Tutorial for building J2EE Applications using JBOSS and ECLIPSE

Table of Contents

Preface.

This tutorial is about building Java 2 Platform, Enterprise Edition (J2EE) components using Eclipse as an Integrated Development Environment (IDE) and JBOSS as the Application Server. Tutorial covers step-by-step development of J2EE components, starting from setting up Eclipse, JBOSS and Lomboz. Lomboz uses Xdoclet (Attribute Oriented Programming) for rapid development of J2EE components. Importantly, all of the tools used in this tutorial can be downloaded free of charge, so there should be nothing stopping you!

Eclipse provides an excellent IDE, with features like refactoring and debugging. JBOSS is integrated within Eclipse using JBOSS plug-in. Lomboz is another plug-in used for building J2EE components, which provides wizards for bean creation, method creation, deployment of bean, test client creation etc. JBOSS is an integrated application server with convenient built-in components such as the Hypersonic Database and Jetty as the Web Engine. This bundling of all the key components of a J2EE environment assists beginners in learning how to develop J2EE applications. Having gained some skills and confidence, beginners can move on to tools provided by other vendors.

One of the problems J2EE developers face is that of synchronizing their code with J2EE's deployment descriptors. As development of components progresses, developers have to keep updating deployment descriptors, a generally tedious activity which can lead to other mistakes, instead of this devoting time to the business logic of the application. Xdoclet generates these interfaces and helper classes along with deployment descriptors, by parsing source files. These files are generated from templates that use the information provided in the source code and its JavaDoc tags. XDoclet allows developers to concentrate on only one Java source file per component, that is, concentrate on the business logic of the application, and the rest is done by Xdoclet.

For so many years developers have been looking for some sort of a tool which speeds up this process of development and deployment, and Xdoclet is certainly a step in that direction. Hopefully you'll find that Xdoclet makes development and deployment rapid and easy. So explore the power of Xdoclet, Eclipse and JBOSS!

About the Authors.

Glen McCallum.

Glen joined TUSC in 1990. He has extensive experience in software development for the telecommunications industry, including C, C++, Perl and Java. Outside of TUSC, Glen continues to fiddle with such technologies as he thinks are “cool”. This includes TCP/IP networking, security, cryptography, VPNs, digital video capture, Bluetooth, PDAs and J2EE, all on Linux. His wife and two young sons complete a “pure Linux” household.

Vishal Sharma.

Vishal was born in 1976 in India. He studied mechanical engineering, graduating in 1998. After working in that field for 6 months, he came to Australia and obtained a Master's degree in Computer Science. Since graduating from RMIT Melbourne in 2001 he first worked as a software engineer in Mcom Solutions developing communication protocols. Vishal joined TUSC in 2002 and has enjoyed a varied diet of different projects since then. Most recently he has been involved in building multi-tiered Web applications for the telecommunications industry, with a particular focus on J2EE technology.

Acknowledgments.

We'd like to thank some particular TUSC people: Rod Bower, Telecommunications Business Unit Manager, Sebastian Bellofiore, Delivery Manager and Marcia Abbott, Business Improvement Manager for their support and help in this initiative, and of course our colleagues for their collaboration; Simon Shields, Andrew Hendry and Ray Walford.

Vishal wishes to thank Glen McCallum for his guidance, inspiration and invaluable support in bringing this to you all.

Disclaimer.

Introduction.

Prerequisites for this tutorial.

Before you start this tutorial, you should ideally have a working knowledge of **Java technology, XML, J2EE technology and some exposure to SQL, JDBC concepts, and Xdoclet (Attribute Oriented Programming)**. Even if you are new to a lot of this then don't panic – just expect to do a bit more learning along the way!

All the examples covered in this tutorial were developed on RedHat Linux 8.0, Sun Microsystem's JDK (j2sdk1.4.1_02), using Eclipse 2.1 as the IDE, and Lomboz 2.1_02 plugin for Eclipse. JBOSS-3.2.1 is being used as an application server to deploy applications. Jetty is used as the web server and Hypersonic SQL as the database, both of which are integrated in the JBOSS application server.

Tools required for this tutorial.

You will need a current version of Java Development Kit (JDK) or Java Runtime Environment (JRE) - at least JDK 1.3 onwards. Eclipse 2.1 is required as an Integrated Development Environment (IDE) along with the Eclipse plug-in Lomboz 2.1_02 for J2EE applications development and JBOSS integration.

JBOSS-3.2.1 is being used as an application server to deploy applications, together with its embedded Jetty web server and Hypersonic SQL database as noted above.

Case Study Overview.

For this tutorial we are using a case study similar to **Inventory,** rather than using any complex scenario or going into strict database design, as **our aim is to learn how to design and develop various J2EE components using this new revolutionary approach of Attribute Oriented Programming with the help of these tools**.

We have a database schema called '**MyStore**' which is composed of five tables.

Table Supplier records details of those suppliers (many) who sell different materials to MyStore as request is sent to these suppliers from **MyStore** manager as need arises.

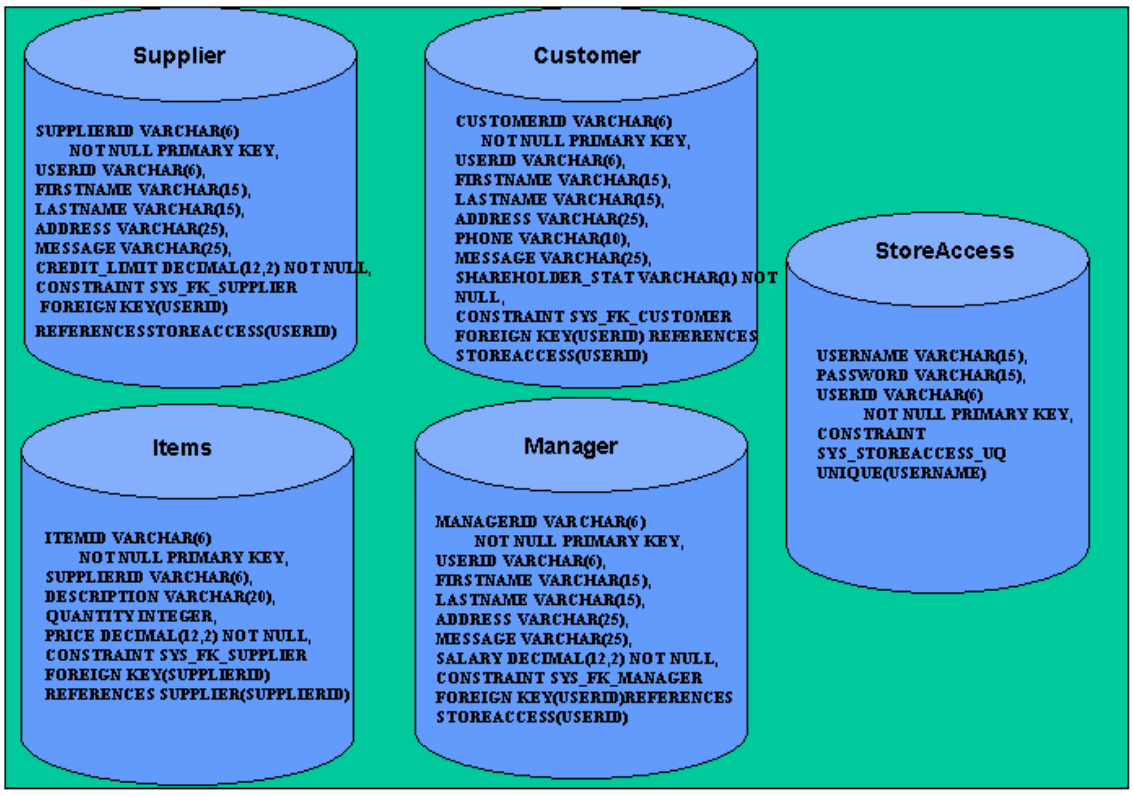
Table Manager records details of managers who run **MyStore,** currently there is only one manager.

Table Customer records details of those customers (many) who have bought some items at least once.

Table Items maintains an inventory of available/non-available items (many).

Table StoreAccess records the authentication details of all customers, suppliers and manager for on-line access of **MyStore**.

The MyStore Database is shown in the figure below.

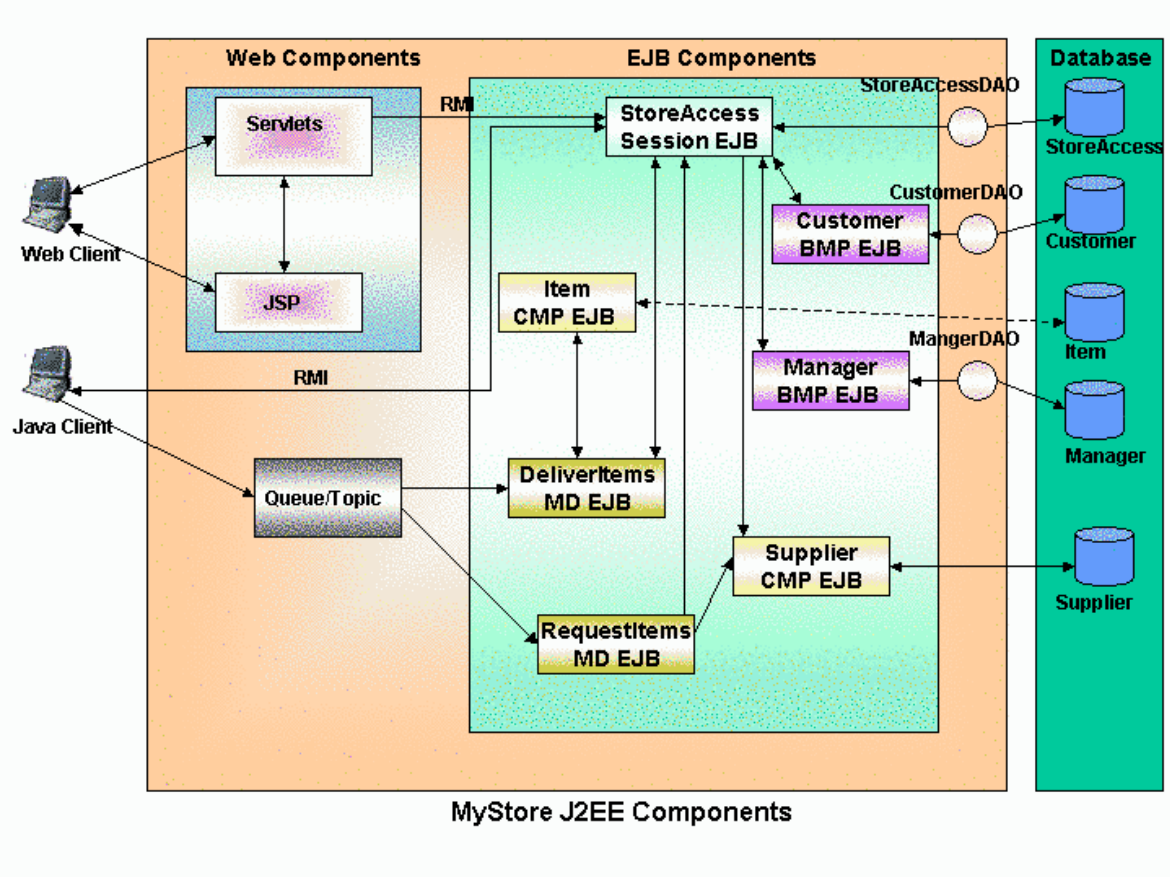


MyStore Database

To access data from the database and do business operations, we will create various J2EE components including Session, Entity and Message-driven

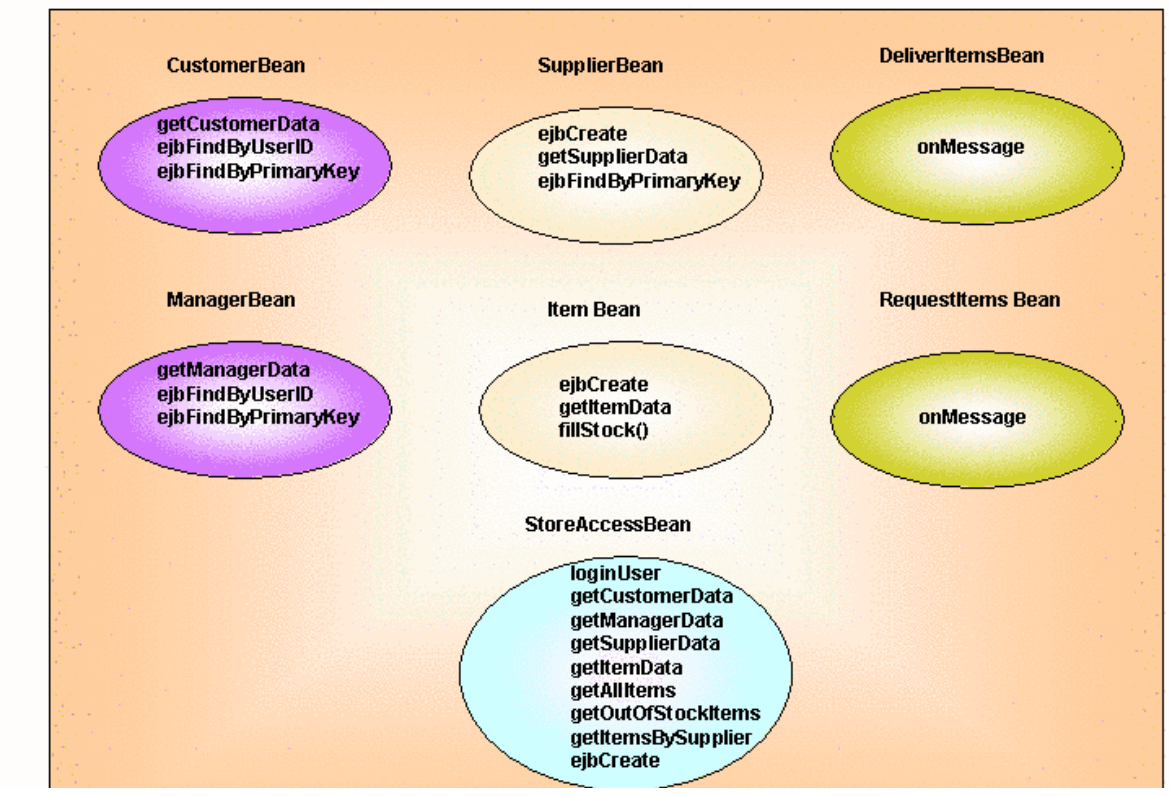
Enterprise Java Bean components along with Web clients using Servlets and JSP pages.

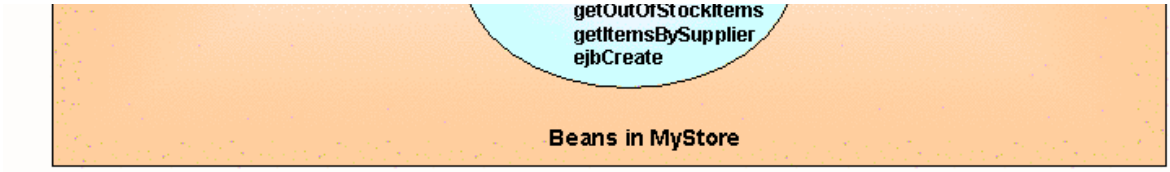
Figure below presents an overview of the MyStore application's architecture.



Note : In the case of Web clients, a request is sent by Servlets/JSP (Java Server Pages) to beans using RMI (Remote Method Invocation). Stateless, Stateful 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, while Container Managed Persistence (CMP) Entity Beans don't require a DAO, as the container manages communication between the Beans and the database, which is a very powerful feature. Finally, in the case of Java clients requests are made via Java middleware technology (RMI, CORBA, Java Messaging) to Entity Beans.

All beans together with their public behaviors/methods which will be implemented during the course of this tutorial are shown below. StoreAccessBean is a Session facade bean, which exposes its interface to the Presentation Tier, while encapsulating the complex business interactions with Customer, Manager, Item and Suppler Entity beans.





MyStore customers, suppliers and manager login into the system using the StoreAccess stateless session bean. Once authenticated, they request information about MyStore's inventory, manager details, customer details and supplier details using the various interfaces available in the StoreAccess bean, which invokes methods on the remaining beans, from where the information is requested. RequestItems and DeliverItems are Message Driven Beans which listen for messages from a JMS producer and transfer messages to appropriate beans.

Now lets start this tutorial, with setting up the environment in chapter 1.

Last updated: Oct 22, 2003, Version 1.2

TOC Next

Tutorial for building J2EE Applications using JBOSS and ECLIPSE

Chapter 1

Configuration of ECLIPSE to use JBOSS and LOMBOZ

Install Eclipse.

First of all we have to set up Eclipse Integrated Development Environment (IDE) with JBOSS as our application server.

So go to this page: http://www.eclipse.org/downloads/index.php and download the binary for the Eclipse editor. You can download the appropriate binary for your platform. This tutorial was developed using Linux (Red Hat 8.0) as the operating system.

Example : eclipse-SDK-2.1-linux-gtk.zip, eclipse-SDK-2.1-win32.zip

Note : Eclipse does not include a Java runtime environment (JRE). You will need at least a 1.3 level Java runtime or Java development kit (JDK) installed on your machine in order to run Eclipse.

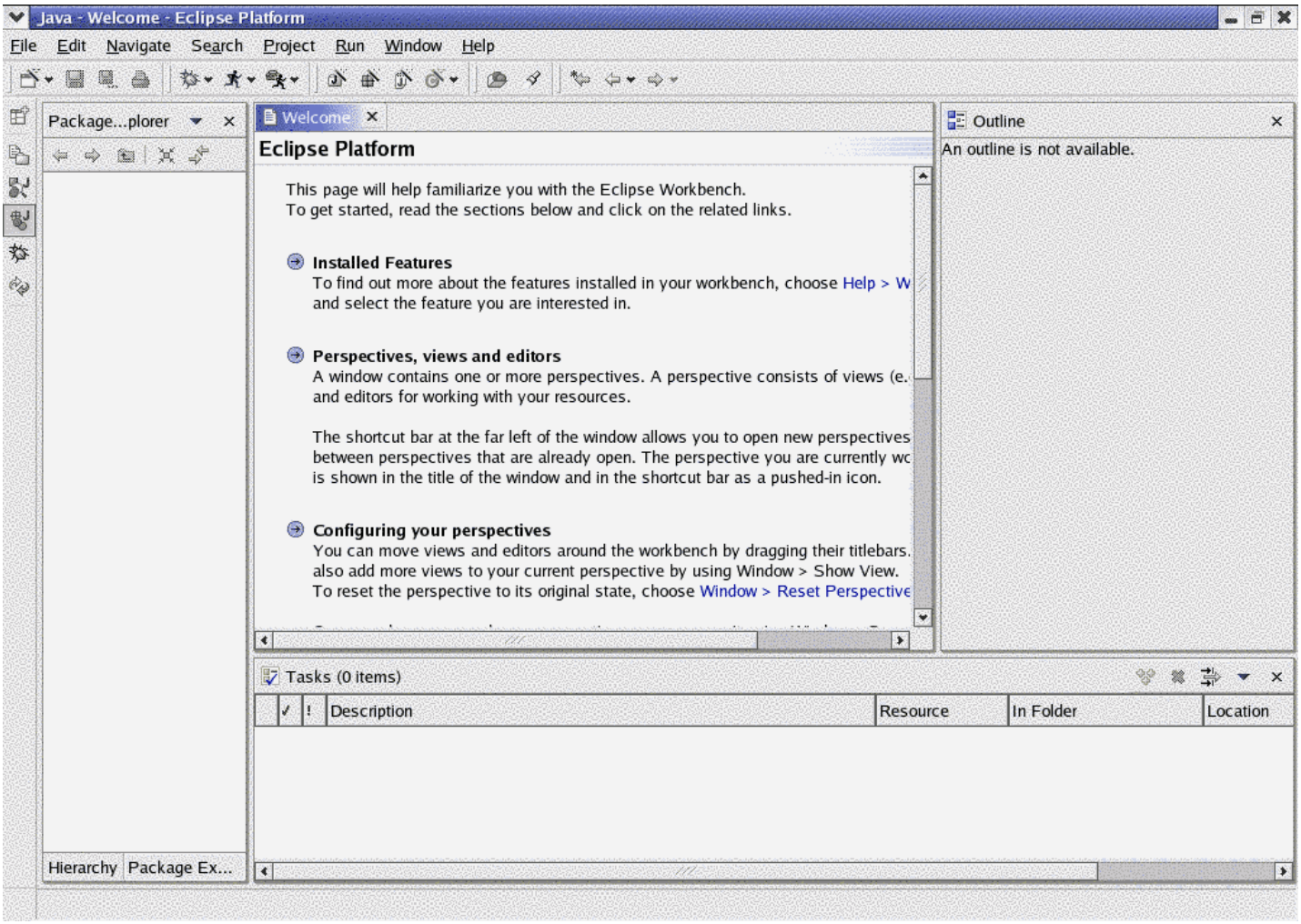
Now unzip this file into your home directory.

```
[vishal@localhost eclipse]$ unzip eclipse-SDK-2.1-linux-gtk.zip
Archive: eclipse-SDK-2.1-linux-gtk.zip
Archive: eclipse-SDK-2.1-linux-gtk.zip
inflating: eclipse/plugins/org.eclipse.core.boot_2.1.0/boot.jar
inflating: eclipse/plugins/org.eclipse.core.boot_2.1.0/splash.bmp
inflating: eclipse/plugins/org.eclipse.core.boot_2.1.0/boot.xml
inflating: eclipse/plugins/org.eclipse.core.boot_2.1.0/plugin.properties
  ------------------------------------------------------------------------
  ------------------------------------------------------------------------
  ------------------------------------------------------------------------
inflating: eclipse/install.ini
inflating: eclipse/startup.jar
inflating: eclipse/readme/readme_eclipse.html
```

Once you have unzipped this file you will have a directory called 'eclipse' in your home directory.
Go into that directory and run the script file called 'eclipse'.

```
[vishal@localhost eclipse]$ cd eclipse
[vishal@localhost eclipse]$ ls
cpl-v10.html features install.ini plugins startup.jar
eclipse icon.xpm notice.html readme workspace
[vishal@localhost eclipse]$ ./eclipse
```

This will run the Eclipse IDE on your workstation. Let's get familiar with this environment.



Install JBOSS.

Now Eclipse is installed and is running on your workstation, let's install JBOSS.

You can download JBOSS ready to run. Because JBOSS comes with a simple embedded Database (Hypersonic) and a web server (JBossWEB or Tomcat) you can use it out of the box without any initial configuration. Downloads can be accessed from either the JBOSS Project Page on SourceForge where you will always find up to date downloads (http://sourceforge.net/projects/jboss), or from the JBOSS Home Page where you will find useful information about JBoss, the official, free forums, project information etc., and also the download page http://www.jboss.org/downloads.jsp. For this tutorial we are using JBOSS-3.2.1 with the built-in 'Jetty' JBOSSWeb HTTP sever.

Example : **jboss-3.2.1.zip** *(includes JBossWeb HTTP server and JSP/Servlet engine, EJB, CMP2.0, JCA, IIOP, Clustering, JTS, JMX.)*

Note : JBOSS does not include a Java runtime environment (JRE).

First login as user 'root' and then unzip this file in a suitable place where you have enough space. I have unzipped it under an /opt/jboss/ directory.

It will create a directory named jboss-3.2.1 under /opt/jboss/.

Before you run JBOSS, in order to check your installation of JBOSS, make sure have you have following environment variables set.

```
JBOSS_HOME: /opt/jboss/jboss-3.2.1
JAVA_HOME: /usr/java/j2sdk1.4.1_02
CLASSPATH: /usr/java/j2sdk1.4.1_02/lib/tools.jar
```

Now go into directory jboss-3.2.1 and you will have all these directories.

```
[root@localhost jboss-3.2.1]# ls
bin client docs lib server
```

Go into the bin directory and you will find a script named run.sh.

