message).getObject();
        RequestStoreItem person = (RequestStoreItem) seObj;
        // RequestStoreItem e = (RequestStoreItem) msg.getObject();
    System.out.println("Got message in MyMDB :\n"+ person.getUsername());
    System.out.println("Got message in MyMDB :\n"+ person.getPassword());
    ctx = new InitialContext(props);
    accessBean = (MyBeanRemote) ctx.lookup("OfficeStoreMDB/MyBean/remote");
    System.out.println(accessBean);
```

```
    String id = accessBean.verifyThisUser(person.getUsername(),
    person.getPassword());
    System.out.println("elan is here" + id);
    accessBean2 = (StoreInventoryRemote) ctx.lookup("OfficeStoreMDB/StoreInve
ntoryBean/remote");
    if (id != null) {
    String inventoryID = person.getItemNumRequested();
    System.out.println("inventoryID is " + inventoryID);
    accessBean2.addInventory("ITEM1", "SUPL1", "her", "23",(float) 12.9);
    list = accessBean2.getStoreInventoryData(inventoryID);
    for (Iterator iter = list.iterator(); iter.hasNext();) {
        element = (StoreInventory2) iter.next();
    }
System.out.println("supplier id is : "+ element.getSupplierid());
    }
    }
    } catch (Exception e) {
System.out.println("Error in RequestStoreItems.onMessage() "+ e.getMessage()
);
e.printStackTrace();
}
System.out.println("Leaving RequestStoreItemsBean.onMessage()");
}
```

13.2.4. Creating a Test Client

Use the procedure in 8.3.5 to create a test client with a new class called RequestMDBTestClient in the project OfficeSupplyStoreClient with a package name,us.com.unisys.client.

To save time, copy a completed version of the test client from the following location to a similarly named location in your workspace:

```
...\examples\13\OfficeSupplyStoreClient\appClientModule\us\com\unisys\client\Re
questMDBTestClient.java
```

The following imports are needed:

```
import java.io.IOException;
import java.util.Date;
import java.util.Properties;
import javax.jms.ObjectMessage;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

```
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.unisys.RequestStoreItem;
```

getContext Method

Insert the following method to get the instance, as follows:

```
private InitialContext getContext() throws NamingException {
      Hashtable icProperties = new Hashtable();
      icProperties.put( InitialContext.INITIAL_CONTEXT_FACTORY,
                     "org.jnp.interfaces.NamingContextFactory");
      icProperties.put( InitialContext.PROVIDER_URL,
                     "jnp://127.00.0.1:1099");
      InitialContext initialContext = new InitialContext(icProperties);
      return initialContext ;
   }
```

testRequestMDB Method

Add the testRequestMDB method to test the MDB. The method performs the following steps:

1. Add a data object that is to be sent as the message.
2. Create the initial context reference.
3. Create the connection factory reference.
4. Use this context to perform the lookup, where the lookup string is **queue/RequestStoreItemsMessageQueue**.
5. Create the queue connection.
6. Create the queue sender.
7. Create the queue session for the bean.
8. Create the object message and pass the data object in the message.
9. Send the message.
10. Commit the session and close both the session and the connection.

The testRequestMDB method has the following code:

```
public void testRequestMDB() {
      QueueConnectionFactory factory = (QueueConnectionFactory) getContext()
.lookup("ConnectionFactory");
    QueueConnection connection = factory.createQueueConnection();
    //QueueSession session = connection.createQueueSession(true, 1);
    QueueSession session = connection.createQueueSession(true, Session.AUTO_A
CKNOWLEDGE);
    Queue queue = (Queue) getContext().lookup("queue/testQueue");
    // System.out.println("Creating the sender now...");
    QueueSender sender = session.createSender(queue);
    RequestStoreItem content = new RequestStoreItem(new Date());
    content.setUsername(username);
    content.setPassword(password);
    content.setString("This is new title.");
    ObjectMessage message = session.createObjectMessage();
    //message.setIntProperty("sessionId", 501);
    message.setObject(content);
    System.out.println("Setting the object in message now...");
    //message.setObject(ri);
    //message.setObject(username);
    System.out.println("Reading ObjectMessage:");
    System.out.println(" " + message.getObject());
    System.out.println("Sending the message.");
    sender.send(message);
    // System.out.println("Shutting down.");
    session.commit();
    connection.close();
    session.close();
    // System.out.println("Done...");
    } catch (Exception e) {
    e.printStackTrace();
    }
```

main Method

Add a main method that enables running this class as an application, as follows:

```
public static void main(String[] args) {
   RequestMDBTestClient test = new RequestMDBTestClient();
   test.testRequestMDB();
}
```

13.2.5. Testing the Bean

To run the client and test the bean

1. Start JBoss AS.
2. Right-click the **RequestMDBTestClient** node, point to **Run As**, and click **Java Application**.

The output window should have content like Figure 13–3.

```
11:43:19,837 INFO  [EJB3Deployer] Deployed: file:/D:/ela/jboss-4.2.3.GA/server/de
fault/tmp/deploy/tmp69573566899399001740fficeStoreMDB.ear-contents/OfficeSupplySt
oreMDBEJB.jar
11:43:19,837 INFO [TomcatDeployer] deploy, ctxPath=/OfficeSupplyStoreMDBEJBWeb, w
arUrl=.../tmp/deploy/tmp69573566899399001740fficeStoreMDB.ear-contents/OfficeSupp
lyStoreMDBEJBWeb-exp.war/
11:43:19,899 INFO [EARDeployer] Started J2EE application: file:/D:/ela/jboss-4.2.
3.GA/server/default/deploy/OfficeStoreMDB.ear
11:43:19,946 INFO [Http11Protocol] Starting Coyote HTTP/1.1 on http-127.0.0.1-808
0
11:43:19,946 INFO [AjpProtocol] Starting Coyote AJP/1.3 on ajp-127.0.0.1-8009
11:43:19,962 INFO [Server] JBoss (MX MicroKernel) [4.2.3.GA (build: SVNTag=JBoss_
4_2_3_GA date=200807181439)] Started in 11s:334ms
11:43:30,109 INFO [STDOUT] username is : sa
11:43:30,109 INFO [STDOUT] password is : sa
11:43:30,187 INFO [STDOUT]  Timezone :=  -330
11:43:30,203 INFO [STDOUT] Setting the object in message now...
11:43:30,203 INFO [STDOUT] Reading ObjectMessage:
11:43:30,203 INFO [STDOUT]  com.unisys.RequestStoreItem@ad00b2
11:43:30,203 INFO [STDOUT] Sending the message.
11:43:30,250 INFO [STDOUT] Entering RequestStoreItemsBean.onMessage()
11:43:30,250 INFO [STDOUT] Got message in MyMDB :
sa
11:43:30,250 INFO [STDOUT] Got message in MyMDB :
sa
11:43:30,250 INFO [STDOUT] jboss.j2ee:ear=OfficeStoreMDB.ear,jar=OfficeSupplyStor
eMDBEJB.jar,name=MyBean,service=EJB3
11:43:30,296 INFO [STDOUT] Entering UserAccessDAOImpl.init()
11:43:30,296 INFO [STDOUT] i am here in Bean class using Resource annotation
11:43:30,296 INFO [STDOUT] StoreAccessID is 100
11:43:30,296 INFO [STDOUT] Leaving UserAccessDAOImpl.verifyThisUser()
11:43:30,312 INFO [STDOUT] inventoryID is ITEM1
11:43:30,312 INFO [STDOUT] before begin inside addBook
11:43:30,374 INFO [STDOUT] invId is *******ITEM1
11:43:30,499 INFO [STDOUT] supplier id is : SUPL1
11:43:30,499 INFO [STDOUT] Leaving RequestStoreItemsBean.onMessage()
```

Figure 13–3. Client Output from the MDB

Appendix A
Best Practices

This appendix contains recommended best practices for working with Eclipse IDE, JavaDoc and XDoclet, and jar files.