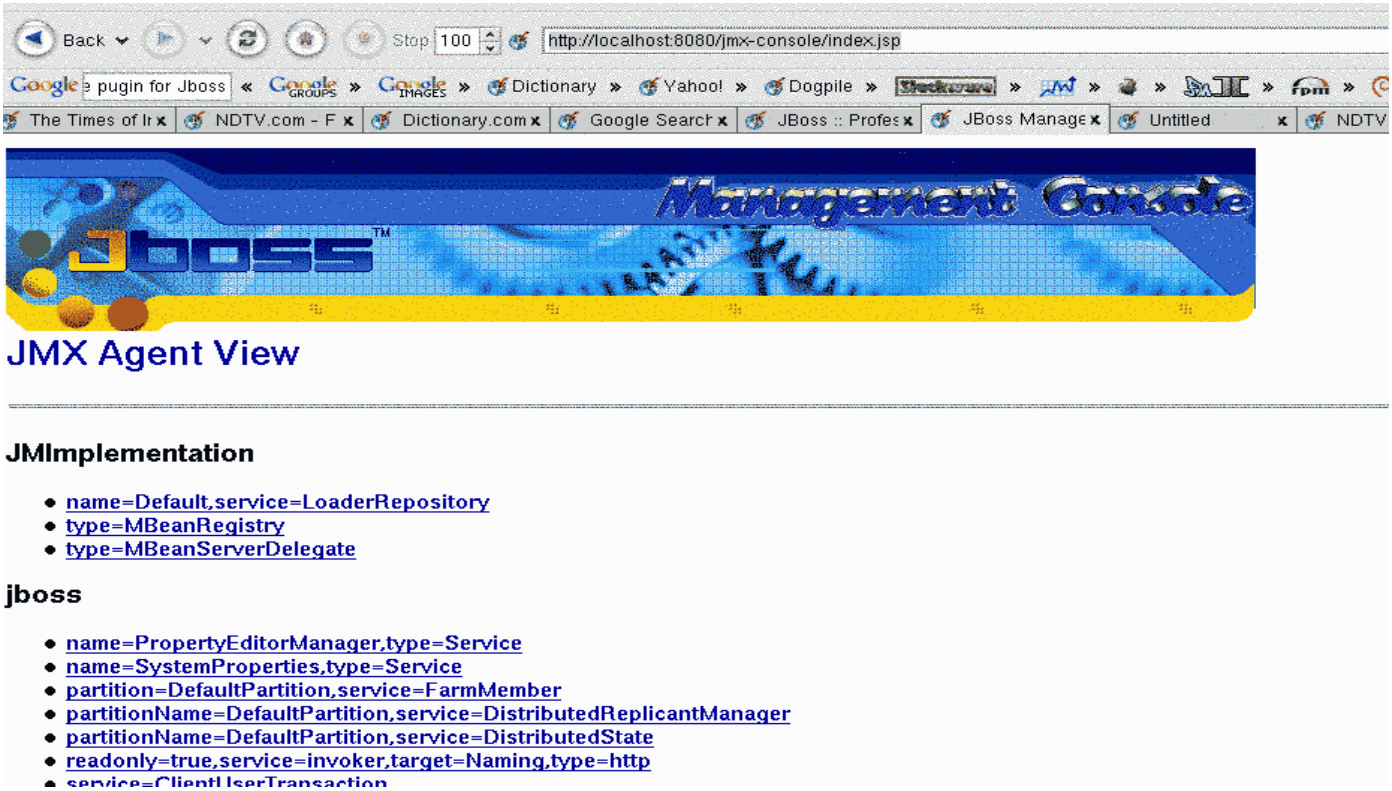
Run this script from the command line. Hopefully it will start, and you will see this message.

```
15:12:59,876 INFO [Server] JBoss (MX MicroKernel) [3.2.1 (build: CVSTag=JBoss_3_2_1 date=200305041533)] Started in 59s:676ms.
```

Note: This means that JBOSS is installed and running successfully. Now, it's time to configure it from within the eclipse IDE.

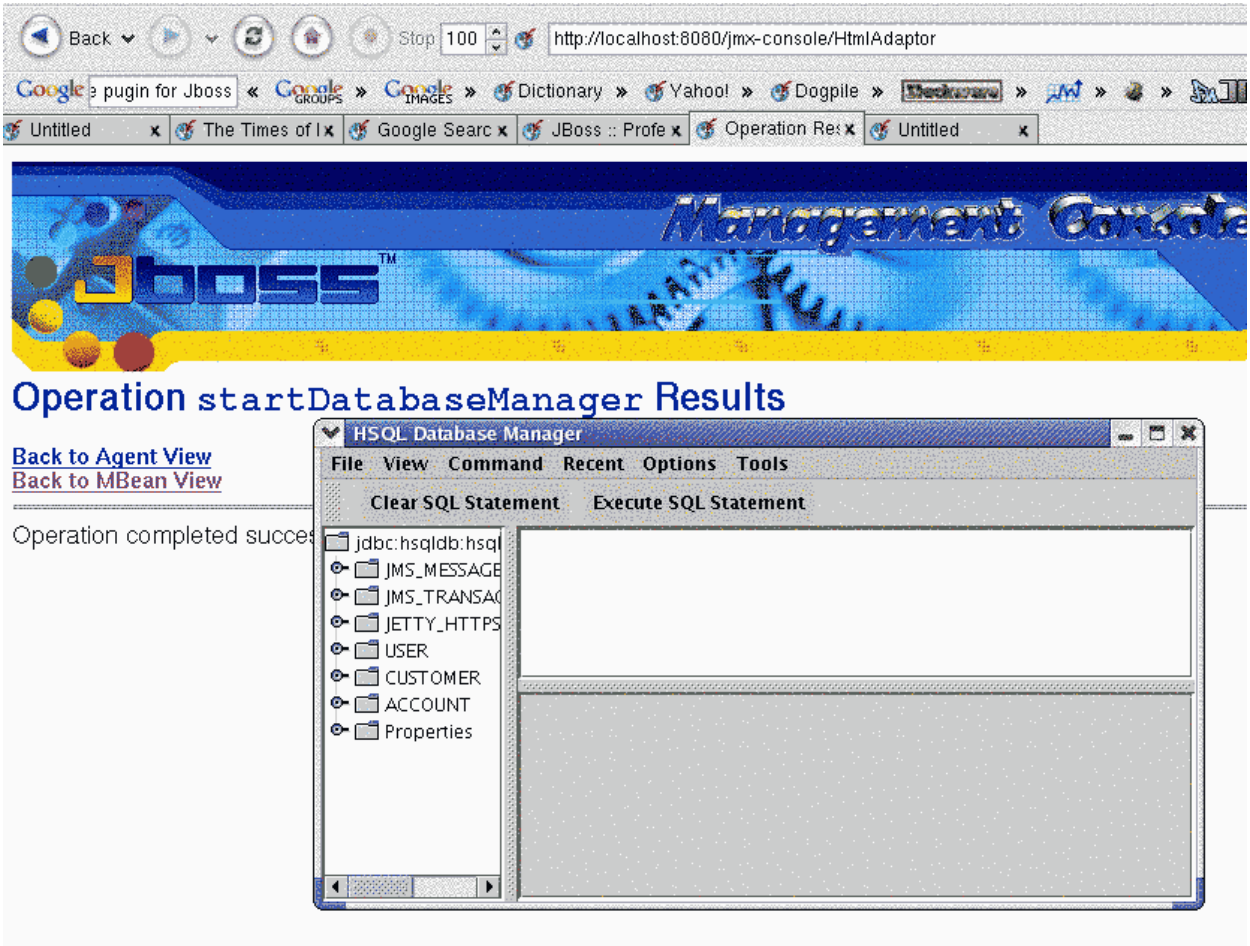
In order to test JBOSS is running correctly, go to your web browser and access this url 'http://localhost:8080/jmx-console/index.jsp'.

Note: This page is referred as the 'JMX Management Console'.



If you can access this screen from your browser, JBOSS is running successfully.

Note that on this page under the sub-heading of '**jboss**' there is a link '**service=Hypersonic**'. Go to that link; that will bring up an 'MBean View' page. On this page there is an MBean operation called 'startDatabaseManager'; go there and press the 'Invoke' button. This will bring up the console for the embedded Hypersonic database, from where you can access schemas and do other database management operations.



Creating the Database Schema.

As mentioned in the case study, the schema for our Store's Inventory is composed of five relations. We will load this schema in the Hypersonic embedded database.

In order to that, we will use the following script files.

a) myStoreSchema.script – to create the database schema.

b) myStoreSchema.data – to populate the tables with data.

c) myStoreSchemaDrop.script – to drop/delete all tables used in this tutorial.

Downloads :

myStoreSchema.script

myStoreSchema.data

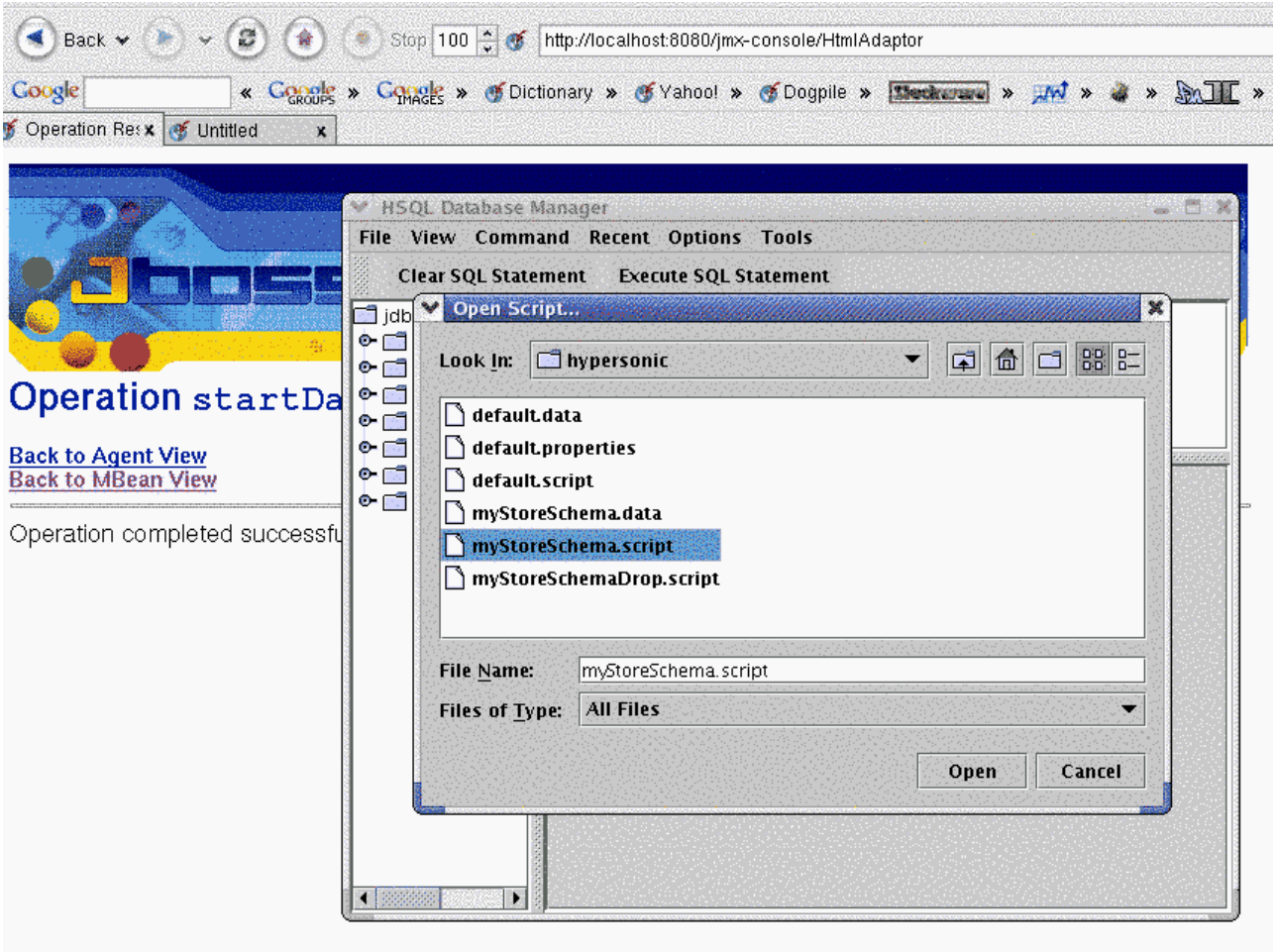
myStoreSchemaDrop.script

First of all copy these 3 files into the $JBOSS_HOME/server/default/data/hypersonic/ directory. In this example it is /opt/jboss/jboss-3.2.1/server/default/data/hypersonic/

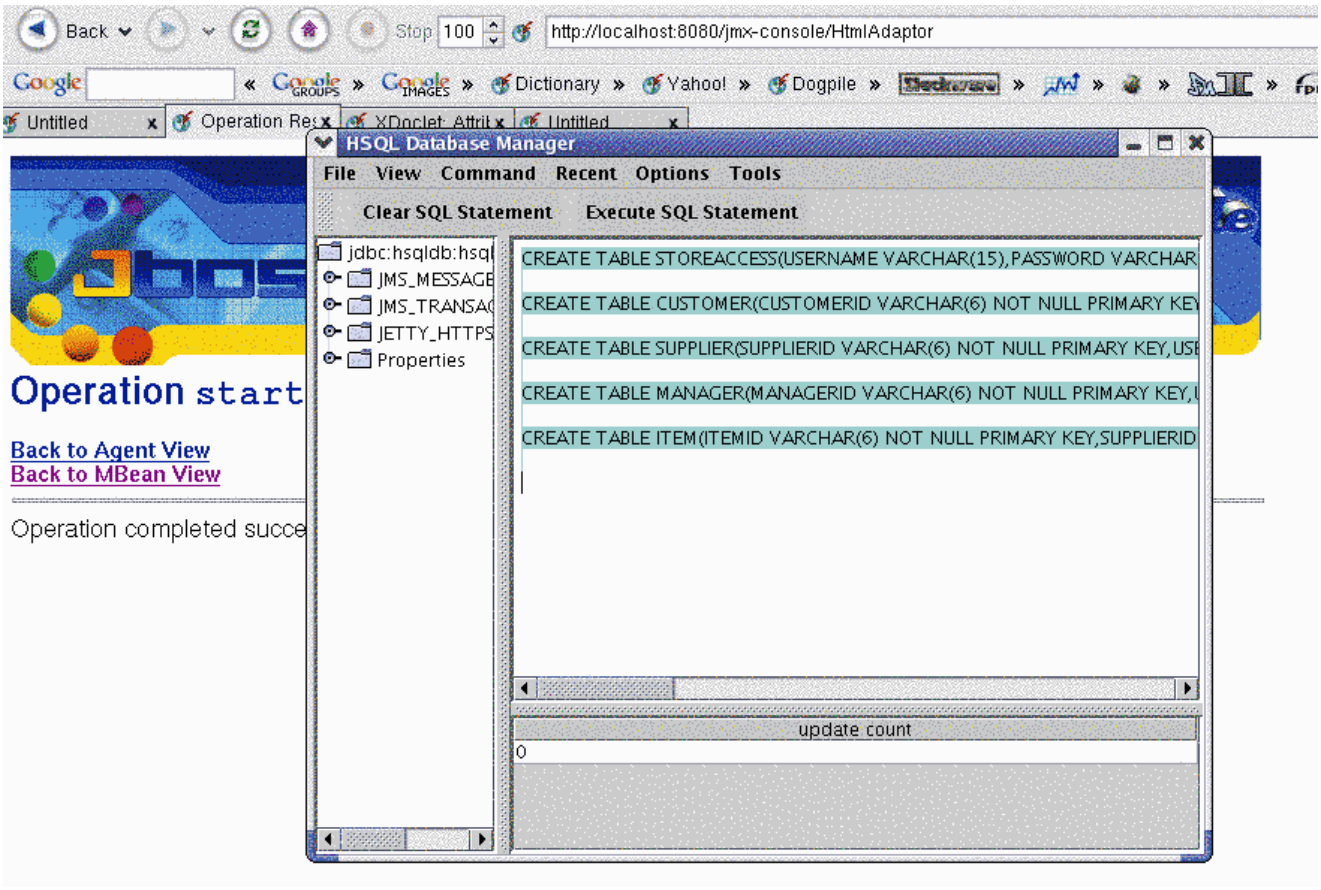
Now access the HSQL Database Manager using the JMX-Management Console shown above.

Note: Please drop/delete any tables already present in the database with same name beforehand, or naturally errors will be raised when you will run these scripts.

On the top level menu of HSQL Database Manager File > Open Script.. > select myStoreSchema.script.

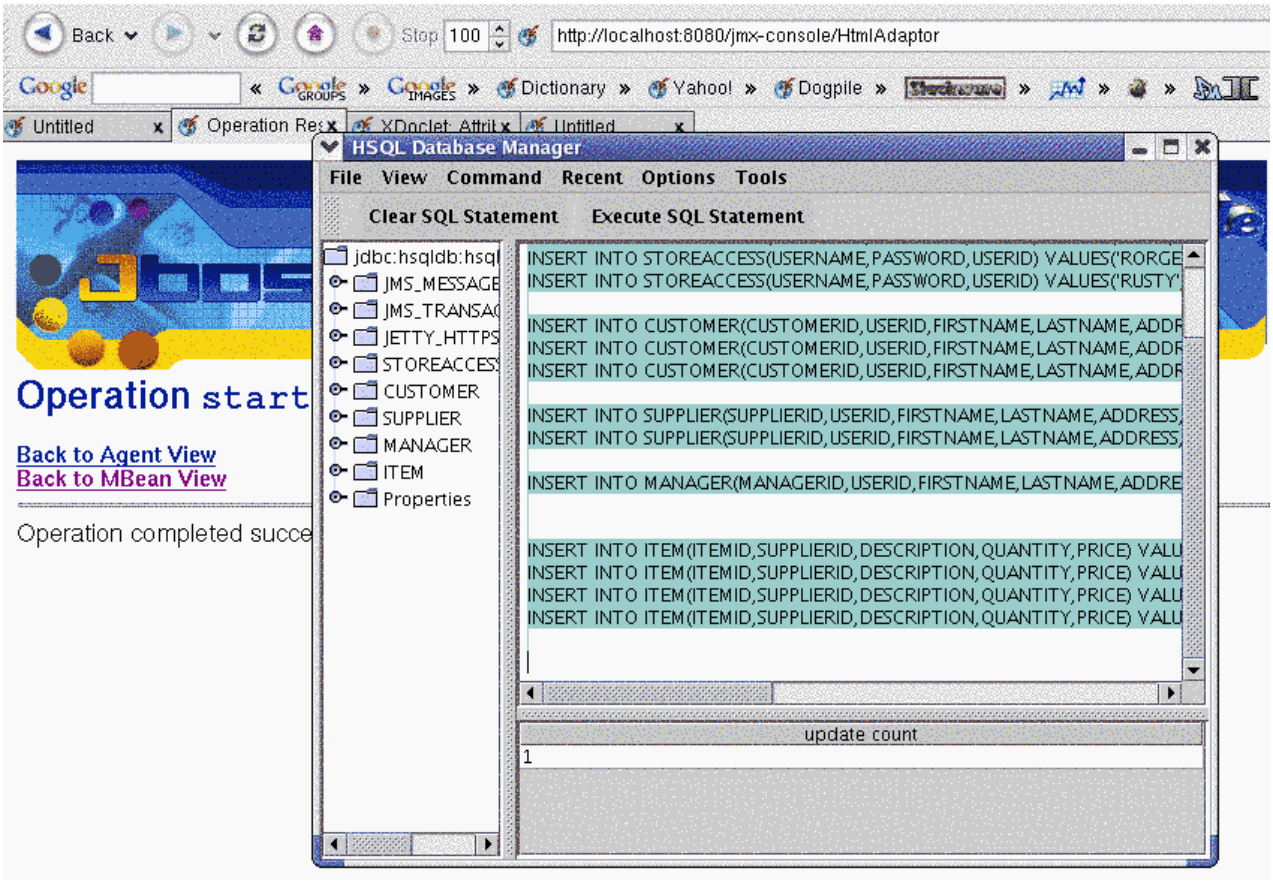


After clicking Open > press Execute SQL Statement and the following screen will appear as shown in the figure below.

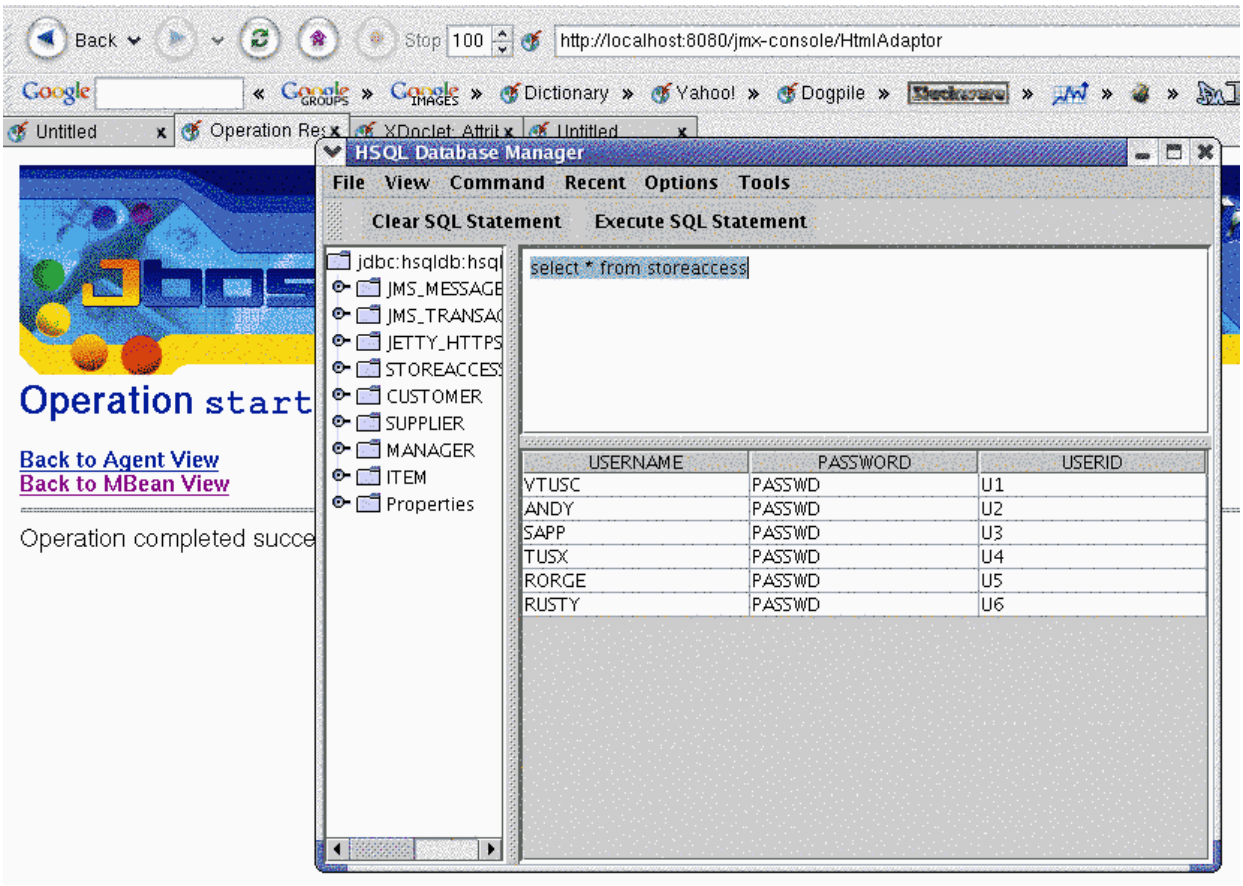


Go to View > Refresh Tree. If you can see five new tables created then your schema is ready, and it's time to populate them with data.

Go to File > Open Script.. > select myStoreSchema.data > Open.. > Execute SQL Statement.



Now your data is loaded in the database, in order to verify just execute any query and see the results of that as shown in figure below.



Make sure you 'commit' (Options->commit) to save your schema in the database.

Note : Shutdown Jboss server as we will run it later from inside eclipse.

Install Lomboz.

In order to run JBOSS from within Eclipse you need a plug-in, and for this tutorial we are using Lomboz which is available from http://sourceforge.net/projects/lomboz or from http://www.objectlearn.com/index.jsp.

Note : We are using lomboz.21_02.zip for this tutorial.

Now unzip this file in a temporary directory; you will have a directory named plugins. Go to the directory plugins/com.objectlearn.jdt.j2ee/servers. This directory contains configuration files for various servers. For this tutorial we supply a configuration file for jboss3.2.1 available under downloads.

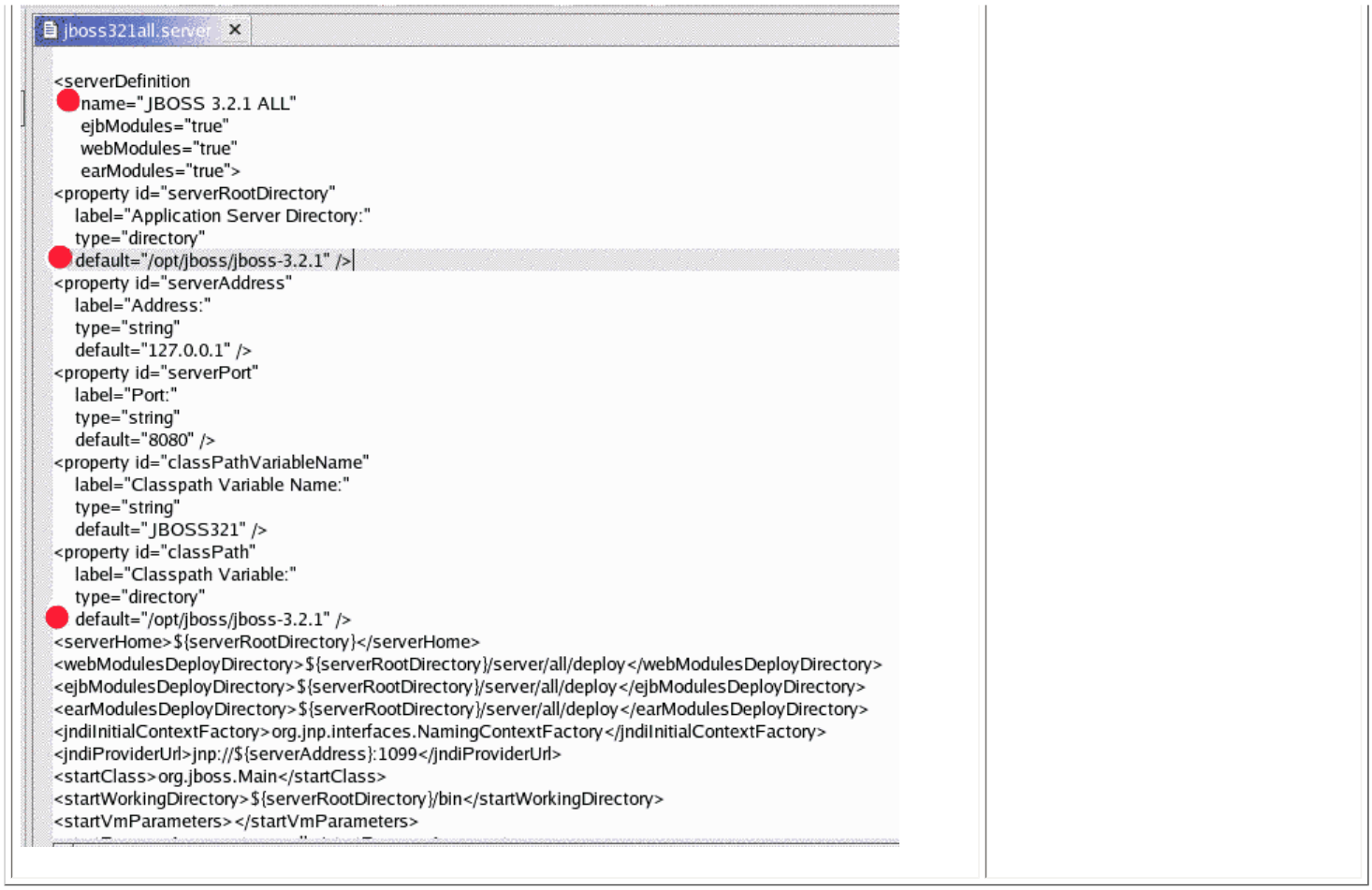
Note : If there are multiple configuration files the plugin will presently default to the first configuration file within the (eclipse/plugins/com.objectlearn.jdt.j2ee/servers/) directory in alphabetical order, so rename all the configuration files to a '.bak'
extension. This is just to avoid slip-ups whilst we're learning.

Now, copy the supplied jboss321all.server file into the servers directory.

```
cd plugins/
[vishal@localhost plugins]$ ls
com.objectlearn.jdt.j2ee
[vishal@localhost com.objectlearn.jdt.j2ee]$ cd servers
[vishal@localhost servers$ ls
jboss244.server.bak jboss303Tomcat4112.server.bak jboss300all.server.bak jboss321all.server
tomcat403.server.bak weblogic70.server
jboss300.server.bak tomcat410.server x.log
```

Here's a snippet from the file '**jboss321all.server**' used for configuration. **For this tutorial we will use this file, which you can download under Downloads.**

	Downloads : *jboss321all.server*



You can change the various settings in this file according to your environment (e.g. the port number).

Once you have saved this file under the com.objectlearn.jdt.j2ee/servers directory, move the com.objectlearn.jdt.j2ee directory under $HOME/eclipse/plugins/ directory, where $HOME is your home directory.

```
[vishal@localhost temp/plugins] mv com.objectlearn.jdt.j2ee /home/vishal/eclipse/plugins/.
```

Now, we will configure Lomboz using this **jboss321all.server** file.

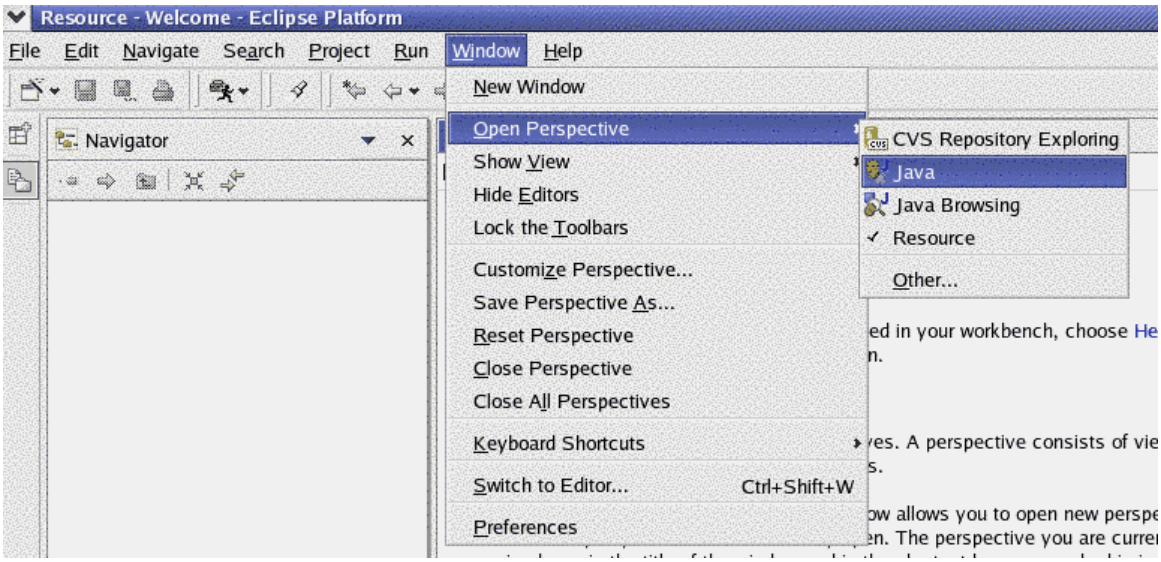
Note : Now you can now delete the temporary directory.

Lomboz Configuration.

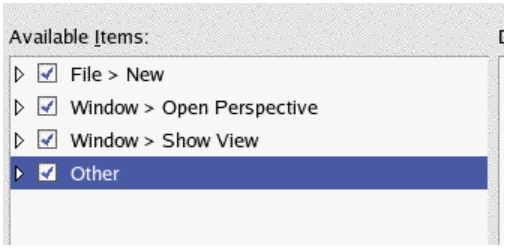
Now run your Eclipse IDE to configure Lomboz ($HOME/eclipse/eclipse).

Go to Window on top level menu > Open Perspective > Java as shown in fig below.

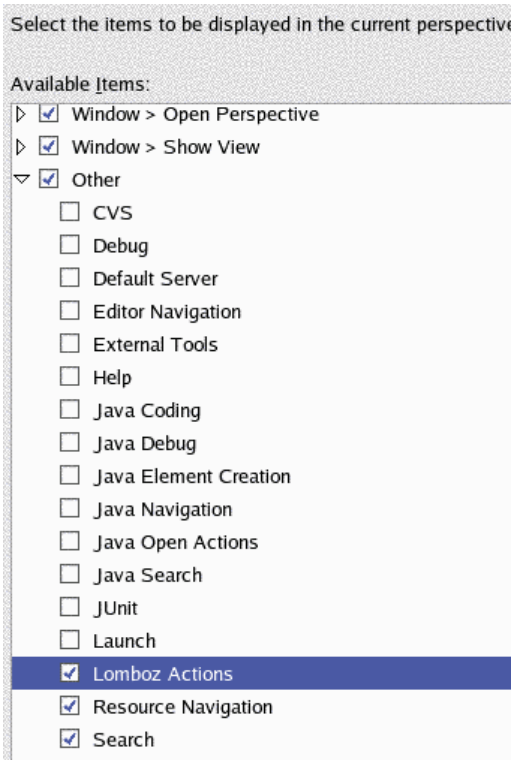
As we want to customize Java perspective for this tutorial:



Go to Window again at top level menu > Customize Perspective > Other.



Select Lomboz Actions sub node.



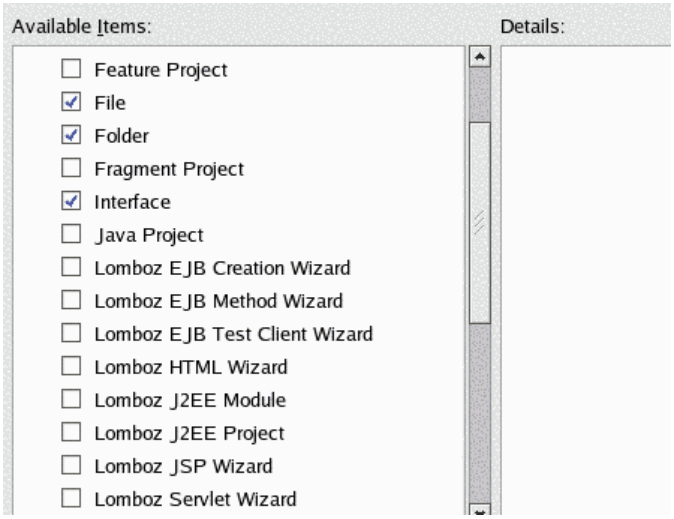
Click OK. Now you will have an icon under your top level menu like this.

Click on this icon will have a Lomboz J2EE View window in your workbench.

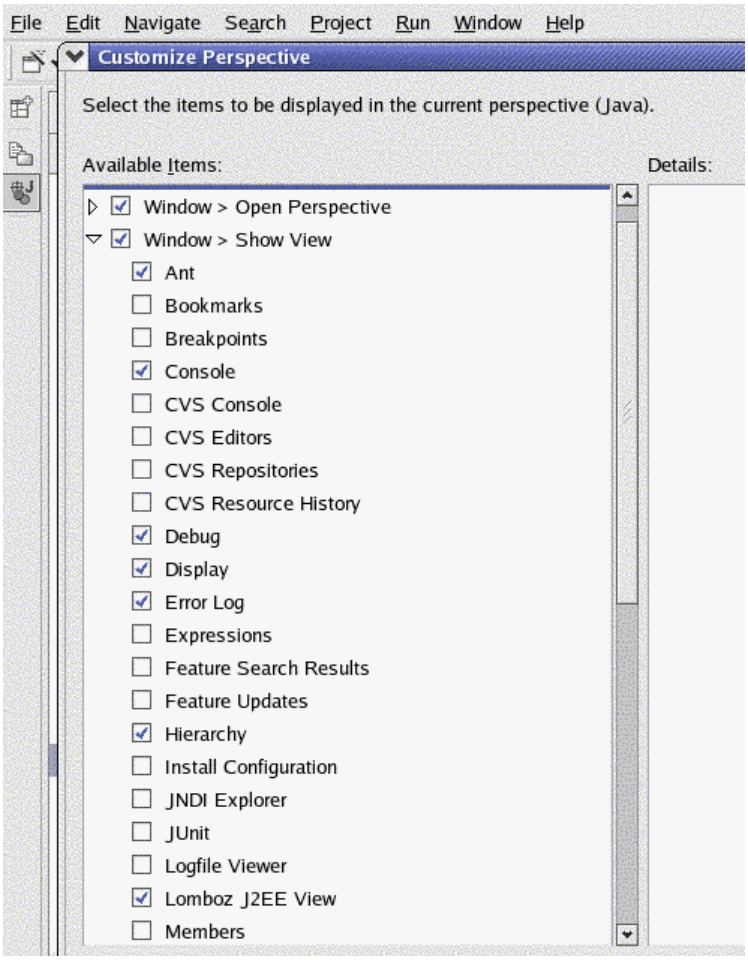
Now, Lomboz J2EE view is available.

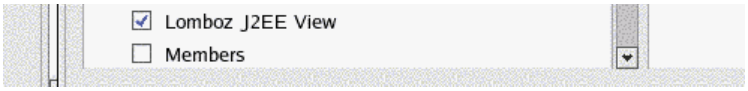
Go to Window > Customize Perspective.

Select File > New sub node. Select all nodes starting with Lomboz.

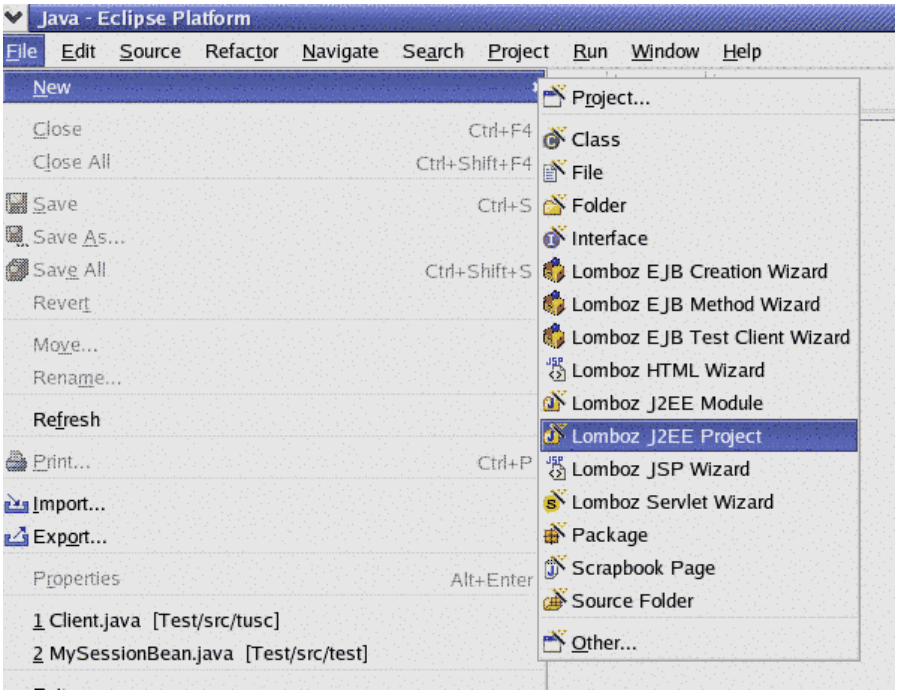


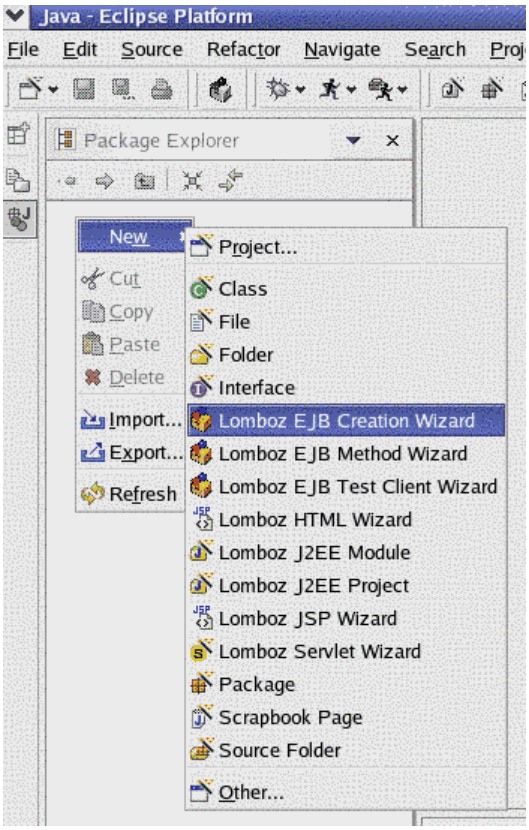
Select Window > Show view. Select Console, Lomboz J2EE View and others as shown below in figure.





Now click ok. You will have all those options available from File > New > and Lomboz menu will appear, or by right clicking in the Package Explorer a pop up menu will appear.





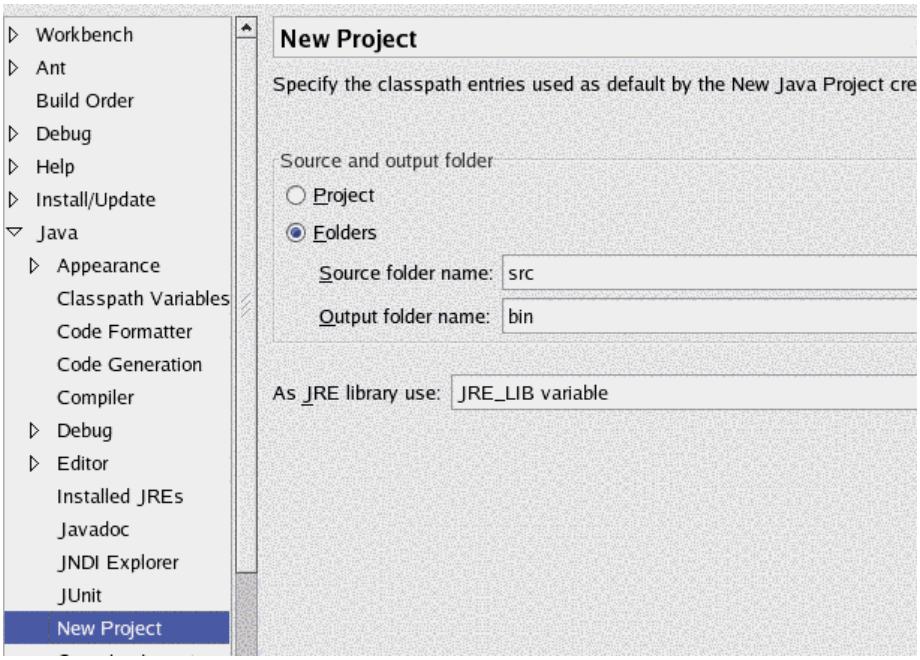
Configure JBOSS to run from within Eclipse.

First of all we have to configure the Java Development settings, because Lomboz requires different directories for source and binaries.

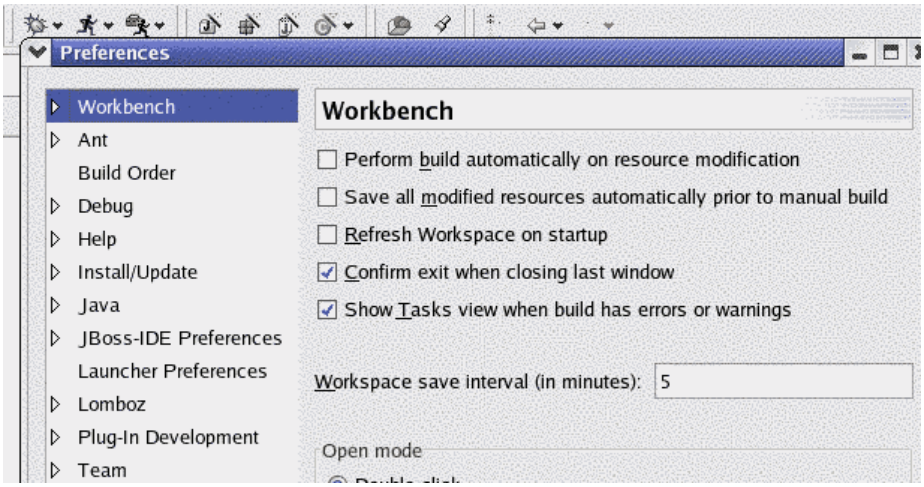
Go to Window > Preferences > Java > New Project.

Enter src and bin for the names of these folders, which are the defaults. Please do not modify these names as they are required by some of the Lomboz tasks.

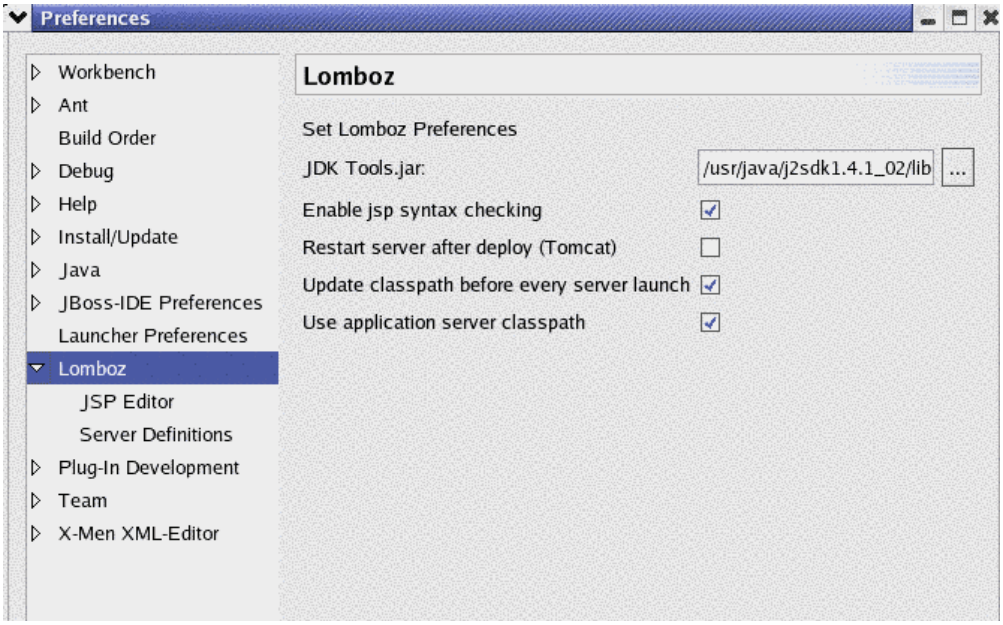
Also, make sure JRE library is set to 'JRE_LIB' variable.



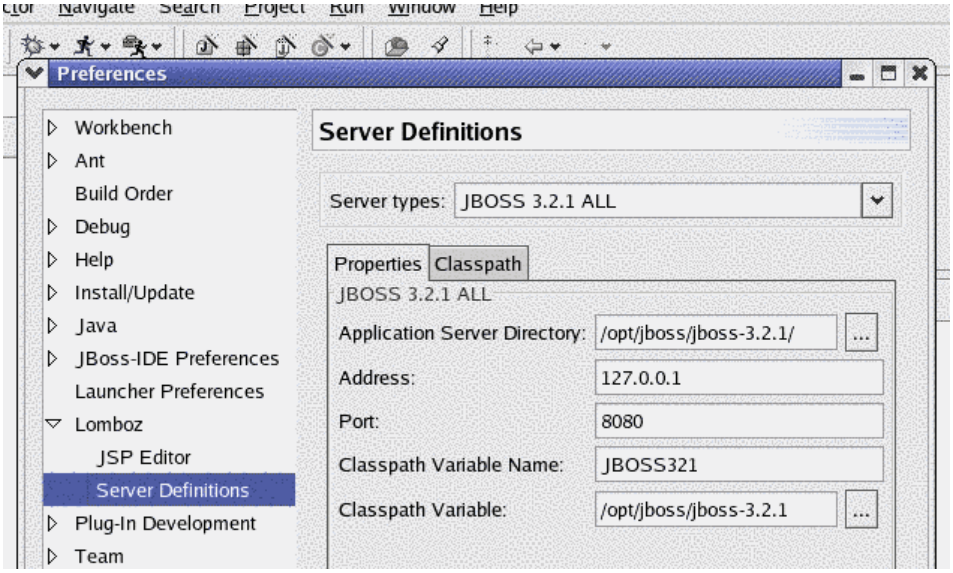
Go to Window on top level menu > Preferences.



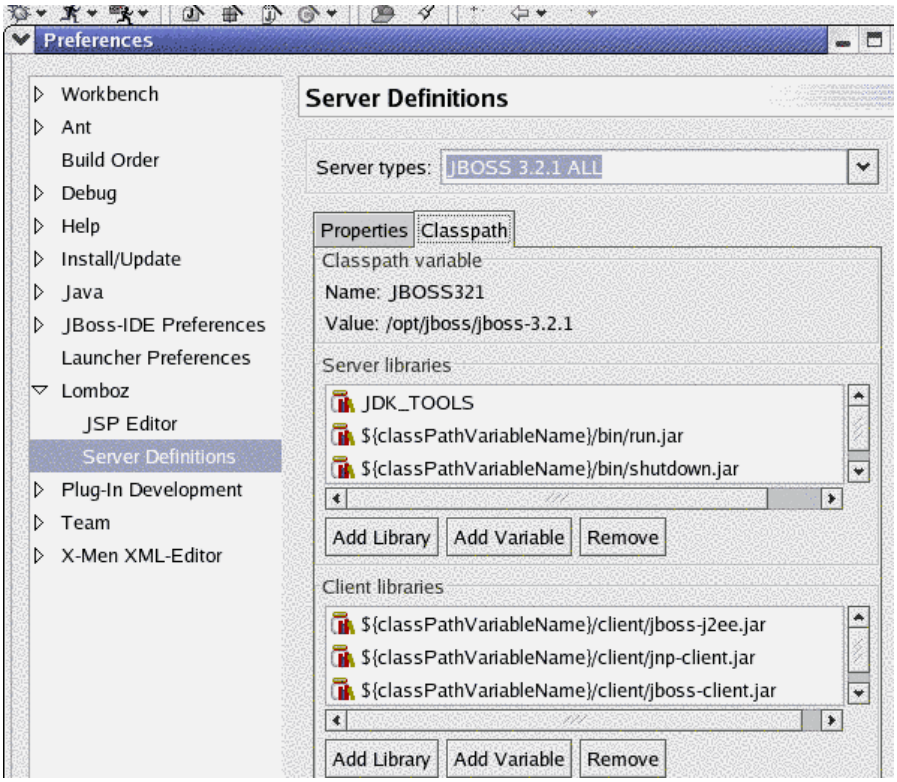
Go to the Lomboz node under the Preferences window. Make sure JDK Tools.jar is set to $JAVA_HOME/lib/tools.jar



Go to the server definitions sub node under Lomboz node. Select JBOSS 3.2.ALL under Server types and the rest of the options will be filled after loading the server configuration file which we made for this tutorial.