A.1. Importing Eclipse IDE Projects

The following procedure is recommended for importing projects:

1. Copy the related projects to a new folder that becomes the new workspace.
2. Remove the metadata folder from the root, if it was not created on your PC.
3. Start Eclipse IDE.
4. When asked for the workspace, point to the folder created in step 1.

   Alternatively, switch the workspace to point to a new workspace (on the **File** menu, click **Switch Workspace**).
5. Import previously exported preferences (refer to A.2.2), or set them manually.
6. Import the projects.
   a. On the **File** menu, point to **Import**, and click **Existing Projects into Workspace**.
   b. Click **Browse**, use the default location (the workspace), and click **OK**.
   c. Select the desired projects.
   d. Click **Finish**.

A.2. Using Eclipse IDE

A.2.1. Workspace Preferences

A new workspace has new preferences; for example, JBoss AS and XDoclet definitions are not set. To avoid resetting all the preferences each time you create a new workspace, export your preferences once they are set up correctly, and then import the preset preferences to the new workspace.

To export or import preferences

- For exporting, on the **File** menu, point to **Export**, and click **Preferences**.
- For importing, on the **File** menu, point to **Import**, and click **Preferences**.

A.2.2. Sharing Projects or Workspaces

You can share projects with other PCs, but do not share workspaces. Workspaces have unique PC installation information, which can be different on different PCs. Set up your own workspace and import projects (refer to A.1).

A.2.3. Unrelated Projects

To avoid confusion, keep unrelated projects in separate workspaces.

Alternatively, disable an unrelated project by closing it (on the **Project** menu, click **Close Project**).

A.2.4. Removing Workspaces

To safely remove workspace-unique information, delete the entire metadata folder.

A.3. Using JavaDoc and XDoclet

JavaDoc is a method of generating documentation from comments in the source code. JavaDoc comment blocks have a specific format and grammar.

Attribute-oriented programming (XDoclet) extends JavaDoc to generate code. The process of creating Java bean code uses XDoclet tags extensively. Some XDoclet tags are generated automatically by Web Tools wizards. Other XDoclet tags are added manually as bean code is developed. At compile time, XDoclet tags are expanded to generate numerous supporting classes in beans.

A.3.1. Coding JavaDoc and XDoclet

JavaDoc comment blocks start with /** to distinguish them from regular Java comments, which start with /*. Declarations begin with "*@keyword*" tags. Be sure to code XDoclet tags in JavaDoc comment blocks correctly by adhering to the following rules:

- XDoclet tags must occur in JavaDoc comment blocks.
- Nothing other than white space can occur between a JavaDoc comment block and the interface or business method it describes.

Caution

If any code, even a Java comment block, occurs between a JavaDoc comment block and the business method it describes, the association between the XDoclet tags and the business method is lost.

A.3.2. XDoclet Grammar Documentation

XDoclet grammar is documented at the following location:

`http://xdoclet.sourceforge.net/xdoclet.`

A.4. Deploying Jar Files with JBoss Application Server

The example in this guide uses JBoss Application Server (JBoss AS) to deploy the application. Although you can start JBoss AS from a command line prompt or from Eclipse IDE (refer to Section 2), deploying and debugging the application is easiest from Eclipse IDE.

Appendix B
Troubleshooting

These troubleshooting tips are summarized from other subsections in this document.

B.1. Compilation Errors

If you get unexplained compilation errors, try cleaning the project (on the **Project** menu, click **Clean**).

Some errors require several cleaning operations.

B.2. JBoss AS Startup Errors

The following errors can occur when JBoss AS starts.

B.2.1. Port Number Conflicts

If you get port number conflicts when starting JBoss AS, try rebooting the PC because Windows sometimes uses the JBoss AS port number.

B.2.2. Deployment Errors

If you get JBoss AS deployment errors on startup, stop JBoss AS and clean the project deployments by doing the following:

1. Remove all project-related items in the following folder:

   ```
   C:\jboss420GAu\server\default\deploy
   ```

   For example, delete the following items, as shown in Figure B–1:

   - MyEJBProjEAR.ear
   - OfficeSupplyStore.ear
   - OfficeSupplyStoreWebEAR.ear

2. Completely delete the following folders:

   ```
   C:\jboss420GAu\server\default\tmp
   C:\jboss420GAu\server\default\work
   ```

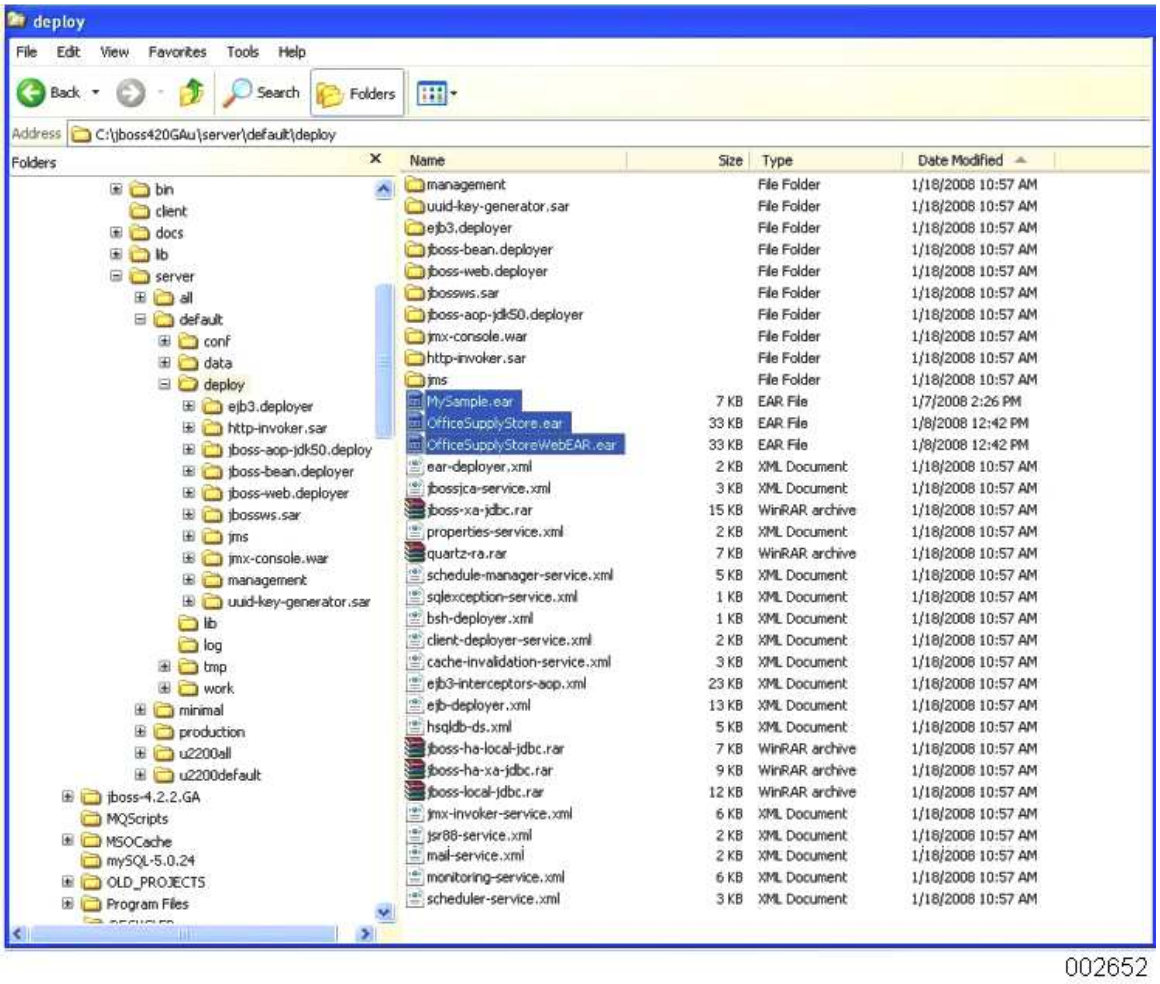

Figure B–1. Items to Delete in the Deployment Folder

Appendix C
Web Services Standards

Web services technology enables integration among enterprise applications, including those from different vendors and different platforms, using XML to exchange data.

C.1. Web Services Standards

Some of the evolving standards for Web services are

- Simple Object Access Protocol (SOAP). A lightweight XML-based Remote Procedure Call (RPC) over HTTP. The SOAP protocol has the following parts:
  - A SOAP envelope that defines a framework for describing what is in a message and how to process it.
  - A set of encoding rules for expressing instances of application-defined data types.
  - A set of rules for representing remote procedure calls and responses.
- Web Services Description Language (WSDL), developed by Microsoft and IBM. An XML-based language for defining Web services that describes the protocols and formats used by the service. Can be used with UDDI.
- Universal Description Discovery and Integration (UDDI). An industry initiative for a universal business registry (catalog of the network of Internet servers that is similar to the Domain Name System) of Web services. UDDI enables software to discover and integrate with services on the Web automatically.
- Electronic Business using XML (ebXML), promoted by the Organization for the Advancement of Structured Information Standards (OASIS) and the United Nations Center for Trade Facilitation and Electronic Business (UN/CEFACT). A framework for developing an XML-based business transaction vocabulary. Designed for global interoperability, ebXML provides descriptors for modeling business processes that include the definition of software components.