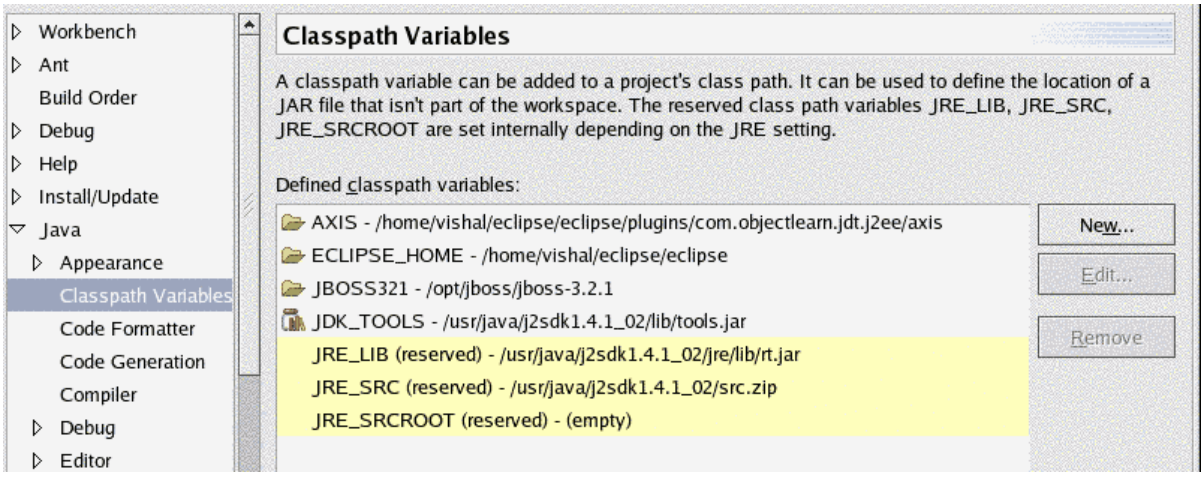
Now, under Server Definitions, go to 'Classpath' and make sure all the paths are correct. If there is a wrong setting for a path, it will show red crosses with jar icons in Server libraries / Client libraries sub section.



You can add or remove libraries from here. Make sure you 'Apply' the changes after altering the various options.

And now verify your 'Classpath Variables'.

Go to Window > Preferences > Java > Classpath Variables

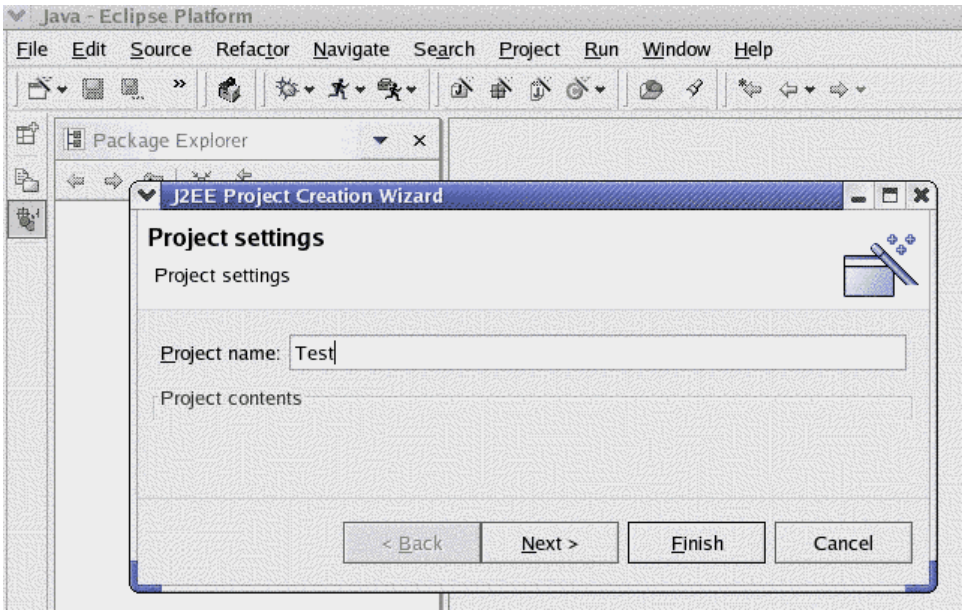


If you get all these Classpath variables right according to your environment.

Test your configuration.

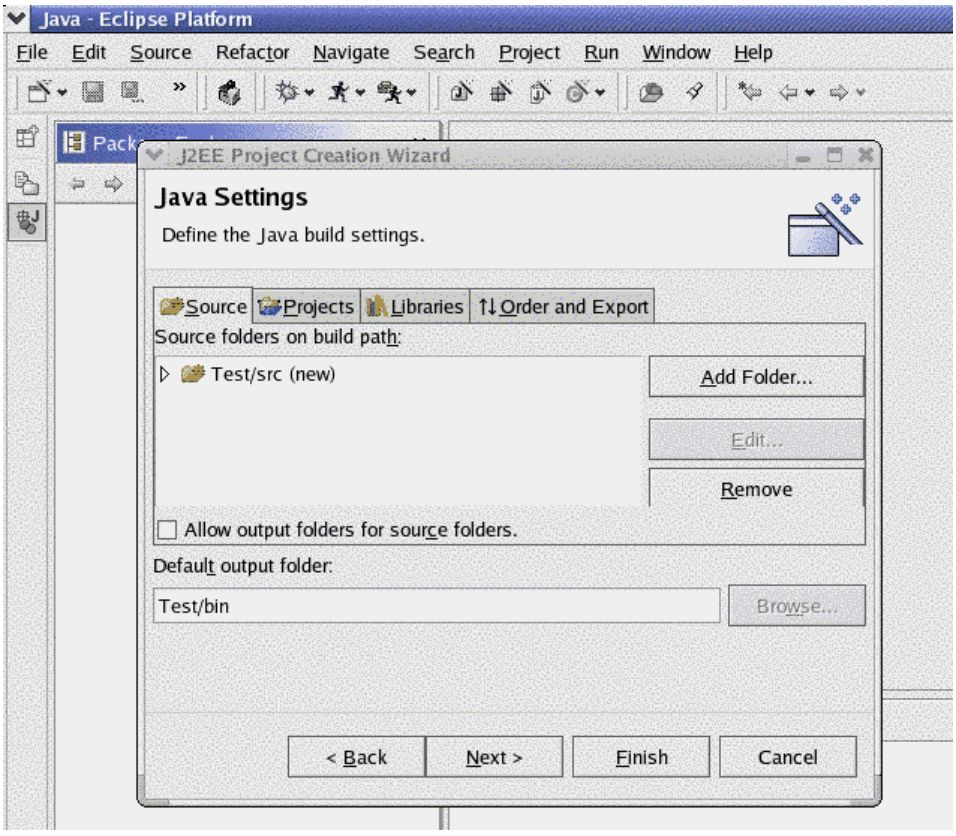
To test your configuration, create a J2EE Project as a test.

Go to File at top level menu > New > Lomboz J2EE Project.



Enter 'Test' as project name and press Next.

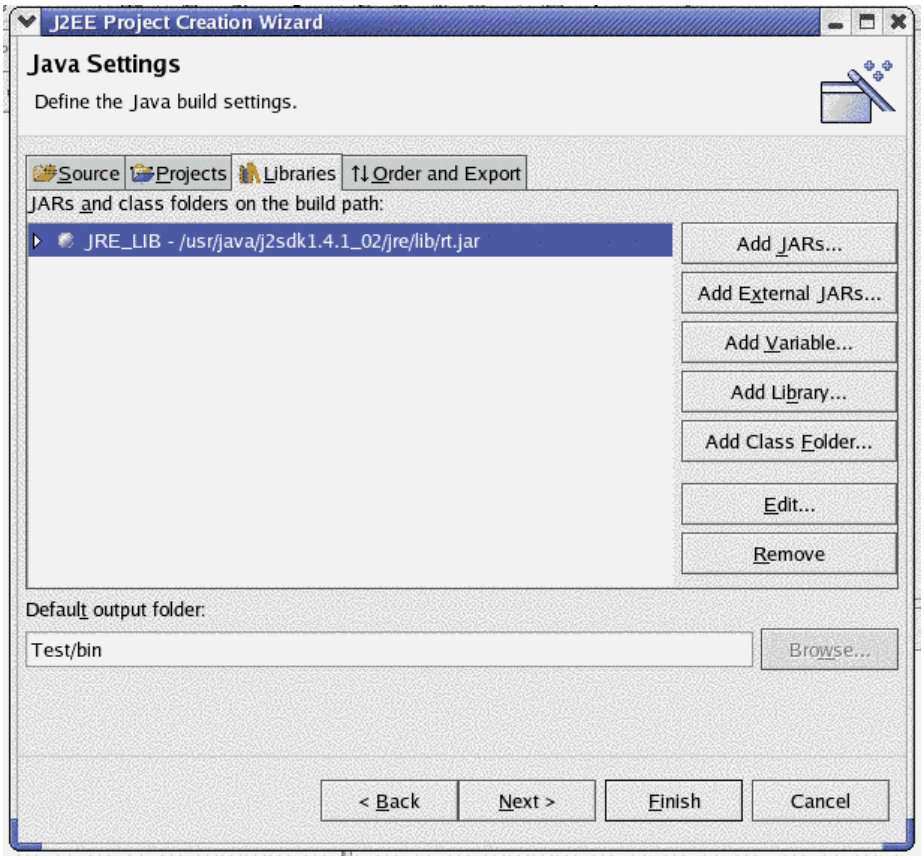
Check your settings here; if you have 'Test/src' on build path and 'Test/bin' under default output folders then it's ok.



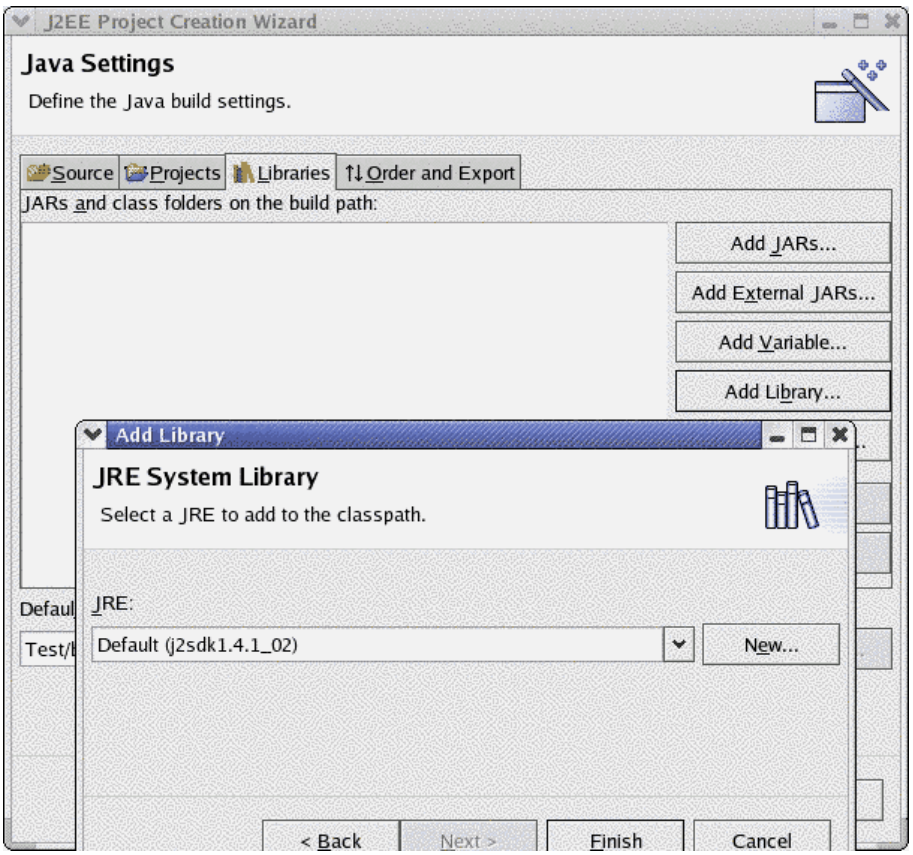
Now click on Libraries tab on the Project Creation Wizard.

Because of a bug in Eclipse 2.1 this library is configured wrongly.

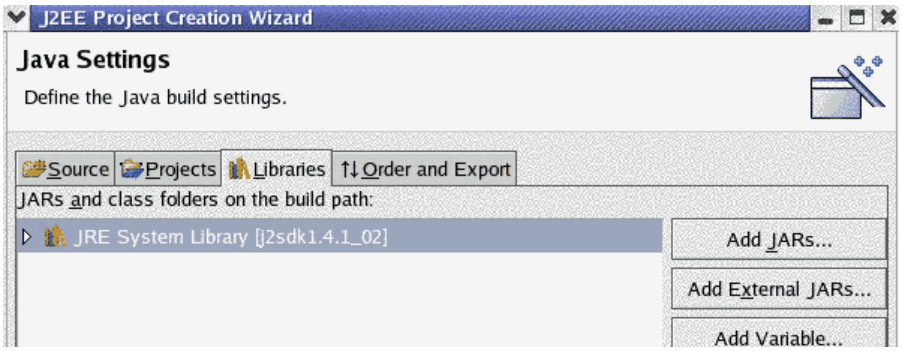
So select this library and Remove it.



Now, add a new library > Add Library.. > Select JRE System library > Next .



Select Default library and press Finish.

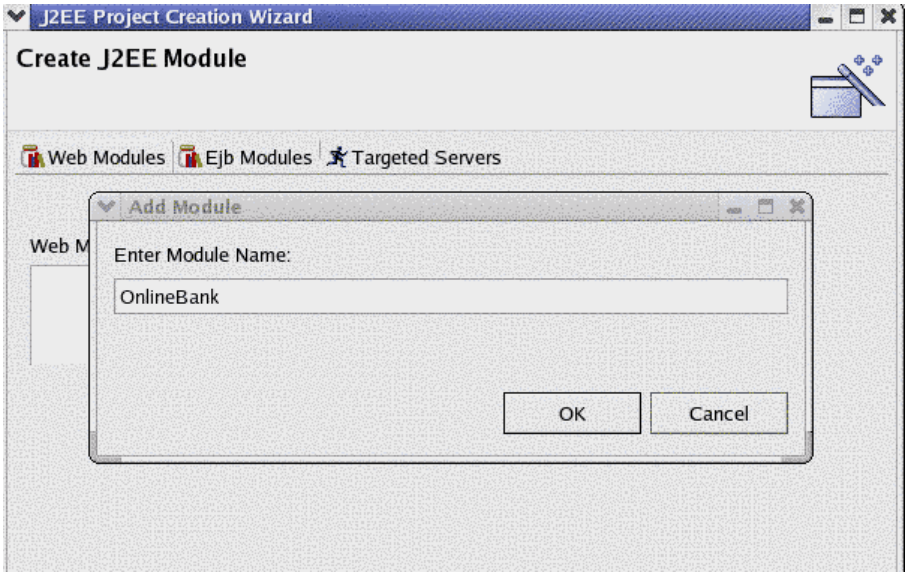


New library is configured.

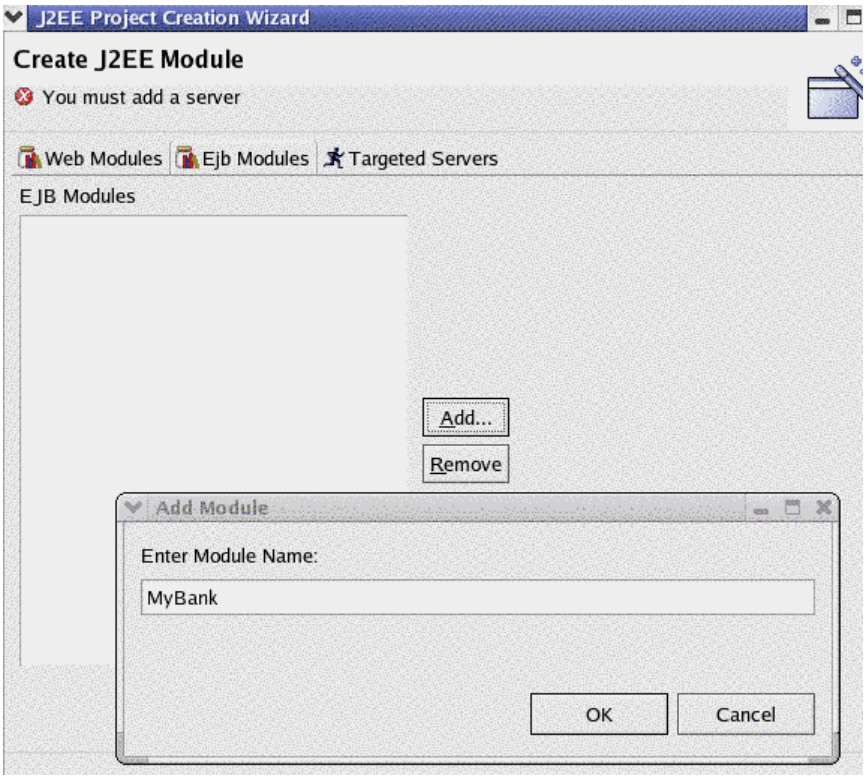
Note: You have to repeat this step whenever you are creating a new J2EE Project, as there is a bug in eclipse 2.1.

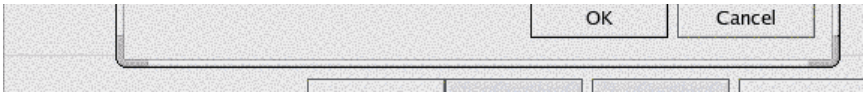
Now press Next.

Go to Web Module > Add.. > Enter Module Name 'OnlineBank' > Ok.



Go to Ejb Modules tab > Add.. > Enter Module Name 'MyBank' > Ok.

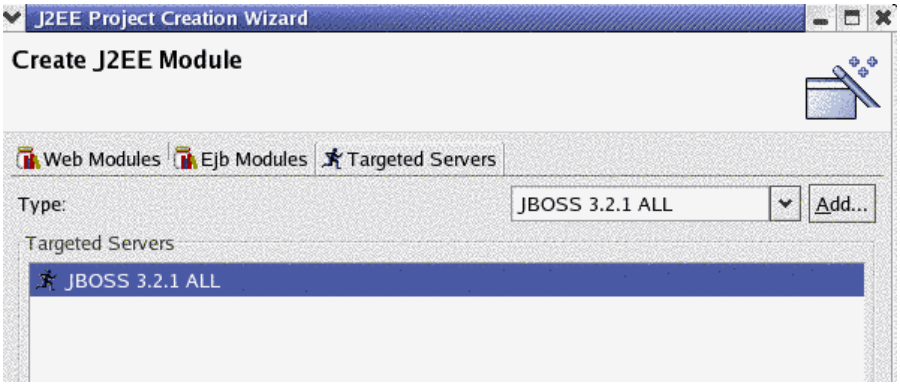




Go to Targeted Server > Add.. > 'JBOSS 3.2.1 ALL' .

Note: 'JBOSS 3.2.1 ALL' is not the file name, but the name assigned to server in the 'jboss321all.server' file used for configuration showed above. Snippet from file below.

```
<serverDefinition
name="JBOSS 3.2.1 ALL"
ejbModules="true"
```



Press Finish.

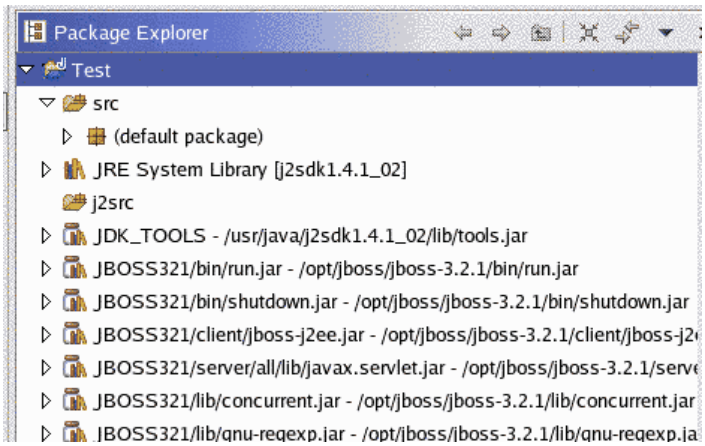
Now you will have all these directories and libraries under Test Project in the Package Explorer.

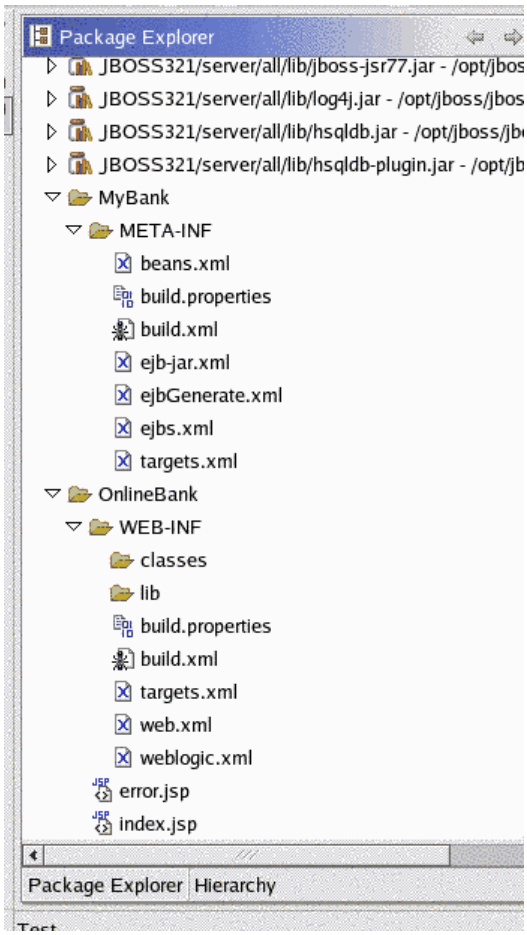
Under Project Test 'src' is the directory where all packages for EJB components and Servlets will be developed, that is your java source files.

The 'MyBank' directory is your EJB Module and will have 7 files at start shown in fig below.

The 'OnlineBank' directory is your Web Module and will have 7 files shown in fig below. Your JSP pages will come under this directory for this module.