These standards are evolving under the direction of the World Wide Web Consortium (W3C) and are regarded as the building blocks of the next generation of distributed computing and the information technology (IT) industry in general.

C.2. Using SOAP and WSDL

Web Services Description Language (WSDL) describes the interface of a network service. WSDL is similar to Interface Definition Language (IDL) in Common Object Request Broker Architecture (CORBA). WSDL specifies which messages an endpoint receives and sends, where

- In the terms of similar mechanisms like Remote Method Invocation (RMI) or CORBA, a service is the interface or set of methods that a client invokes across the network.
- An endpoint is a service instance that processes client requests and returns responses.

WSDL is expressed in XML metadata format.

Simple Object Access Protocol (SOAP) describes the format of the data that is transmitted over the network. Even though SOAP messages are complex, this complexity is transparent to developers because SOAP messages are generated by the JAX-RPC API calls (refer to 12.4). Developers do not generate SOAP messages for client and server-side implementations.

C.3. Web Services in the Java EE Environment

Web services are fully supported in the Java EE 1.4 release from Sun Microsystems.

In the Java EE environment, Web services are built on the Java API for XML-based RPC (JAX-RPC). This API enables building Web services and clients that use remote procedure calls (RPC) and XML. In the JAX-RPC API

- An RPC is represented by an XML-based protocol, such as SOAP.
- The SOAP specification defines the envelope structure, encoding rules, and conventions for representing RPCs and responses.
- Calls and responses are transmitted as SOAP messages (XML streams) over HTTP.
- On the server side, developers specify the remote procedures by defining methods in an interface written in the Java programming language.
- Developers also code one or more classes that implement those methods.
- Client programs are also easy to code. A client creates a proxy, a local object representing the service, and then invokes methods on the proxy.

Using the JAX-RPC API, the developer does not generate or parse SOAP messages. The JAX-RPC run-time system converts the API calls and responses to and from SOAP messages.

Glossary

A

applet
A small client application that is written in the Java programming language and executes in the Java Virtual Machine (JVM) that is installed in the Web browser.

application client
A program module that runs on a client machine and provides a way to handle tasks that require a richer user interface than a markup language can provide. *See also* Web client.

application tiers
The layers of an application that work together to provide services and process data for a business solution. *See also* business tier, client tier, enterprise information system tier, Web tier.

attribute-oriented programming (XDoclet)
An extension to JavaDoc for generating code.

B

bean
See enterprise bean, Enterprise JavaBeans (EJB), entity bean, JavaBeans, message-driven bean, session bean.

bean-managed persistence (BMP) entity bean
An EJB component in which database access is controlled manually. The developer explicitly codes database calls in the bean itself, resulting in greater flexibility in how data is read and written. *See also* container-managed persistence (CMP) bean.

BMP
See bean-managed persistence entity bean.

business tier
The layer of an application that contains components for the business logic to process data in response to calls from clients. Enterprise beans run in the business tier.

C

client tier
The layer of an application that contains components for interacting with the end user and making requests to the business tier for processing data. Web clients and application clients run in the client tier.

CMP
See container-managed persistence entity bean.

container-managed persistence (CMP) entity bean
An EJB component in which the container handles all details of data access. The developer is limited to the data management facilities provided by the Enterprise JavaBeans vendor. *See also* bean-managed persistence (BMP) bean.

Connector Module
A module containing interfaces and Application Programming Interface (API) for creating and accessing elements, such as vendor-name, spec-version, eis-type, version, license, resource adapter, and activation-spec. This module is part of the org.eclipse.jst.j2ee.jca java package. The connector interface contains the APIs for accessing the elements of a connector module. The ResourceAdapter interface contains the APIs for accessing the elements of the internals of a connector module.

D

DAO
See data access object.

data access object (DAO)
A wrapper for code that provides an abstraction layer for data access so that changing the actual storage does not affect the rest of the code logic.

DNS
Also known as Domain Name System (DNS). DNS is a standard technology for managing the names of Web sites and other Internet domains. DNS is a worldwide collection of DNS servers and it enables you to find Web addresses on the Internet.

E

ear file
See enterprise archive file.

EJB
See Enterprise JavaBeans.

EJB component
A component in an Enterprise JavaBeans implementation (enterprise bean).

EJB container
A wrapper for Enterprise JavaBeans for Java EE applications that provides distributed application functionality, such as transaction support, persistence, and lifecycle management. *See also* Java EE container, Web container.

Electronic Business using XML (ebXML)
A framework for developing an XML-based business transaction vocabulary that is designed for global interoperability.

enterprise archive (ear) file
A file for a Java EE application that contains all the components to be deployed on the server. The ear file also contains a deployment descriptor that provides information about the application and its assembled components.

enterprise bean
An instance, type, or implementation of EJB components that are deployed on the server.

enterprise information system (EIS) tier
The layer of an application that contains all enterprise information system software and enterprise infrastructure systems, such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and legacy information systems.

Enterprise JavaBeans (EJB)
A component architecture for developing and deploying component-based applications for a particular business domain, such as banking or finance. A server-side model, Enterprise JavaBeans simplifies the development of middleware applications by providing automatic support for services such as transactions, security, and database connectivity. *See also* enterprise bean.

entity bean
An EJB component that is persistent data stored in one row of a database relation or table. If the client terminates or the server shuts down, the underlying services ensure that the entity bean data is saved.

Extensible Markup Language (XML)
An open standard from the W3C for defining data elements on Web pages and business-to-business documents. XML uses a tag structure that defines what an element contains, as determined by the developer of the page.

I

immutable value object
A holder for information that is extracted from messages (producers) and transferred to beans (consumers) for processing.

Integrated Development Environment (IDE)
A set of programs that can run from a single user interface. For example, programming languages can include a text editor, compiler, and debugger, which are activated from a common interface.

IP Address
The unique identifier of a computer that is used to send data to other computers on a network through different protocols, such as TCP/IP. The 127.0.0.1 IP address is the loopback Internet protocol (IP), also referred to as the "localhost." This address is used to establish an IP connection to the same machine or computer being used by the end-user.