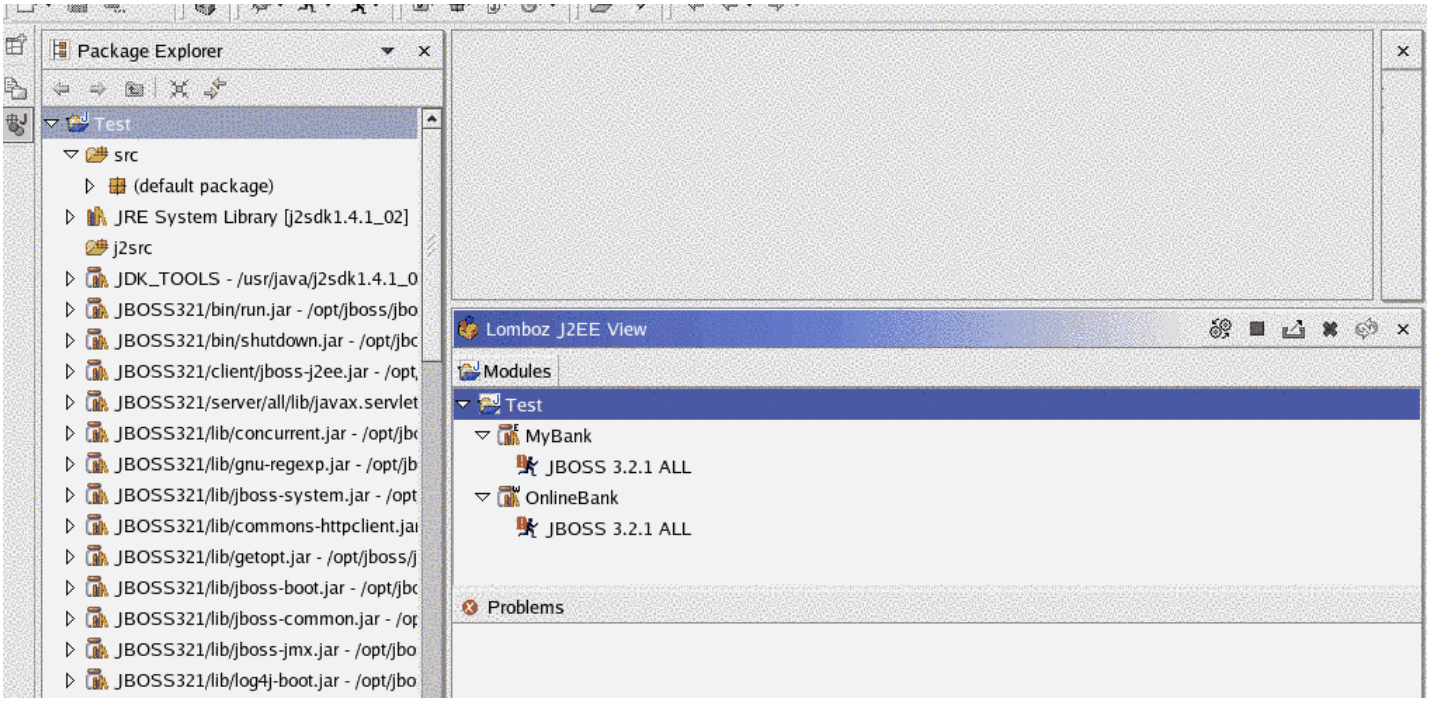
Note: If you are interested in Ant (from the Jakarta Project) then note that both modules have an Ant 'build.xml' file which is used to build the application.





Now go to top level menu option Window > Show View > Lomboz J2EE View.

Lomboz J2EE view will have Test Project , which has two modules 'MyBank' and 'OnlineBank' using JBOSS 3.2.1 as their default server.

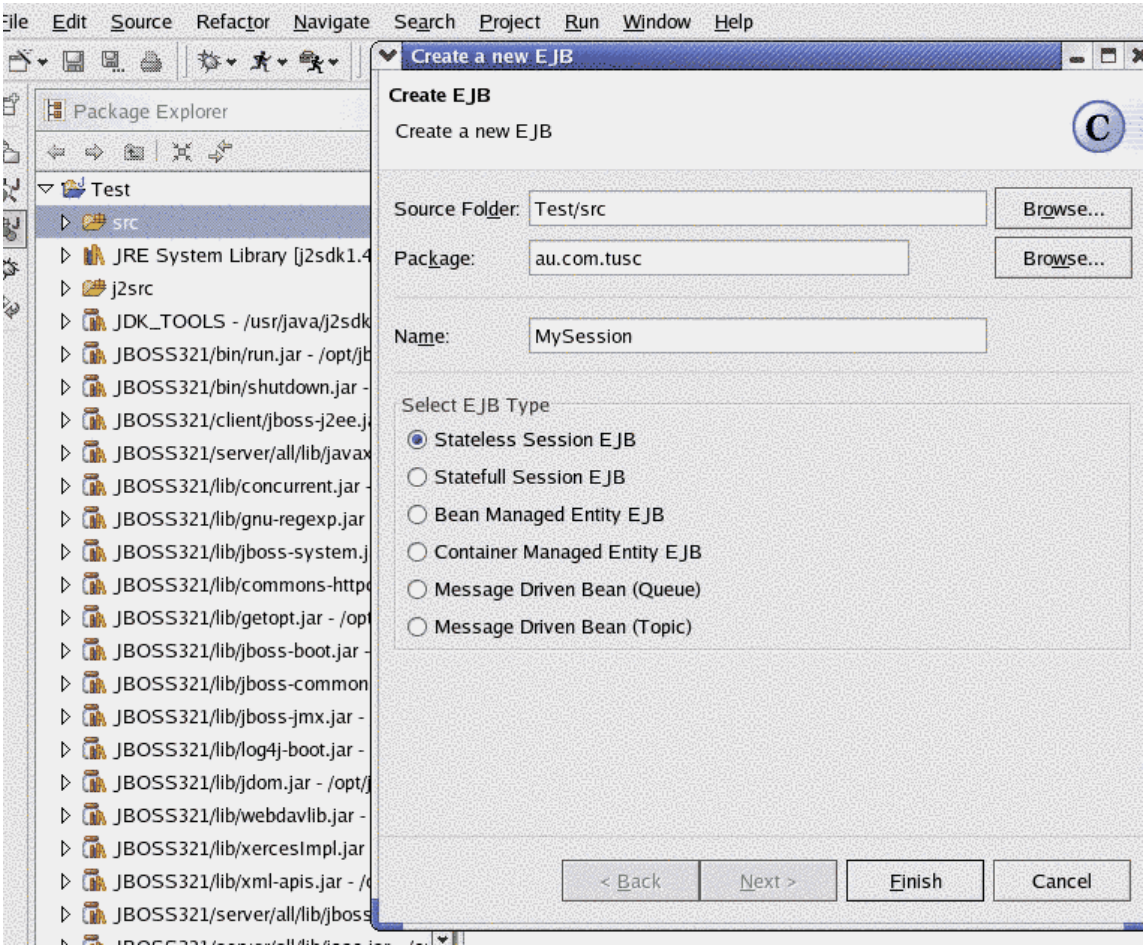


Now it's time to create an EJB,

Go to File > New > Lomboz EJB creation wizard.

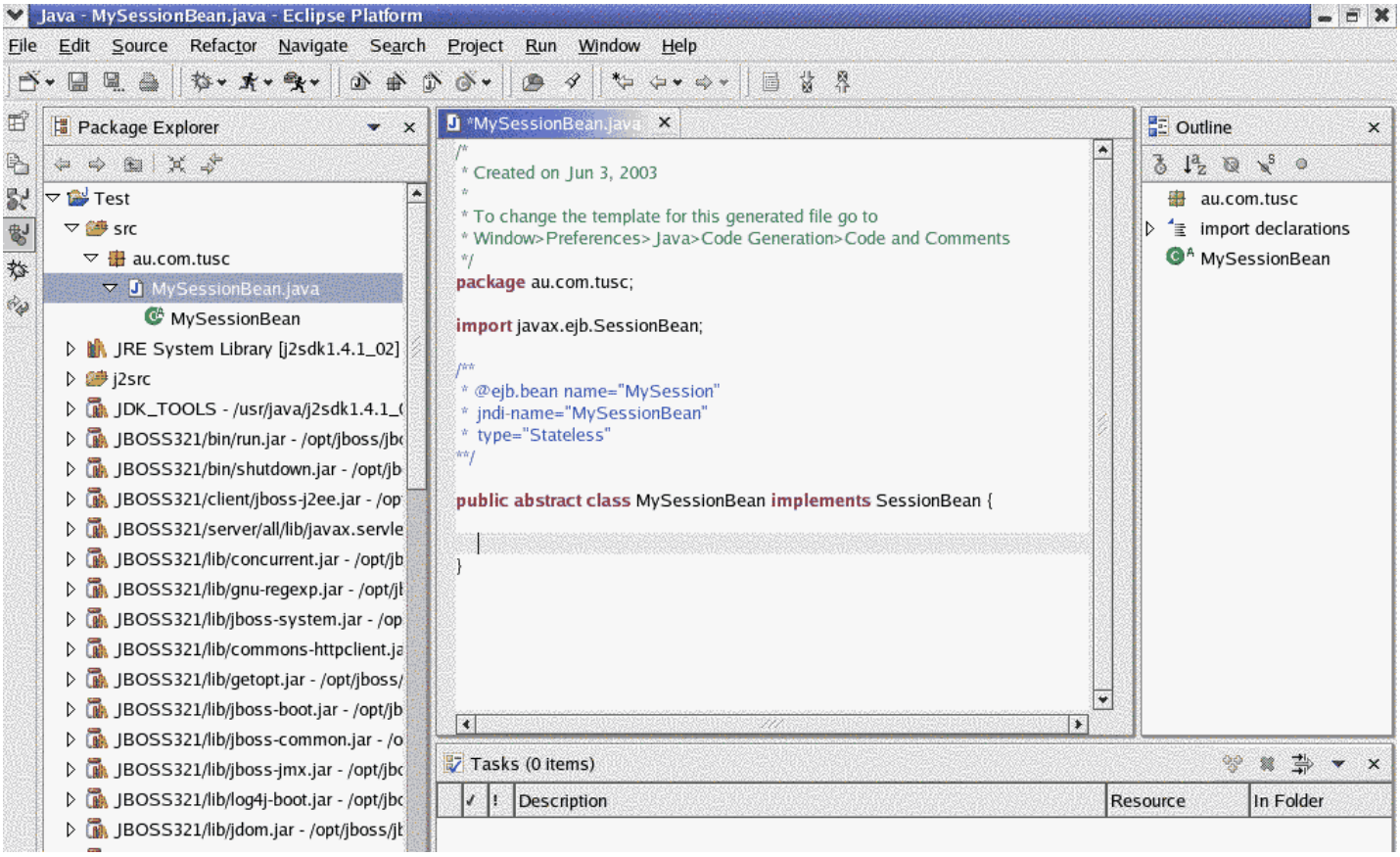
Note: You can access this wizard by right clicking in on project Test in Package Explorer. Go to New > Lomboz EJB creation wizard as explained in the beginning.

Add package name 'au.com.tusc'; you can choose another name if you want. Add 'MySession' in Name. Select Stateless Session EJB option.



This will create a file 'MySessionBean.java' under the 'au.com.tusc' package as shown in figure below.

Note: It will generate the bean name, jndi-name and type of bean in the file. Also the name of the file is appended with word 'Bean' as you gave the name of the bean as 'MySession' only. So be careful with naming conventions, as you only need to specify the bean name in the wizard. Don't append the word Bean to the name as the wizard will do that for you.

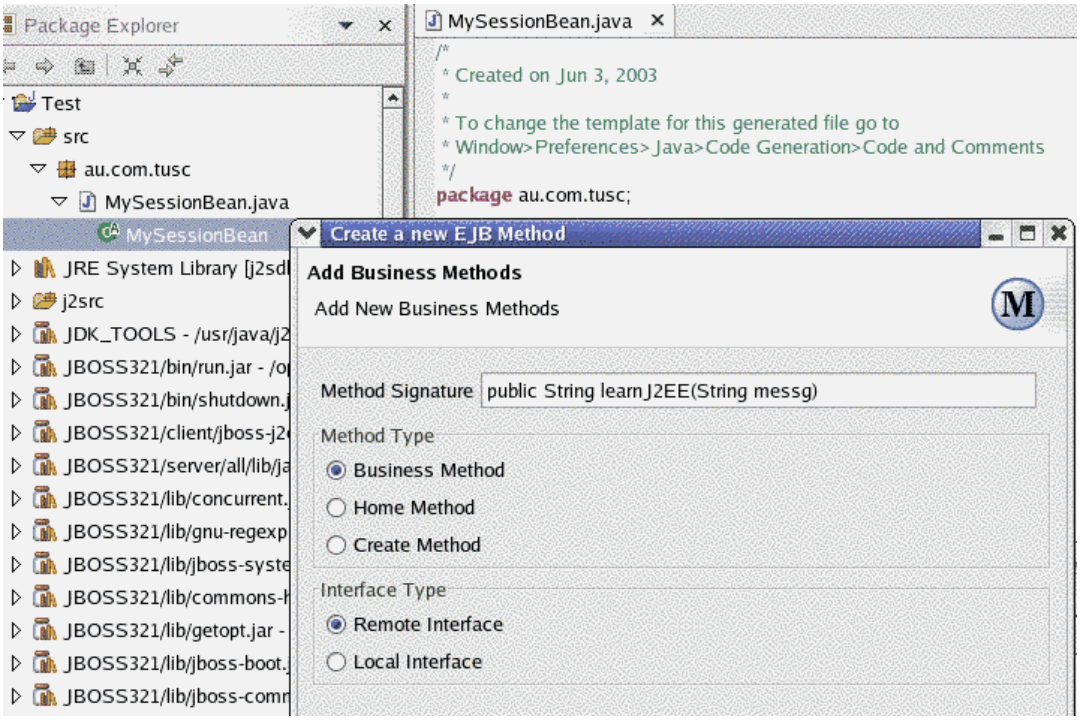


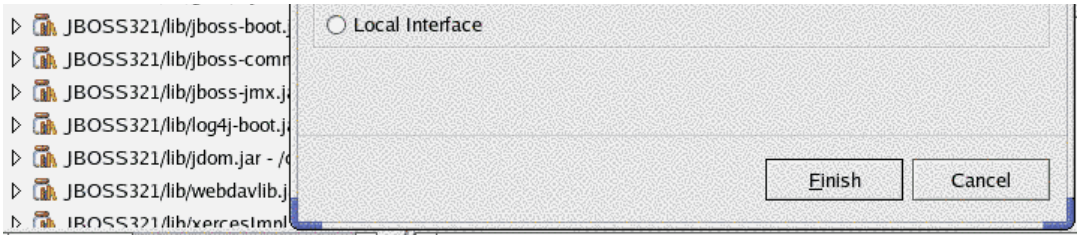
Now we will add a business method via the wizard.

Go to file 'MySessionBean.java' under 'Package Explorer' and expand that, right click on 'MySessionBean' node as shown in the figure below, select New > Lomboz EJB Method wizard.

In the method signature put 'public String learnJ2EE(String messg)', select method type as 'Business Method' and

Interface Type as 'Remote Interface'.





This will generate the required signature for your business method in your bean class.

Now add this line in this method "Me too! ";

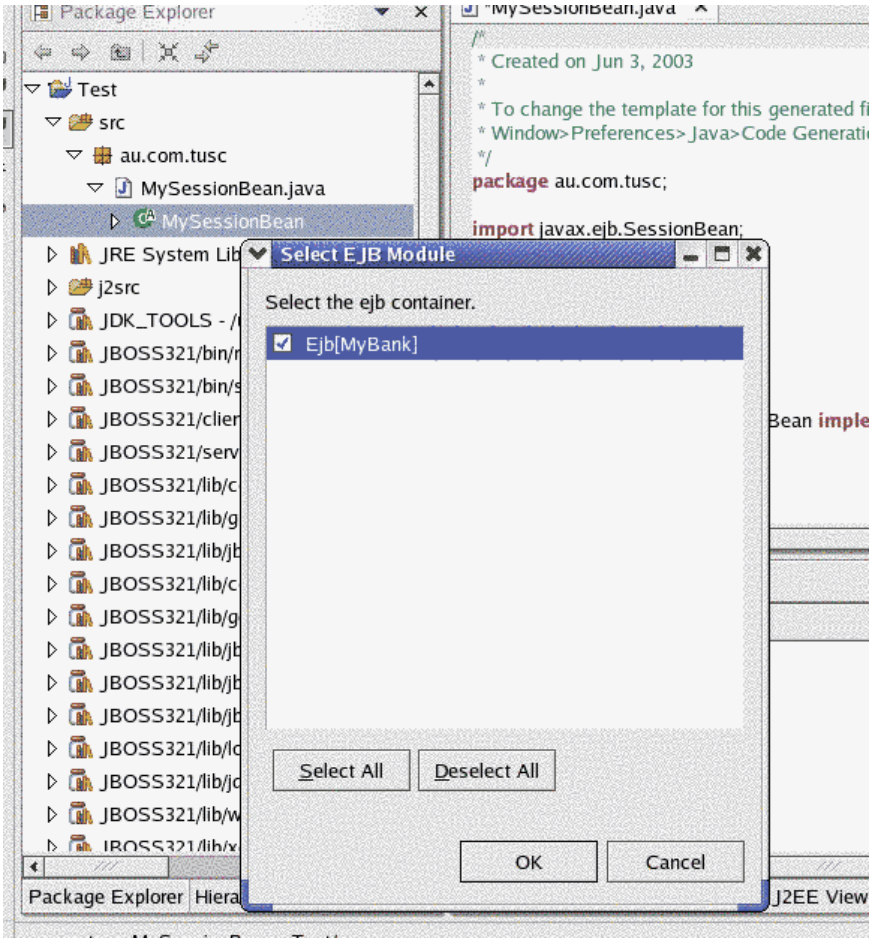
Code Snippet from Bean file

```
/**
 * @ejb.interface-method
 * tview-type="remote"
 *
 **/
   public  String learnJ2EE  (String messg)  {
      return "Me too! ";
   }
```

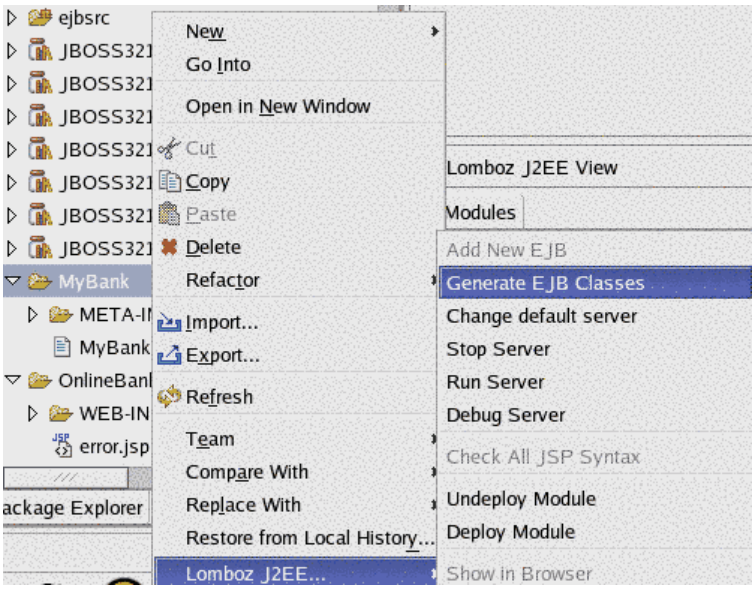
Save the file. Now we will generate the rest of the files (Home and Remote interfaces along with helper classes, using Xdoclet) required to deploy this bean.

First add this Bean to the module.

Go to Package Explorer > Test > MySessionBean.java > MySessionBean, right click on that; a menu will pop up. Go to Lomboz J2EE on this pop-up menu, and select 'Add EJB to module'.



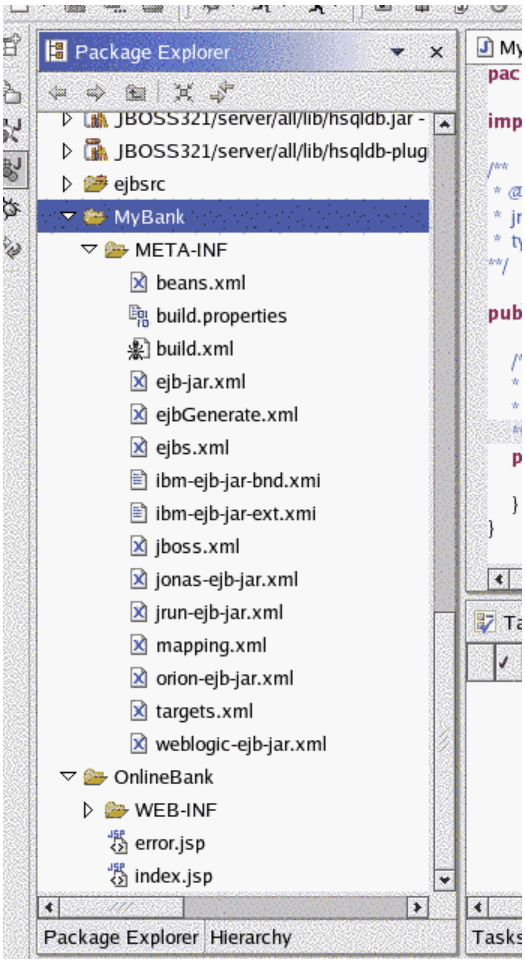
Go to Package Explorer > Test > MyBank (which is a directory), right click on that; a menu will pop up. Go to Lomboz J2EE, which is at the bottom of this pop up menu. Select Generate EJB classes. This will generate the required interfaces, helper classes and files related to deployment; that is, deployment descriptors.



It will create a new directory named ejbsrc, which will have a package named au.com.tusc, and under that, remote and home interfaces, along with necessary helper classes, will be generated.

Note: You will not edit any files generated by Xdoclet under 'ejbsrc' directory at any time while doing this tutorial. Because every time you use option Generate EJB Classes, it generates the necessary interfaces and helper classes as mentioned above. These files are generated by Xdoclet, after parsing your bean class, which is created by Lomboz bean creation wizard. Xdoclet looks at the various declared tags and methods in the bean class and then generates files accordingly. As a result, you just add business methods and their implementations in the bean class, and the rest is left to Xdoclet. Hence at any time in the development, you won't need to (and shouldn't!) edit any generated files. It will become clearer as we progress in this tutorial.

Also under MyBank > META-INF directory now there are 15 files, which includes ejb-jar.xml and jboss.xml and ejb-generate.xml. Initially there were only 6 files as noted earlier. These extra files are needed for deployment of the bean.

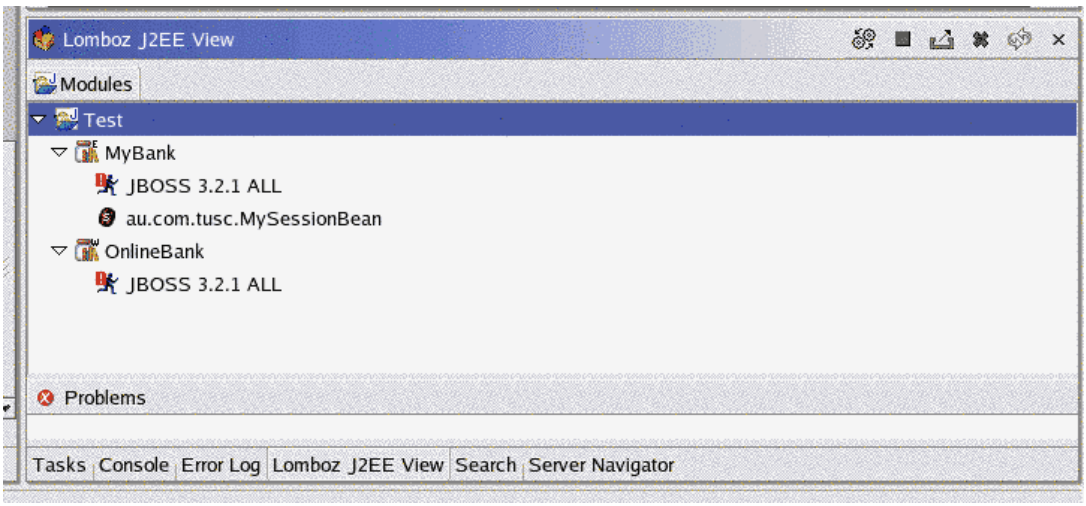


Now let us deploy this bean without going into any further details of deployment.

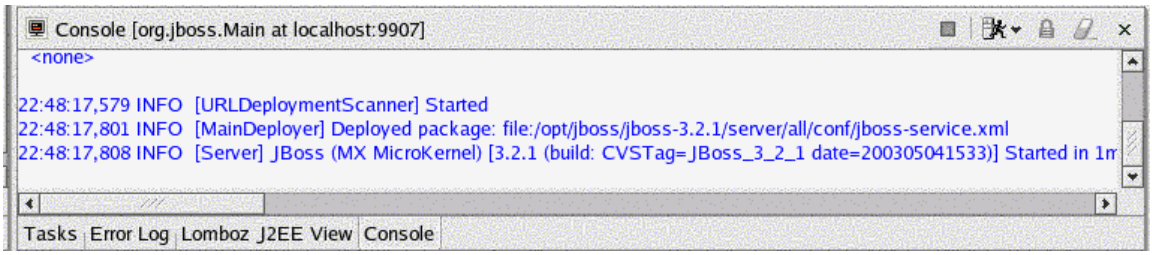
Go to Lomboz J2EE view in your workspace, expand Test > expand MyBank.

You will have au.com.tusc.MySessionBean added to your MyBank EJB module.