J

Java EE
A Java EE project includes Enterprise Application, Web, Application Client, Enterprise Java Bean, and Connector projects. The Java EE project can be created for Java EE specification levels 1.2, 1.3, and 1.4. A Java EE run-time target (target server) should be predefined for project creation. The run-time target is a mechanism to set the JRE and server classpath on a Java EE project for compile time. The Java EE module projects, such as Web, Application Client, Enterprise JavaBean, and Connector can be created as standalone or nested under a new or an existing enterprise application project.

Java EE container
The interface between a component and the low-level platform-specific services that support the component, such as transaction processing, state management, multithreading, and resource pooling. *See also* EJB container, Web container.

Java EE perspective
A presentation of an Eclipse workspace that displays the user interface for developing Java EE projects. At any time, you can switch the perspective to view the workspace in a different manner, such as viewing a Java EE project from the Java perspective.

jar file
See Java archive file.

Java API for XML-based RPC (JAX-RPC)
An API that enables building Web services and clients that use remote procedure calls (RPC) and XML in the Java EE environment.

Java archive (jar) file
A file that contains any number of related files, which can include class files, resource files, XML, other jar files, and any related file.

JavaBeans
A specification for independent Java program modules that are called by applications in the Java environment. JavaBeans are used primarily for developing user interfaces in the client.

Java Database Connectivity (JDBC)
A technology that provides access to relational database systems.

JavaDoc
A method of generating documentation from source code comments. JavaDoc comment blocks start with /**.

Java Message Service (JMS)
A message listener that allows a business component to receive JMS messages asynchronously.

Java Naming Directory Interface (JNDI)
An API that generically accesses naming and directory services using Java technology.

Java perspective
A presentation of an Eclipse workspace that displays the user interface for developing Java projects. *See also* Java EE perspective.

Java project
A collection of Java components that make up a single application tier or a portion of an application tier. *See also* Java EE project.

JavaServer Pages (JSP)
A text-based document that executes like a servlet but allows a natural language approach to creating static content. JSPs are Java EE Web components. *See also* servlet.

Java Transaction API (JTA)
A technology that provides transaction support for Java EE components.

Java Transaction Service (JTS)
A specification for implementing, at a high level, a Transaction Manager to support the Java Transaction API and, at a low level, the Java mapping of the OMG Object Transaction Service (OTS). A JTS Transaction Manager provides transaction services for distributed transactions to support the application server and manage resources and communications.

Java virtual machine (JVM)
An abstract computing technology that provides a platform-independent execution environment to convert Java bytecode to machine language and execute it.

JAX-RPC
See Java API for XML-based RPC.

JBoss Tools
A library of plug-ins that adds JBoss management functions to Eclipse. JBoss tools is an umbrella project for a set of Eclipse plug-ins that support JBoss and related technologies. JBoss tools also also provide support for Hibernate, JBoss AS, Drools, jBPM, JSF, (X)HTML, Seam, Smooks, JBoss ESB, JBoss Portal, and others.

JBoss AS Tools
A feature in JBoss AS (Application Server) that consists of a number of additional views for managing an installed JBoss server through the JBoss AS perspective. These additional views include the standard console and properties views and the servers view. The servers view enables installed servers to be configured, monitored, and managed.

JBoss General Availability (GA)

JBoss Application Server™ (JBoss AS) is a certified, full-featured, robust Java™ Platform, and the Enterprise Edition 5 (Java EE 5) open source application server developed by the JBoss Open Source Community.

JBoss Enterprise Application Platform (EAP)

JBoss EAP is an enterprise-ready JBoss AS version from RedHat. EAP is an integrated, tested, and certified Enterprise Platform. It includes patches, updates, SLA-based support, multi-year maintenance policies, and Red Hat Open Source Assurance. EAP provides long-term stability, supportability, and maintainability.

JDBC

See Java Database Connectivity.

JEM

Acronym for Java EMF Model Runtime Software Development Kit.

JMS

See Java Message Service.

JNDI

See Java Naming Directory Interface.

JSP

See JavaServer Pages.

JTA

See Java Transaction API.

JTS

See Java Transaction Service.

JVM

See Java virtual machine.

Localhost

The local computer on which a program is running. For example, if you are running a Web browser on your computer, your computer is the "localhost."

M

message-driven bean

An EJB component that combines features of a session bean and a Java Message Service (JMS) message listener. The EJB container asynchronously invokes an MDB when it receives a message from the JMS queue.

O

Object Transaction Service (OTS)
A specification from Object Management Group that expands the traditional transaction processing monitor model to object-oriented systems.

OMG
Acronym for Object Management Group.

OTS
See Object Transaction Service.

P

presentation tier
See Web tier.

R

Remote Method Invocation (RMI)
Java middleware technology for making requests between the client tier and business tier.

RMI
See Remote Method Invocation.

S

servlet
A Java programming language class that dynamically processes requests and constructs responses. Servlets are Java EE Web components. *See also* JavaServer Pages.

session bean
An EJB component that is a transient conversation with a client. When the client finishes executing, the session bean and its data are gone. Transactions that use stateless beans must contain all the parameters required to carry out a transaction in a single call. Transactions can use stateful session beans when executing a series of calls to perform one transaction. Stateless session beans do not retain state information once they finish, while stateful session beans retain state information between calls.

session facade
A business tier pattern that provides a uniform business service abstraction to presentation tier clients and hides the business object implementation in the lower-level beans.

Simple Object Access Protocol (SOAP)
A lightweight XML-based Remote Procedure Call (RPC) over HTTP that includes rules for describing message contents and processing, application-defined data types, and remote procedure calls and responses.

SOAP
See Simple Object Access Protocol.

U

UDDI
See Universal Description Discovery and Integration.

Universal Description Discovery and Integration (UDDI)
A universal business registry of Web services that enables software automatically to discover and integrate with services on the Web.

Unisys JBoss
The Unisys implementation of JBoss AS enables you to install and deploy JBoss EAP 5.1 on a ClearPath OS 2200 IDE for Eclipse system that includes at least one ClearPath OS 2200 JProcessor Specialty Engine (JProcessor). JBoss EAP includes JBoss 5.1A and JBoss 4.3A which runs on JProcessor.

W

war file
See Web archive file.

Web archive (.war) file
A file for Web components (servlets, JSPs, and static components, such as HTML and image files), which contains classes and files that are used in the Web tier, along with a Web component deployment descriptor.

Web client
A Java EE client that consists of dynamic Web pages, which use various markup languages, and a Web browser, which renders the pages received from the server. *See also* application client.

Web container
A wrapper that manages the execution of JSP and servlet components for Java EE applications. *See also* EJB container, Java EE container, Web tier.

Web Modules
The smallest deployable and usable unit of Web resources. A Java EE Web module corresponds to a Web application as defined in the Java Servlet specification. In addition to Web components and Web resources, a Web module can contain server-side utility classes (database beans, shopping carts, and so on) and client-side classes (applets and utility classes).

Web services
Technology that enables integration among enterprise applications, including those from different vendors and different platforms, using XML to exchange data.

Web Services Description Language (WSDL)
An XML-based language for defining Web services that describes the protocols and formats used by the service.

Web tier
The layer of an application that contains components (servlets and JSPs) for creating the interface for the client to interact with the end user. *See also* Web container.

WSDL
See Web Services Description Language.

X

XDoclet
See attribute-oriented programming.

XML
See Extensible Markup Language.

XML deployment descriptor
Information specific to a bundled component that provides a mechanism for configuring application behavior at assembly or deployment time.

Index

E

G

H

I

J

L

M

N

O

P

R

S

T

U

W

X

3839 3831–002