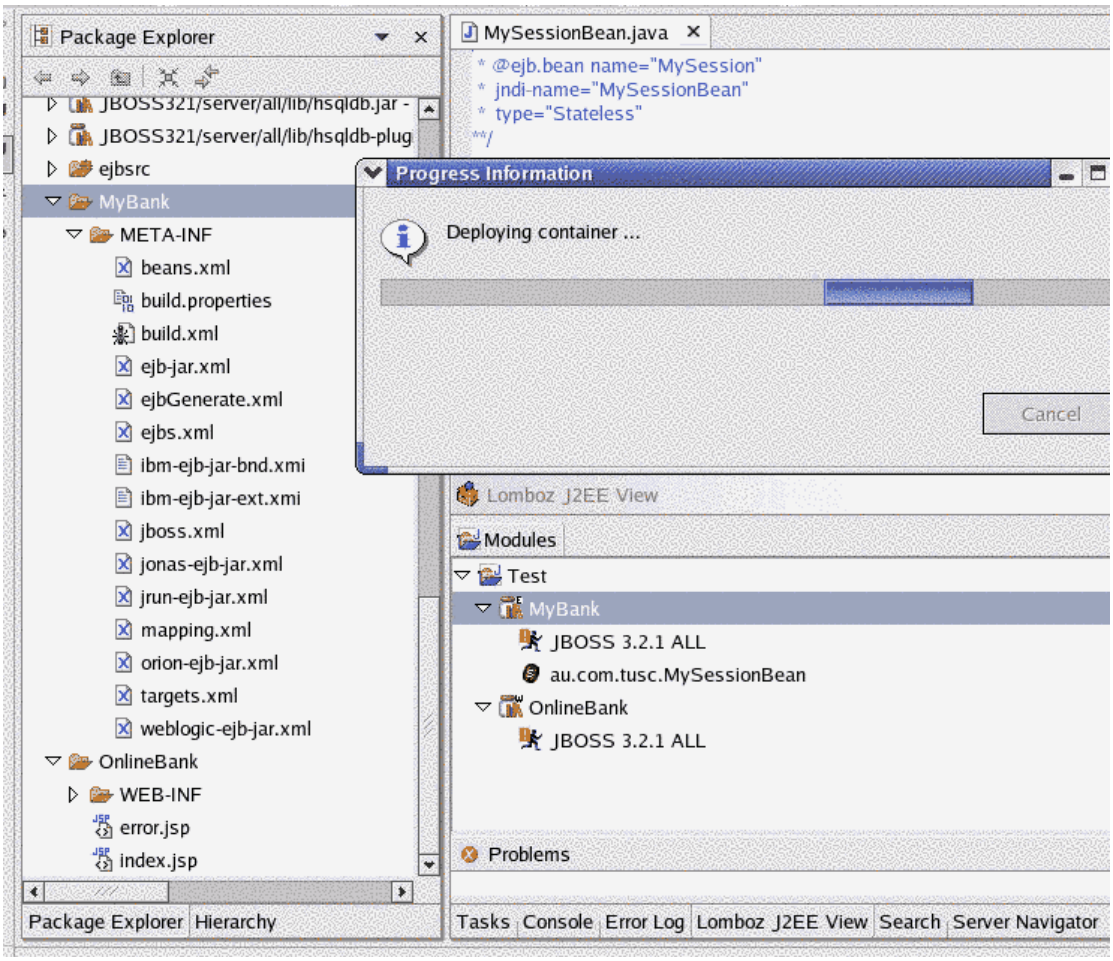
Right click on JBOSS 3.2.1 ALL icon as shown in fig below.



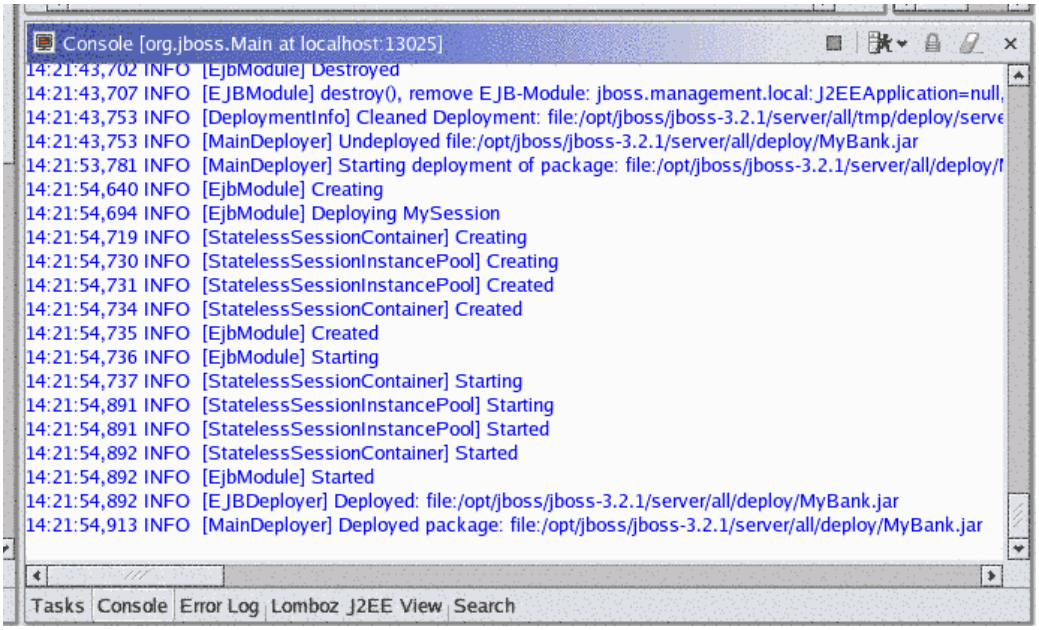
A menu will pop up; select option Debug Server. It will start the server with Debug mode and after successful start it will show this message in the console under your workspace as shown in the figure below.



Go to Lombo J2EE View > expand Test > select MyBank and right click on that; a menu will pop up; select option Deploy.



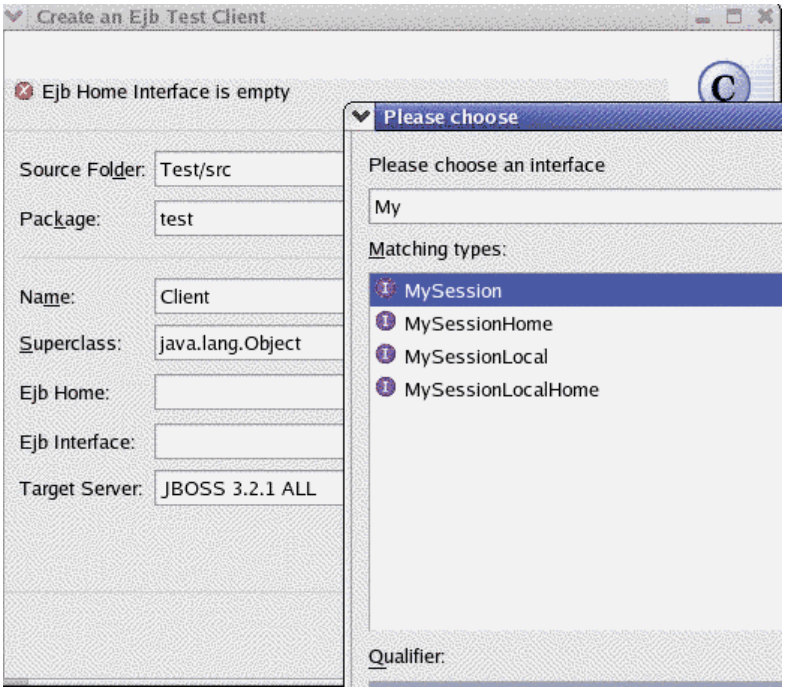
Once the bean is deployed, a message will come in your 'Console' confirming that, as shown in the figure below.



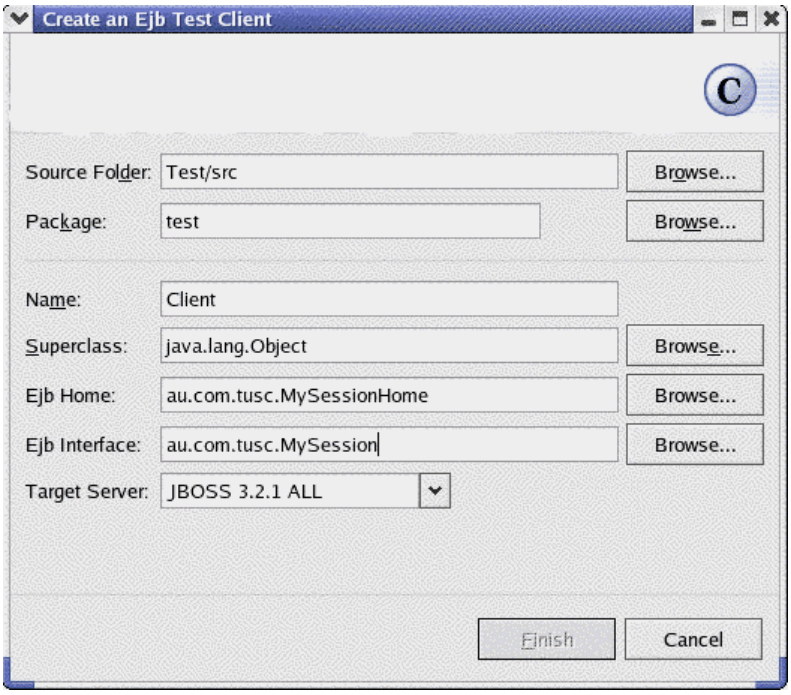
Now we will create a client to access this bean deployed on JBOSS (application sever).

Go to Package Explorer > Test > src, right click on that, a menu will pop up > select EJB Test client.

Enter Package name 'test' and Client name as 'Client'.

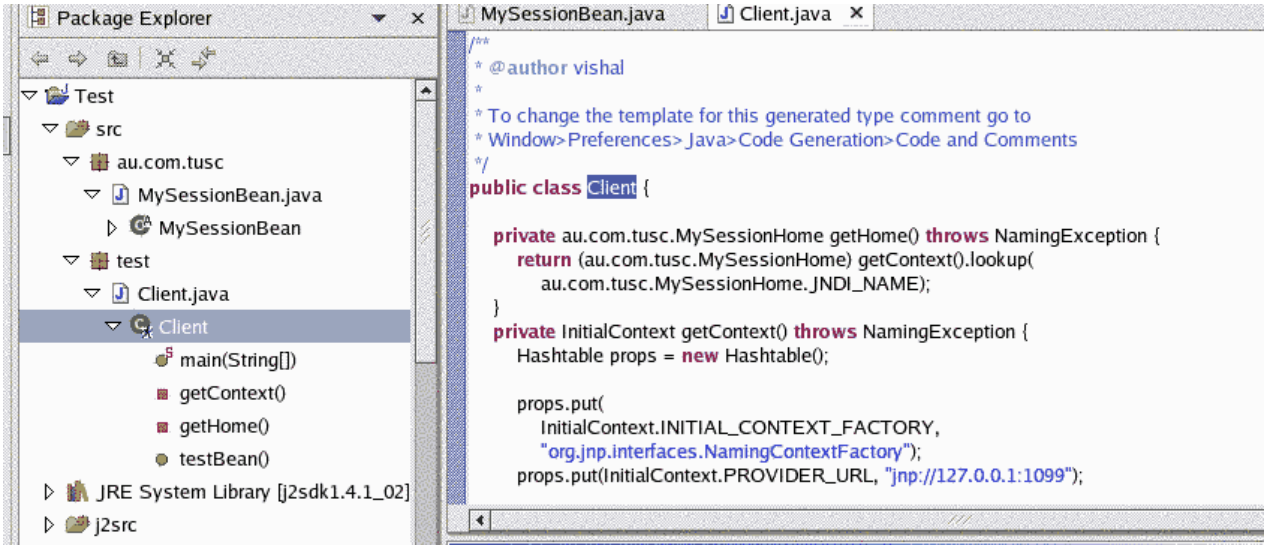


Select EjbHome as 'MySessionHome' and Ejb Interface as 'MySession' > Finish as shown in the figure below.



Now the client is generated, it's time to call business methods on the deployed bean.

We need some code to invoke the method on the bean..



Go to Client.java and write these lines under the 'testBean()' method.

String request = "I'm tired of 'Hello, world' examples..";

System.out.println("Request from client : " + request);

System.out.println("Message from server : " + myBean.learnJ2EE(request));

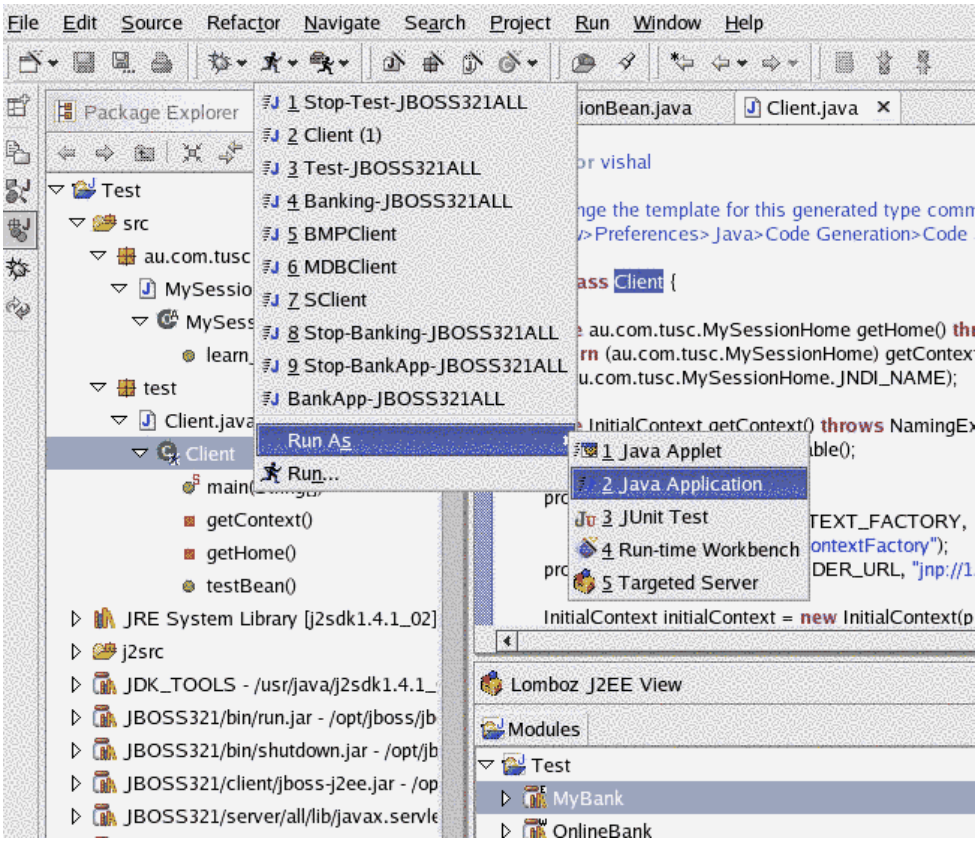
Code Snippet from Client.java.

```
public void testBean() {
   try {
       au.com.tusc.MySession myBean = getHome().create();
       //-------------------------------------
        //This is the place you make your calls.
        //System.out.println(myBean.callYourMethod());
       String request = "I'm tired of 'Hello, world' examples..";
       System.out.println("Request from client : " + request);
       System.out.println("Message from server : " + myBean.learnJ2EE(request) );

    } catch (RemoteException e) {
        e.printStackTrace();
    } catch (CreateException e) {
        e.printStackTrace();
    } catch (NamingException e) {
        e.printStackTrace();
```

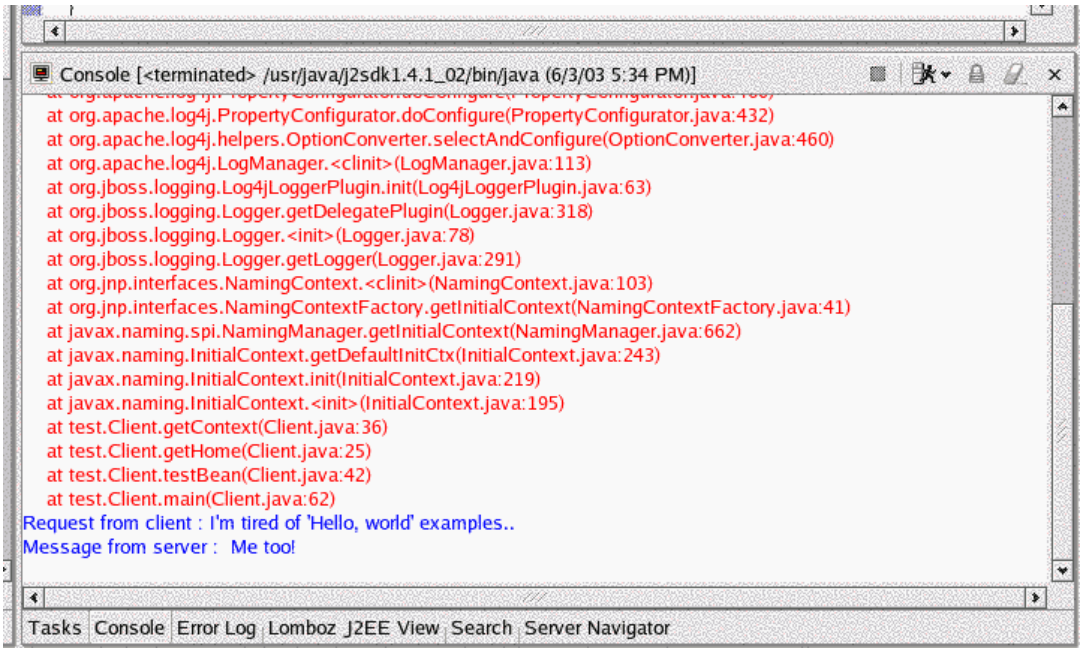
```
    }
}
```

Now first select Client node under Package Explorer. Go to top level menu and select this icon as shown below.

On this icon go to > Run as > Java Application.



Your Client is going to make a request, the results of which will be displayed in the console under your workspace, as shown below.



Note: Ignore these exceptions regarding org.apache.log4j.Property as it requires 'log4j.properties' on the clients path. We will cover this later on as this doesn't affect the functioning of the bean.

You have successfully created a bean and invoked an operation on it. Now, let's have a brief overview of J2EE concepts in the next chapter, before we start implementing the case study.

Prev TOC Next

Tutorial for building J2EE Applications using JBOSS and ECLIPSE

Chapter 2.

Overview Of J2EE Technology and Concepts

The Java 2 Enterprise Edition (J2EE) is a multitiered architecture for implementing enterprise-class applications and web based applications. This technology supports a variety of application types from large scale web applications to small client server applications. The main aim of J2EE technology is to create a simple development model for enterprise applications using component based application model. In this model such components use services provided by the container, which would otherwise typically need to be incorporated in application code. Note this may not be ideal in all scenarios: for example, a small scale application might be a better fit for a light-weight Java technology solution (e.g. Servlets, JSPs, etc.).

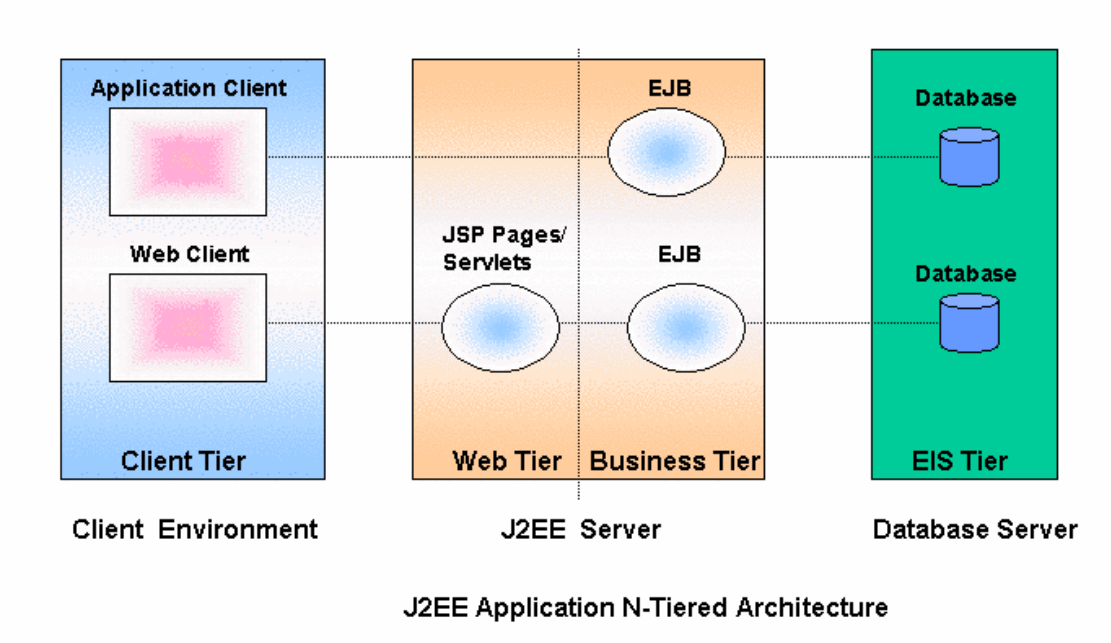
J2EE Components :

J2EE applications are made up 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 and that communicates with other components in the application. The J2EE specification defines the following J2EE components:

Application clients and applets are components that run on the client.

Java Servlet and JavaServer Pages technology components are Web components that run on the web server.

Enterprise JavaBeans components (enterprise beans) are business components that run on the application server.



J2EE Application N-Tiered Architecture

All these 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 the J2EE application server.

In addition to these primary components, it includes standard services and supporting technologies which are :

Java Database Connectivity (JDBC) technology provides access to relational database systems.

Java Transaction API (JTA) or Java Transaction Service (JTS) provides transaction support for J2EE components.

Java Messaging Service (JMS) for asynchronous communication between J2EE components.

Java Naming and Directory Interface (JNDI) provides naming and directory access.

Note : All J2EE components are written in the Java programming language

J2EE Clients :

There are normally two types of J2EE client: a Web client and an Application client as shown above in figure.

A Web client consists of two parts, dynamic Web pages containing various types of markup language (HTML, XML and others), which are generated by Web components running in the Web tier, and a Web browser, which draws the pages received from the server. Another category of web clients are sometimes called as thin client. Thin clients usually do not do things like query databases, execute complex business rules, or connect to any legacy applications. When we use a thin client, heavyweight operations like these are handled by enterprise beans executing on the J2EE server where they can leverage the security, speed, services, and reliability of J2EE server-side technologies.

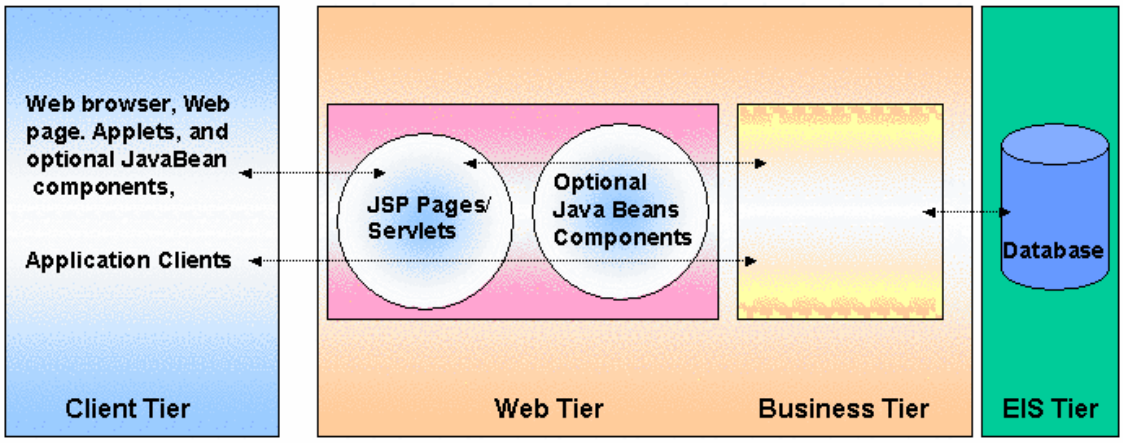
A Web page received from the Web tier can include an embedded applet. An applet is a small client application written in the Java programming language that executes in the Java virtual machine installed in the Web browser. However, client systems will need the Java Plug-in and possibly a security policy file in order to execute applets successfully in the Web browser. Web components are often a preferred API for creating a Web client program because no plug-ins or security policy files are needed on the client systems. Moreover this allows cleaner and more modular application design because they provide a means to separate application logic from Web page design.

An Application client runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It normally has a graphical user interface (GUI) created using the Swing or Abstract Window Toolkit (AWT) APIs. Application clients directly access enterprise beans running in the business tier. But if there is need for a web client it can open an HTTP connection to establish communication with a servlet running in the Web tier.

Note: If required, a command line interface can be used.

Web Components :

J2EE Web components can be either servlets or JSP pages. Servlets are Java programming language classes that dynamically process requests and construct responses. JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content. HTML pages and applets are bundled with Web components during application assembly, but are not considered Web components by the J2EE specification. Similarly, server-side utility classes can also be bundled with Web components like HTML pages, but are not considered Web components by the J2EE specification. Shown below in figure is Web components communication.

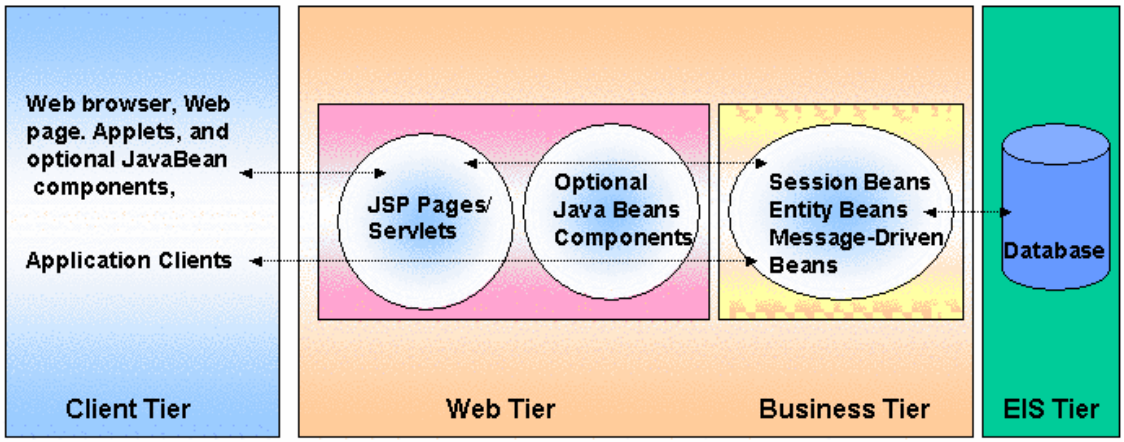


Web Components Communication

The Web tier might include a JavaBeans component to manage the user input and send that input to enterprise beans running in the business tier for processing as shown above in the figure..

Business Components :

Business code, which is logic that solves or meets the needs of a particular business domain such as banking, retail, or finance, is handled by enterprise beans running in the business tier.



Business Components Communication

The figure above shows communication with business components, where an enterprise bean receives data from client programs, processes it (if necessary), and sends it to the enterprise information system tier for storage. An enterprise bean also retrieves data from storage, processes it (if necessary), and sends it back to the client program.

There are three kinds of enterprise beans: session beans (stateless and stateful), entity beans (bean managed and container managed), and message-driven beans. A session bean represents a transient conversation with a client. When the client finishes executing, the session bean and its data are gone. In contrast, an entity bean represents persistent data stored in one row of a database relation/table. If the client terminates or if the server shuts down, the underlying services ensure that the entity bean data is saved. A message-driven bean combines features of a session bean and a Java Message Service (JMS) message listener, allowing a business component to receive JMS messages asynchronously.

Note : Java Beans are not considered J2EE components by the J2EE specification as JavaBeans are different from Enterprise Beans. JavaBeans component architecture can be used in both server and client tiers to manage the communication between an application client or applet and components running on the J2EE server or between server components and a database, whereas Enterprise JavaBeans components are only used in the business tier as a part of the server tier. JavaBeans have instance variables and has an accessor and mutator methods to access properties of bean or say, accessing the data in the instance variables which simplifies the design and implementation of JavaBeans components.

Enterprise Information System Tier :

The enterprise information system tier handles enterprise information system software and includes enterprise infrastructure systems such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and other legacy information systems. J2EE application components might need access to enterprise information systems for database connectivity.

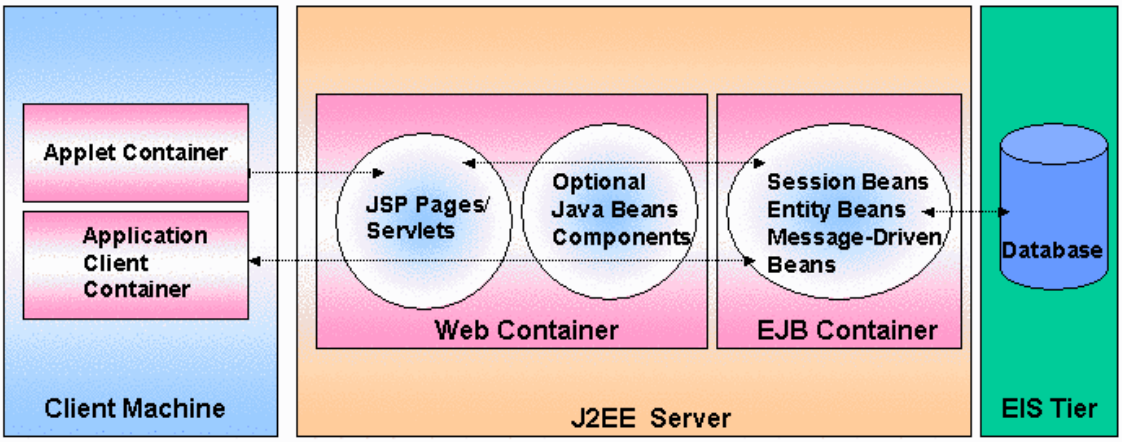
J2EE Containers :

J2EE containers provide access to the underlying services of the J2EE Server environment via containers for different types of components. Traditionally, application developers have to write code for handling transaction, state management, multithreading, resource pooling etc. Now the J2EE container provides these services allowing you to concentrate on solving business problems.

Containers are the interface between a component and the low-level platform-specific functionality that supports the component. For example, before a Web, enterprise bean, or application client component can be executed, it must be assembled into a J2EE application and deployed into its container.

The assembly process involves specifying container settings for each component in the J2EE application and for the J2EE application itself. Container settings customize the underlying support provided by the J2EE server, which includes services such as Java Naming and Directory Interface, security, transaction management etc.

The J2EE server provides Enterprise JavaBeans (EJB) and Web containers. EJB container manages the execution of enterprise beans for J2EE applications, whereas Web container manages the execution of JSP page and servlet components for J2EE applications. Other than these two containers there are an Application client container and Applet container, which are not part of J2EE server as these reside on the client's machine as shown below.



An Application client container manages the execution of application client components whereas an Applet container manages the execution of applets. These are typically the JRE (Java Runtime Environment) and a Java-enabled Web browser respectively.

Packaging :

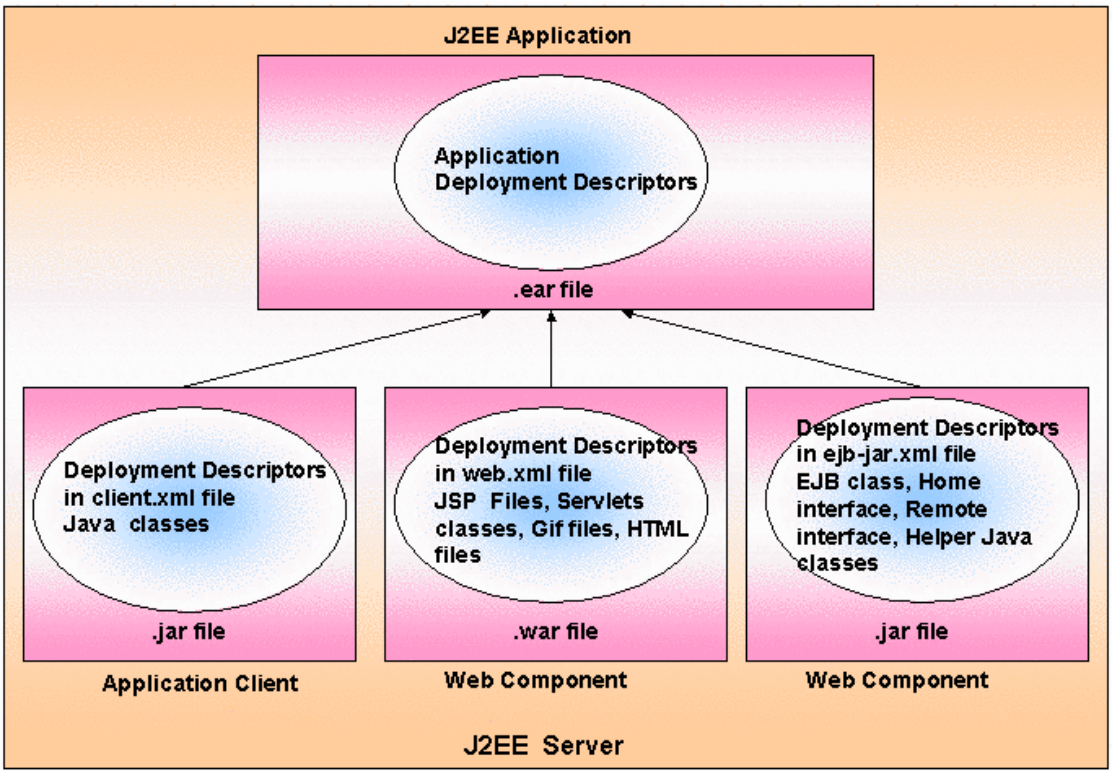
In order to deploy a J2EE application, after developing different components, it is packaged into special archive files that contain the relevant class files and XML deployment descriptors. These XML deployment descriptors contain information specific to each bundled component and are a mechanism for configuring application behavior at assembly or deployment time. These are bundled into different archive types for different component types.

Web components are archived in Web Archive (.war) file which contains servlets, JSP and static components such as HTML and image files. The .war file contains classes and files used in web tier along with a Web component deployment descriptor.

Business components are archived in Java Archive (.jar) file which contains an EJB deployment descriptor, remote, and object interface files along with helper files required by EJB component.

Client side class files and deployment descriptors are archived in Java Archive (.jar) file which make up the client application.

J2EE application is bundled in an Enterprise Archive (.ear) file which contains the whole application along with deployment descriptor that provides information about the application and its assembled components.



J2EE Platform Roles :

Building the different components of a J2EE application involves various roles in the development, deployment and management of an enterprise application.

The application component provider develops the reusable components of J2EE application, which can be Web components, enterprise beans, applets, or application clients for use in J2EE applications.

The application assembler takes all building blocks from the application component provider and combines them into J2EE applications.

The deployer is responsible for the installation/deployment of components in a J2EE environment or J2EE server.

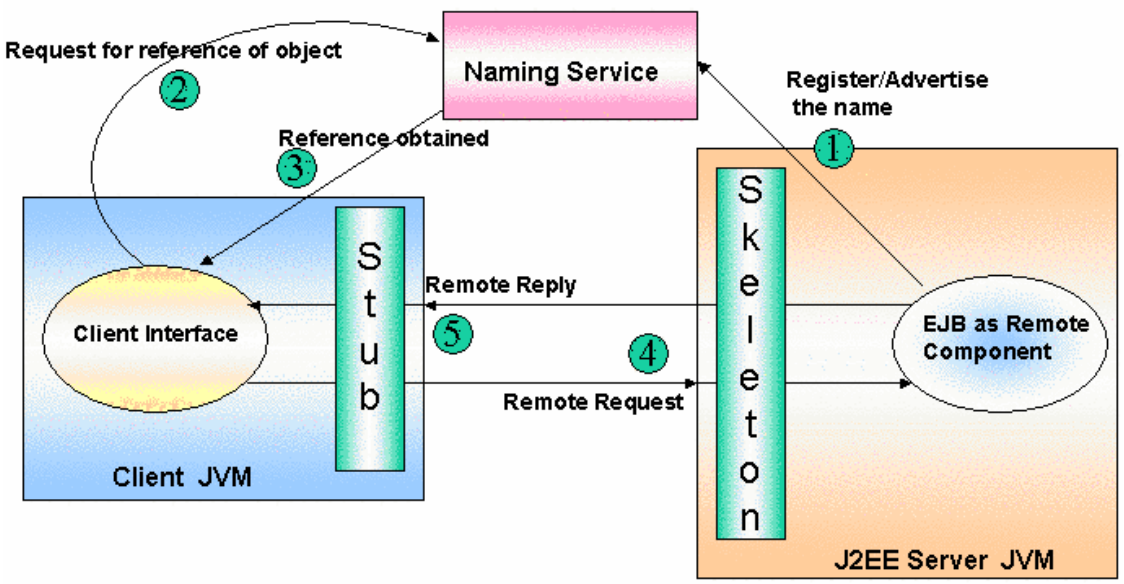
The system administrator is responsible for configuration and administration of computing systems in an enterprise.

The tool provider is a vendor used to develop, package and deploy J2EE applications.

Note : All these above mentioned roles can be assigned to a person or an organization.

Distributed Architecture in J2EE :

All the J2EE applications implement a distributed architecture. In this an object is associated with a name, where names are provided by naming service, by advertising to various components and resolving client references to these service components as shown in figure below.



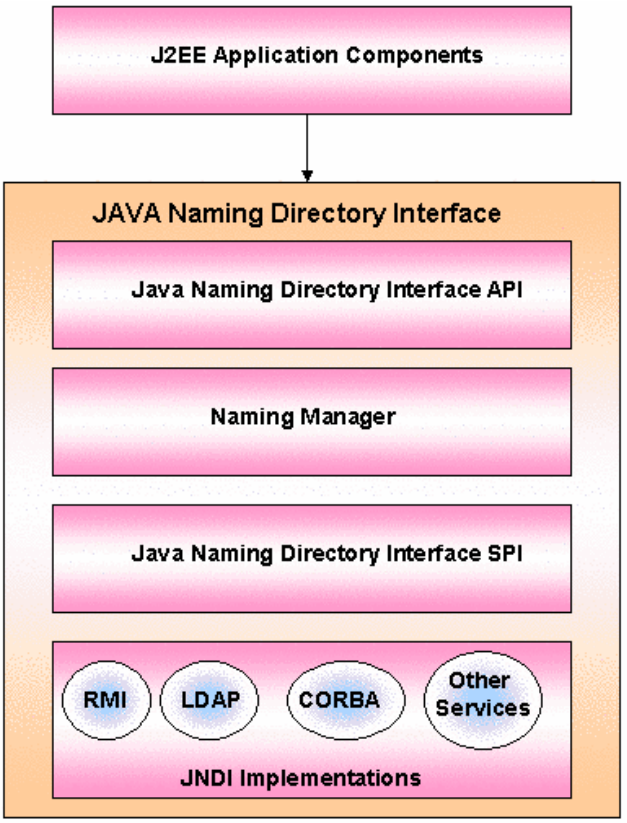
As a result of that, object references are obtained, by looking up for an object by its advertised name, once found, reference is obtained, and then carry out the necessary operations on that object using the host's services. A remote object advertises its availability with the name service using a logical name and the name service translates the name to the physical location of the object in the J2EE environment. Once the client request obtains a reference to a remote component, it can send requests to the remote component. The runtime system system handles the distributed communication between the remote objects, which includes serialization and deserialization of parameters.

Note : Various naming services used in distributed systems are RMI (for Java-only implementations), CORBA naming services, LDAP, DNS, NIS. The JBOSS application server uses RMI as its naming service.

Serialization and Deserialization are the same as marshalling and unmarshalling for those familiar with RPC terminology.

Java Naming Directory Interface (JNDI) Architecture :

J2EE uses the JNDI API to generically access naming and directory services using Java technology. The JNDI API resides between 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 in a naming service as mentioned above. The client code remains unchanged, regardless of what naming service is used or what technology it is based on, as this doesn't make any difference to clients locating remote objects via the JNDI API.

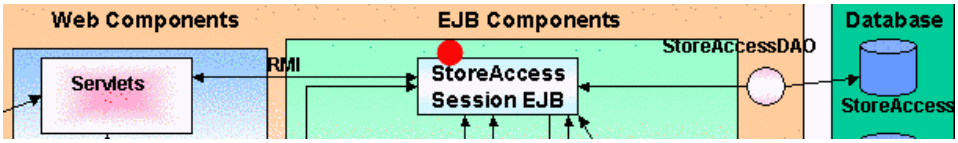
Prev TOC Next

Tutorial for building J2EE Applications using JBOSS and ECLIPSE

Chapter 3.

Creating a Stateless Session Bean

This chapter covers how to create a stateless session EJB component. This bean will be responsible for authenticating the user by communicating with the database using Data Access Object (DAO) which encapsulates Java Database Connectivity (JDBC) code. A DAO has all attributes (fields) and behavior (methods) corresponding to the bean it is being used for.



All customers, supplier and manager of MyStore have been assigned a unique username and userid to access services of MyStore, but in order to access these services all these entities have to first login into the system (MyStore). The method for authentication is named **loginUser**, which takes **two String parameters, username and password and returns the userID if authentication is successful.**

Note : This method loginUser is a business method, normally business methods carry out operations or processing on values EJB components. From clients perspective, clients can see only business methods and invoke them on bean.

Tasks :

1. Create a J2EE project named MyStore.
2. Create a Stateless Session Bean named StoreAccess.
3. Add a business method in bean named loginUser with the following signature

   public String loginUser (String username, String password)
4. Create a DAO named StoreAccessDAOImpl under package au.com.tusc.dao. Generate the DAO interface.
5. Implement the method named loginUser, generated in DAO interface, in StoreAccessDAOImpl. Method signature is

   public String loginUser (String username, String password)
6. Add callback methods and implement them.
7. Deploy StoreAccess bean.
8. Create your test client named SessionClient under package au.com.tusc.client.
9. Run your client and test the bean.

Create J2EE Project :

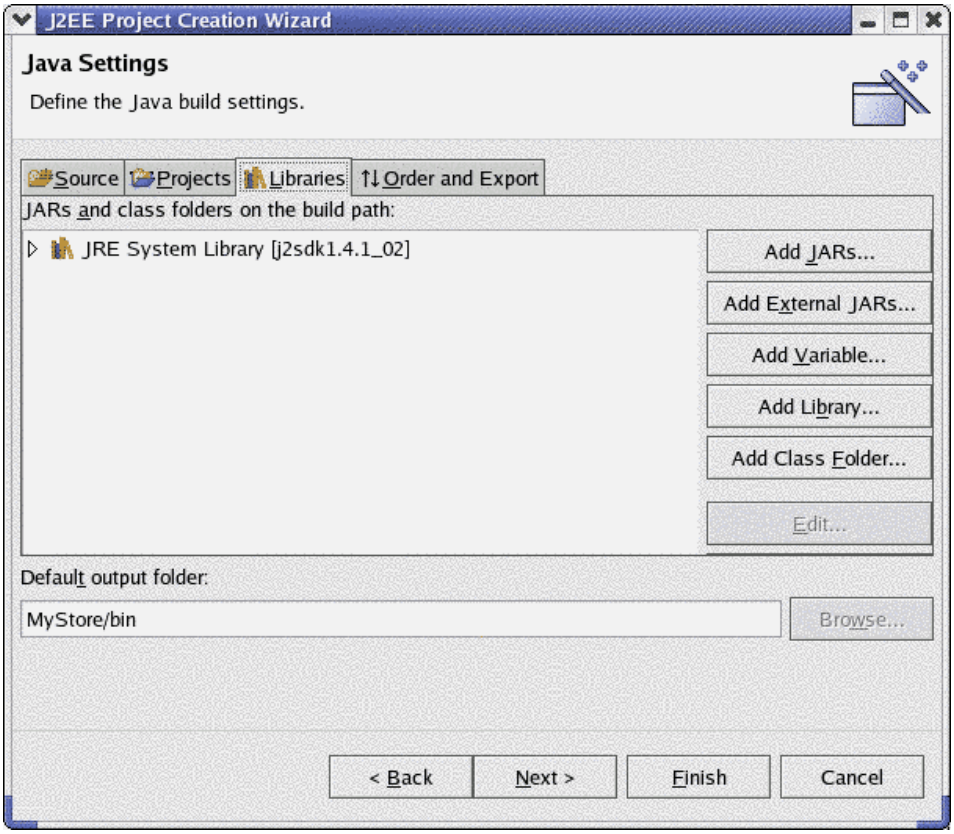
Now, lets start to write our first component of this tutorial.

Go to File > New > LombozJ2EE Project, project creation wizard will pop up.

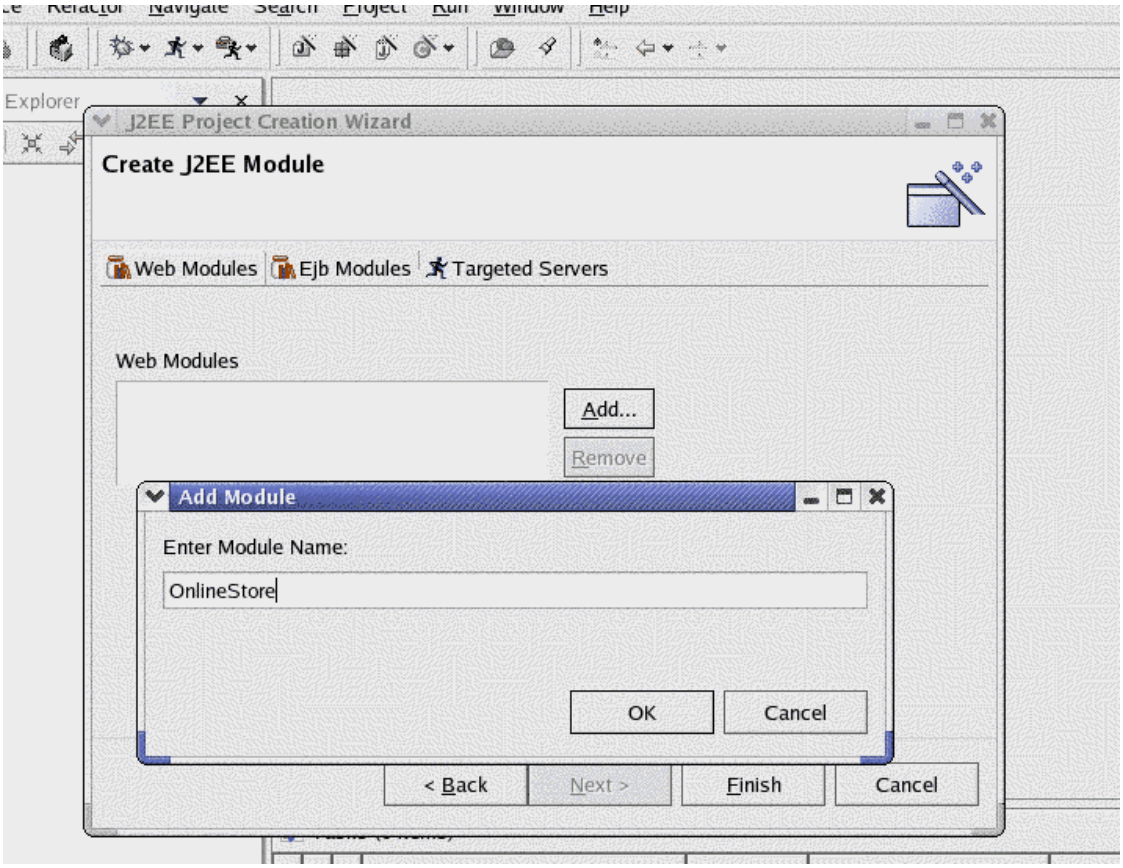
Insert Project Name MyStore > Next .

Under Java Settings Check source, should be MyStore/src , libraries pointing to $JAVA_HOME > Go Next as shown in fig below.

Note: This step is shown in chapter1, as there is a bug in eclipse 2.1, so its important that you check your library settings are right.



Under Create J2EE Module, select Web Modules tab > Add.., enter Module name as OnlineStore > OK as shown in figure below.



Under Create J2EE Module, select EJB Modules tab > Add.., enter Module name as MyStoreMgr > OK .

Under Create J2EE Module, select Targeted Servers tab > Select JBOSS 3.2.1 ALL > Add.. > Finish.

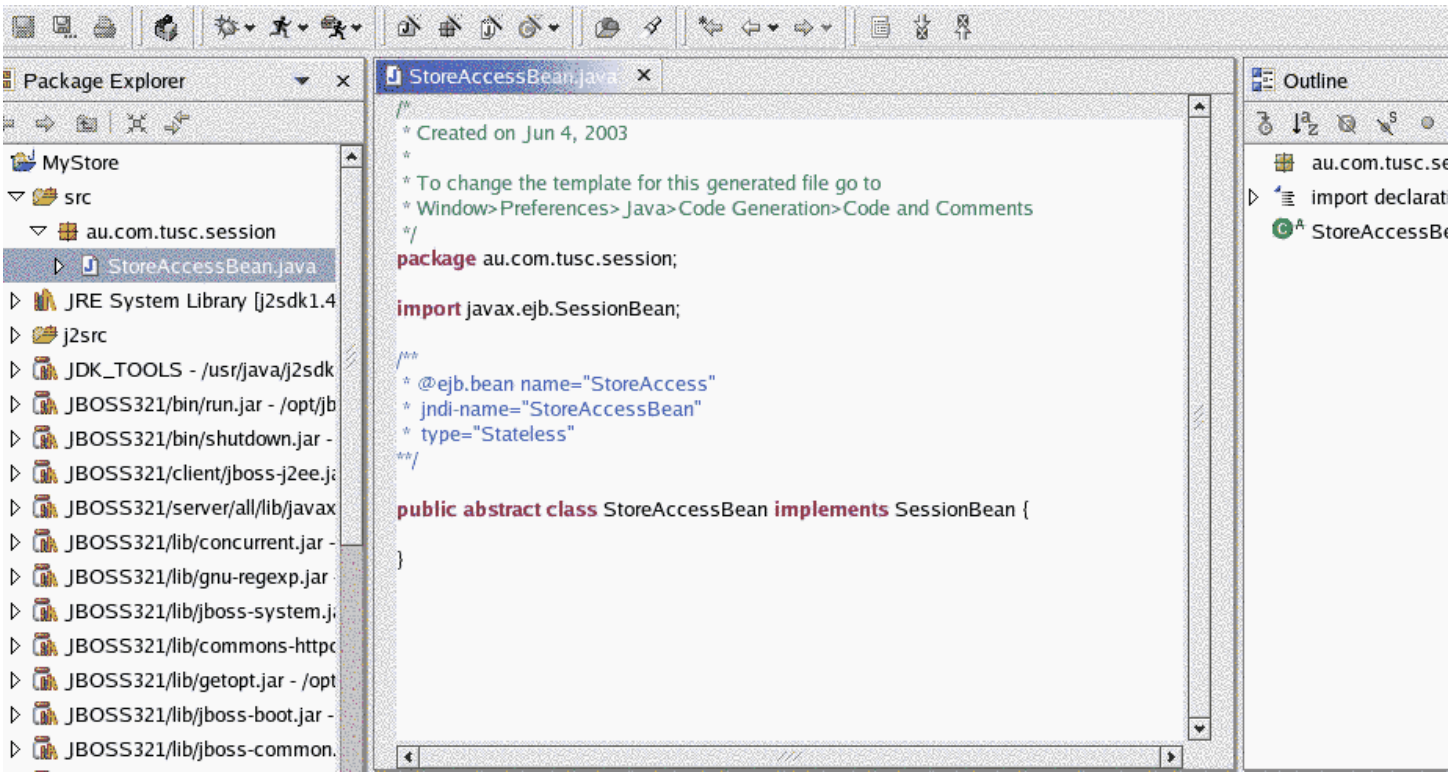
Create Stateless Bean :

Go To Package Explorer > Expand Mystore (project) node > select src, right click and menu will pop up.

On pop up menu > New > Lomboz EJB Creation Wizard.

Enter package name au.com.tusc.session, bean name StoreAccess and select bean type as stateless > Finish.

This will create a package named au.com.tusc.session under src and StoreAccessBean under that package as shown in the figure below.

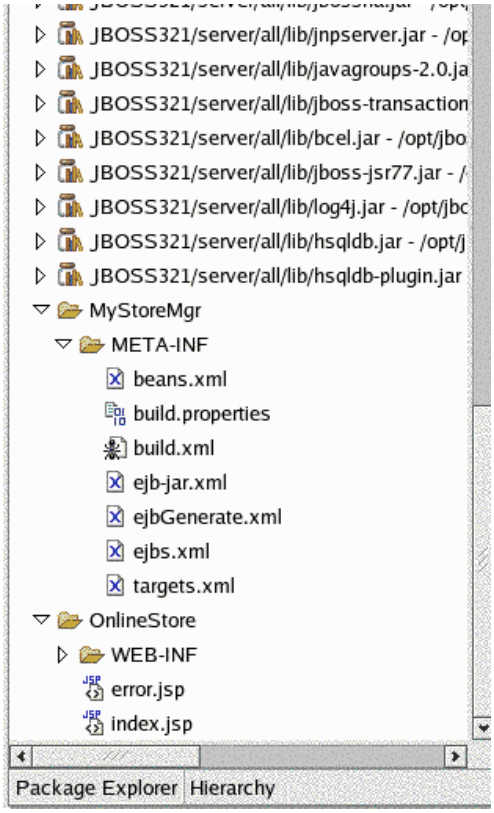


As we can see from the figure below it has created a class level tag @ejb.bean, which has assigned the bean type, name and its JNDI name which will be generated in Home interface. This tag will also generate deployment descriptors in ejb-jar.xml and jboss.xml file as well once you generate your EJB classes, which is covered later on in this chapter.

```
package au.com.tusc.session;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;



/**
 * @ejb.bean name="StoreAccess"
 * jndi-name="StoreAccessBean"
 * type="Stateless"
```

Note: It will generate the bean name, jndi-name and type of bean in the file. Also, the name of file is appended with word 'Bean' as you gave the name of the bean as StoreAccess only. Again, be careful with naming conventions, specifying the bean name only in the wizard without adding the word 'Bean' to the name as the wizard appends that for you.

Expand MyStoreMgr/META-INF node under Package Explorer. You will find there are seven files which are generated by Lomboz using Xdoclet as shown in the figure below.



Now we are going to generate all the interfaces including Home, Remote, DAO and other helper classes. We will explain why later on, but for the time being just follow the steps.

But before we get too excited, there are a few concepts to cover here.

Go to MyStoreMgr/META-INF > select and open ejbGenerate.xml.

Note: Lomboz uses this file to generate required interfaces and helper classes, so in the event that you have special needs then you will have to customize this file. See ejbdoclet under the Xdoclet documentation.

'ejbGenerate.xml' file is generated only once when you create your EJB module. So any changes made in this file will be reflected even if you modify your bean class and generate your classes again and again.

As we can see from the code snippet of file shown in figure at right, there are following tags defined.

<dataobject/> is defined for generating data Objects for holding values of EJB component's persistent fields, which correspond to columns in the associated table in the database.

Note: <dataobject/> has been deprecated in favour of Value Object which is more powerful in terms of relationships (1-1, 1-n and n-m).

<utilobject/> Creates method for generating GUID and for accessing Remote and Local Home objects.

<remoteinterface/> Generates remote interfaces for EJBs.

<localinterface/> Generates local interfaces for EJBs.

<homeinterface /> Generates remote home interfaces for EJBs.

<localhomeinterface/>Generates local home interfaces for EJBs.

<entitypk/>Generates primary key classes for entity EJBs.

<entitybmp/>Creates entity bean classes for BMP entity EJBs.

<entitycmp/>

<session/> Generates session bean class.

Note : There is no tag for generating a DAO.

ejbGenerate.xml

```
<ejbdoclet
    destdir="${ejbsrc.dir}"
    mergedir="${ejb.dd.dir}"
    excludedtags="@version,@author,@todo"
    addedtags="@lomboz generated"
    ejbspec="2.0"
    force="${xdoclet.force}"
    verbose="true" >

  <fileset dir="../src" defaultexcludes="yes">
    <patternset includesfile="META-INF/ejbs.xml" />
  </fileset>

    <dataobject/>
    <valueobject/>
    <utilobject cacheHomes="true" includeGUID="true"/>

    <remoteinterface/>
    <localinterface/>
    <homeinterface />
    <localhomeinterface/>

    <entitypk/>
    <entitycmp/>
    <entitybmp/>
    <session/>

    <deploymentdescriptor
      destdir="${ejb.dd.dir}"
      validatexml="false"
      mergedir="${ejb.dd.dir}"
      />
```

```
        mergedir="${ejb.dd.dir}"
        />
```

So, we have to include this <dao/> tag.

For details, please refer ejbdoclet under Xdoclet documentation.

As we can see from the code snippet from this file the following tags are defined.

<jboss/> is a JBOSS specific tag required for JBOSS. You have to specify datasource, datasourcemapping and preferredrelationmapping. As it differs for different databases, so you may have to specify values appropriate to your environment. If these tags are commented out in JBOSS they default to the correct values for the built-in Hypersonic SQL database, but for the moment we'll set them anyway.

ejbGenerate.xml

```
<jboss
 version="3.0"
 unauthenticatedPrincipal="nobody"
 xmlencoding="UTF-8"
 destdir="${ejb.dd.dir}"
 validatexml="false"
 datasource="PLEASE_MODIFY_THIS"
 datasourcemapping="PLEASE_MODIFY_THIS"
 preferredrelationmapping="PLEASE_MODIFY_THIS"
/>
```

The other two files which are of importance to us are ejb-jar.xml and jboss.xml. The file ejb-jar.xml has all the deployment descriptors for beans and jboss.xml has the JBOSS specific deployment descriptors required by JBOSS.

Note : ejb-jar.xml file is generated every time you generate interface and helper classes for your bean. For the first time, it is empty, and jboss.xml will be generated every time when you will generate your classes for your bean.

Setup DAO :

Now, let's customize ejbGenerate.xml for setting up a DAO.

We have included a <dao> tag specifying the destination directory for the generated DAO interface and what pattern to be used.

ejbGenerate.xml

```
<ejbdoclet
    destdir="${ejbsrc.dir}"
    mergedir="${ejb.dd.dir}"
    excludedtags="@version,@author,@todo"
    addedtags="@lomboz generated"
    ejbspec="2.0"
    force="${xdoclet.force}"
    verbose="true" >


 <fileset dir="../src" defaultexcludes="yes">
   <patternset includesfile="META-INF/ejbs.xml" />
 </fileset>

   <dataobject/>
   <valueobject/>
   <utilobject cacheHomes="true" includeGUID="true"/>
   <dao pattern="{0}" destDir="${ejbsrc.dir}" />

   <remoteinterface/>
   <localinterface/>
   <homeinterface />
   <localhomeinterface/>
```

Note : For details, please refer ejbdoclet under Xdoclet documentation.

We have included the datasource, datasoucremapping and preferredrelationmapping as shown in code snippet of ejbGenerate.xml file on right.

datasource="java:/DefaultDS" is a local JNDI name for data source to be used.

datsourcemapping="Hypersonic SQL" maps data object/value objects and their types to columns and data types associated with these columns.

preferredrelationmapping="foreign-key" defines type of database to be used.

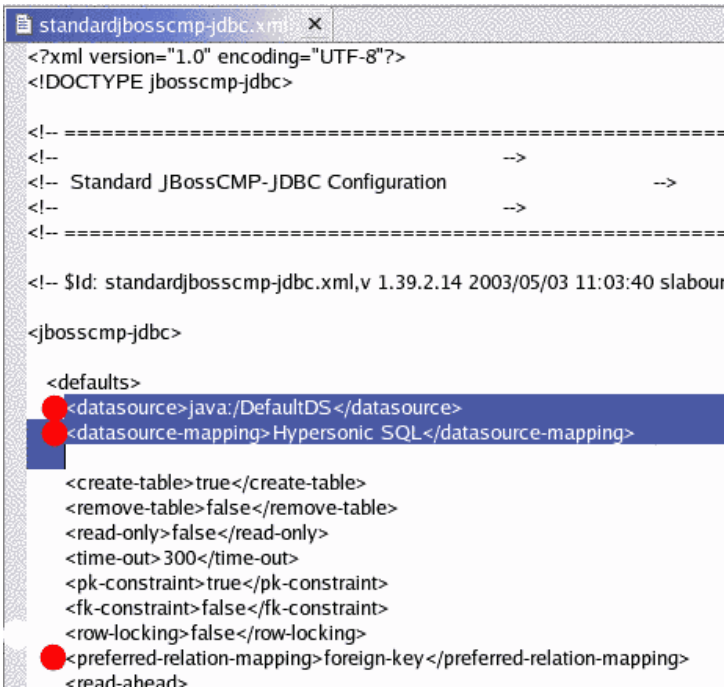
Note : For more details, please refer JBOSS documentation.

ejbGenerate.xml

```
    persistence="weblogic"
/>
<jboss
 version="3.0"
 unauthenticatedPrincipal="nobody"
 xmlencoding="UTF-8"
 destdir="${ejb.dd.dir}"
 validatexml="false"
 datasource="java:/DefaultDS"
 datasourcemapping="Hypersonic SQL"
 preferredrelationmapping="foreign-key"
/>
```

Since we are using the Hypersonic database, these parameters are appropriate to that. These parameters relate to the configuration file standardjbosscmp-jdbc.xml which controls the CMP-to-JDBC mappings for JBOSS. This resides in $JBOSS_HOME/server/conf/ , e.g. /opt/jboss/jboss-3.2.1/server/default/conf/.

Code snippet from standardjbosscmp-jdbc.xml is shown in figure at right.

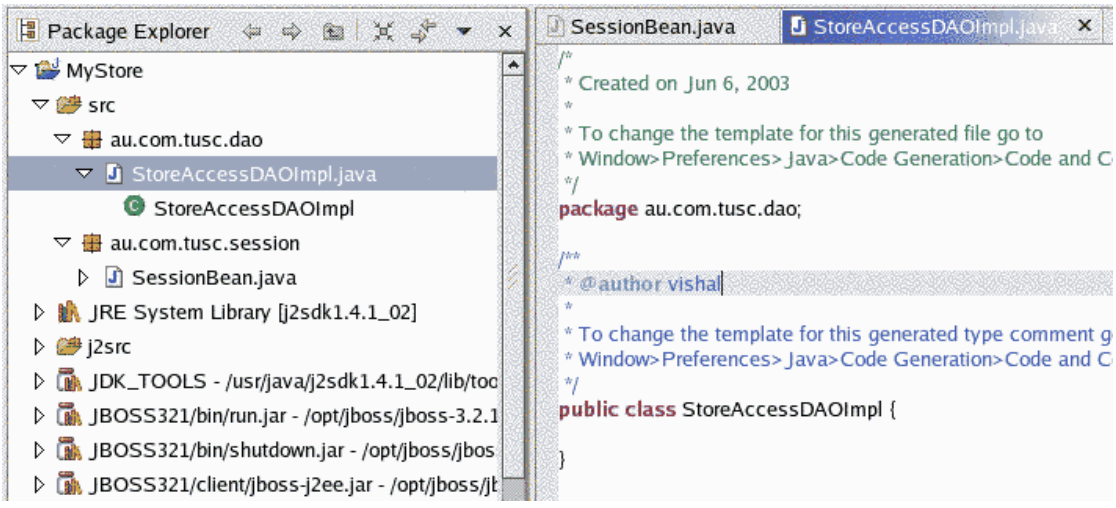


Note : The way Xdoclet works is bit different from some conventional styles of programming, as the Xdoclet tags will generate these (home and remote) interfaces along with necessary helper classes, which then will used in Bean and DAO Implementation class. However, until these are generated, we cannot write any business methods in Bean and JDBC wrappers in the DAO Implementation class. If this seems confusing then just follow the steps, and hopefully it will become more clear.

Create DAO Interface :

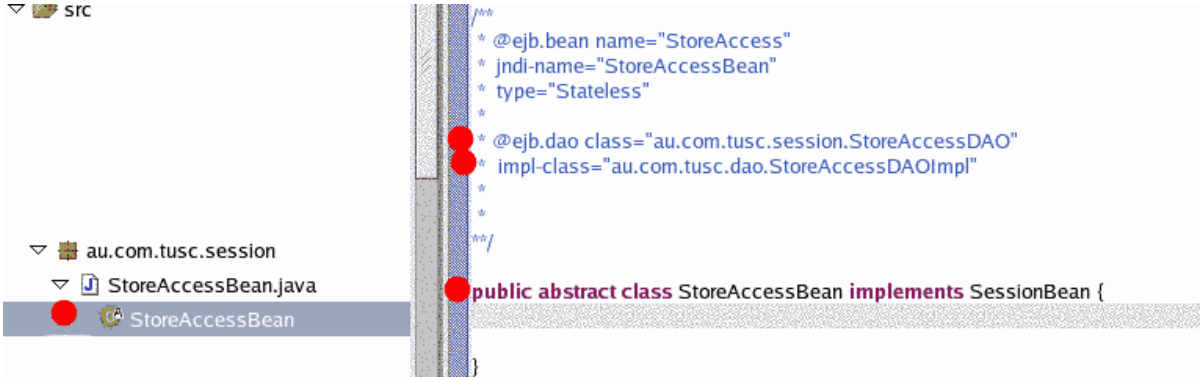
Since we are going to use a DAO to access the database for this Stateless Bean, we have to create a DAOImpl class which will implement that generated DAO interface.

Go to src > add package named au.com.tusc.dao > Add a class StoreAccessDAOImpl in that package.



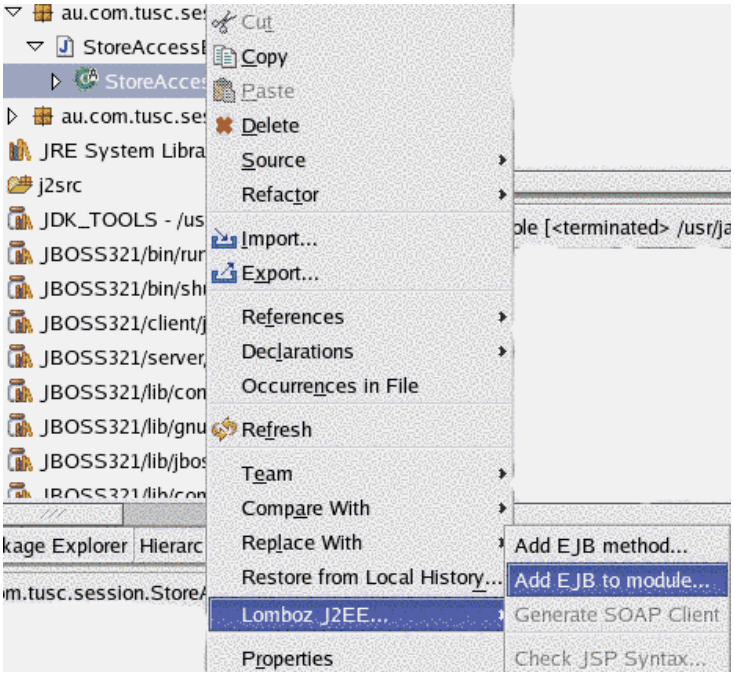
Now go to your Bean class and declare this tag at the class level (that is at the top) as shown below to generate the DAO interface.

```
@ejb.dao class="au.com.tusc.session.StoreAccessDAO"
  impl-class="au.com.tusc.dao.StoreAccessDAOImpl"
```



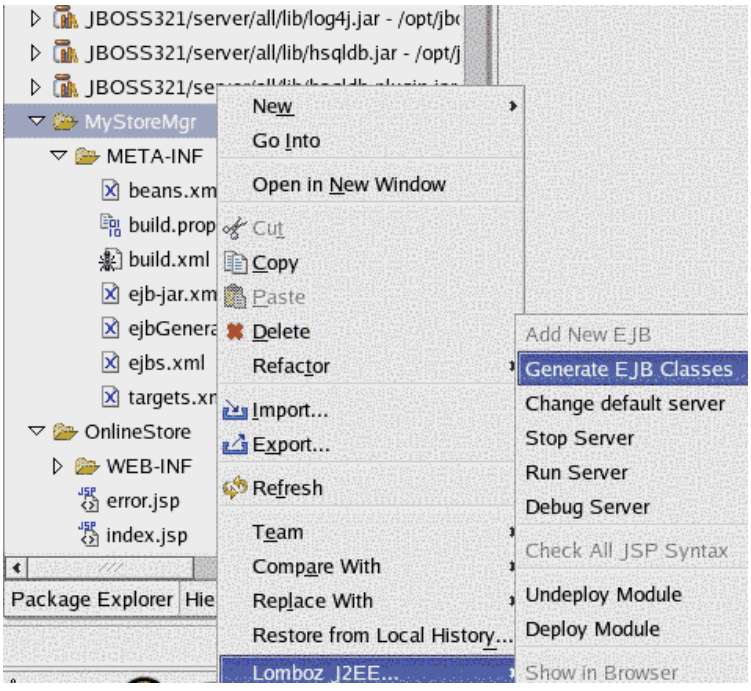
Expand StoreAccessBean node under Package Explorer. Right click and a pop up menu will appear.

On that menu go to Lomboz J2EE > Add EJB to module. Select EJB '[MyStoreMgr]' > OK.

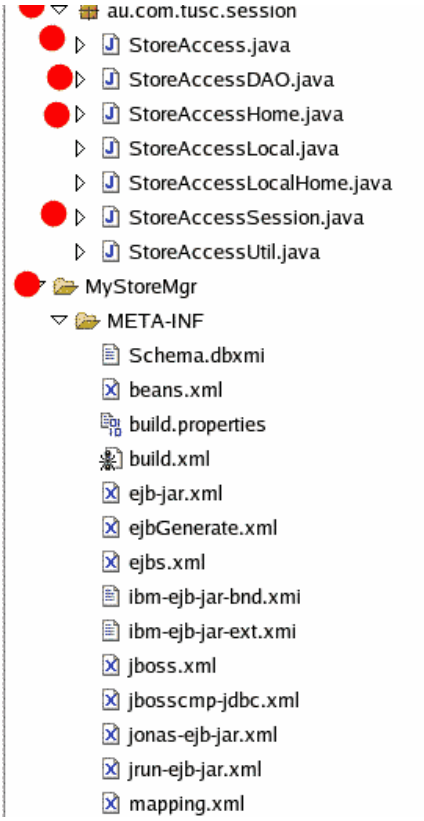


Expand MyStoreMgr node under MyStore Project in Package Explorer. Right click and a menu will pop up.

Go to Lomboz J2EE > Generate EJB Classes as shown in the figure below.



EJB interfaces and helper classes are generated under ejbsrc/au.com.tusc.session directory as shown in the figure at the right.



Seven files are generated.

StoreAccess is the remote object interface.

StoreAccessLocal is the local object interface.

StoreAccessSession extends our bean class named StoreAccesBean.

StoreAccessHome is the remote home interface.

StoreAccessLocalHome is the local home interface.

StoreAccessUtil is a helper class which has methods for accessing Home and LocalHome interface along with generating GUID.

StoreAccesDAO is the DAO interface which we will use to implement our StoreAccessDAOImpl under au.com.tusc.dao.

StoreAccessDAO is generated by this tag declared in StoreAccesBean shown below. If you don't declare this tag in that file it won't generate this interface.

```
@ejb.dao class=au.com.tusc.session.StoreAccessDAO
impl-class=au.com.tusc.dao.StoreAccessDAOImpl
```

Other files of interest which are generated are ejb-jar.xml and jboss.xml under MyStoreMgr/META-INF.

As shown in the figure on the right, a few descriptors are generated in the ejb-jar.xml file.

These descriptors are generated by the following tag declared in the StoreAccesBean file.

```
@ejb.bean name ="StoreAccess"
jndi-name="StoreAccessBean"
type="Stateless"
```

This tag is added by Lomboz's bean creation wizard.

ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans

<ejb-jar >

  <description><![CDATA[No Description.]]></description>
  <display-name>Generated by XDoclet</display-name>

  <enterprise-beans>

    <!-- Session Beans -->
    <session >
      <description><![CDATA[]]></description>

      <ejb-name>StoreAccess</ejb-name>

      <home>au.com.tusc.session.StoreAccessHome</home>
      <remote>au.com.tusc.session.StoreAccess</remote>
      <local-home>au.com.tusc.session.StoreAccessLocalHome</local-home>
      <local>au.com.tusc.session.StoreAccessLocal</local>
      <ejb-class>au.com.tusc.session.StoreAccessSession</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>

    </session>
```

This tag also generates the following descriptors in jboss.xml as shown in the code snippet below.

jboss.xml

MyStore/MyStoreMgr/META-INF/jboss.xml

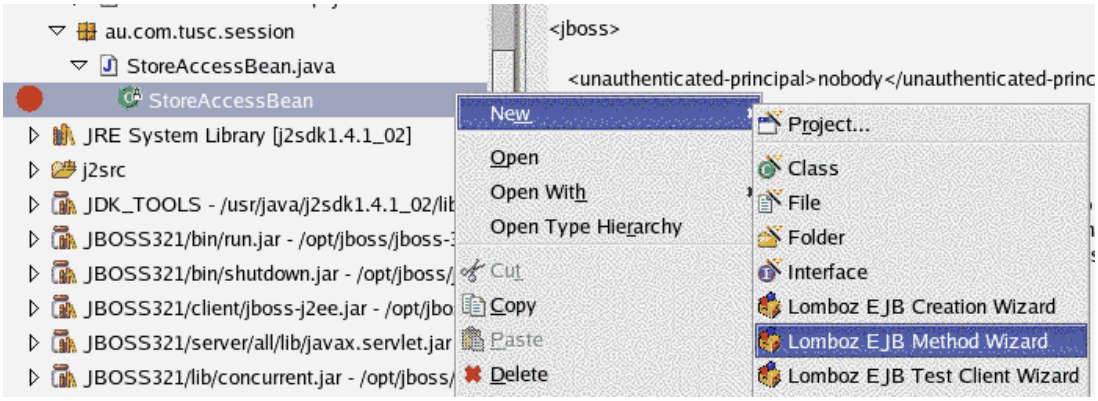
```
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS 3.0//EN" "http://www.jboss.org/j2ee/dtd/

<jboss>

  <unauthenticated-principal>nobody</unauthenticated-principal>

  <enterprise-beans>

   <!--
    To add beans that you have deployment descriptor info for, add
    a file to your XDoclet merge directory called jboss-beans.xml that contains
    the <session></session>, <entity></entity> and <message-driven></message-driven>
    markup for those beans.
   -->

   <session>
     <ejb-name>StoreAccess</ejb-name>
     <jndi-name>StoreAccessBean</jndi-name>
     <local-jndi-name>StoreAccessLocal</local-jndi-name>
   </session>

  </enterprise-beans>

  <resource-managers>
  </resource-managers>

</jboss>
```

So now we know which tags are responsible for generating classes, interfaces and descriptors.

Add Business Method :

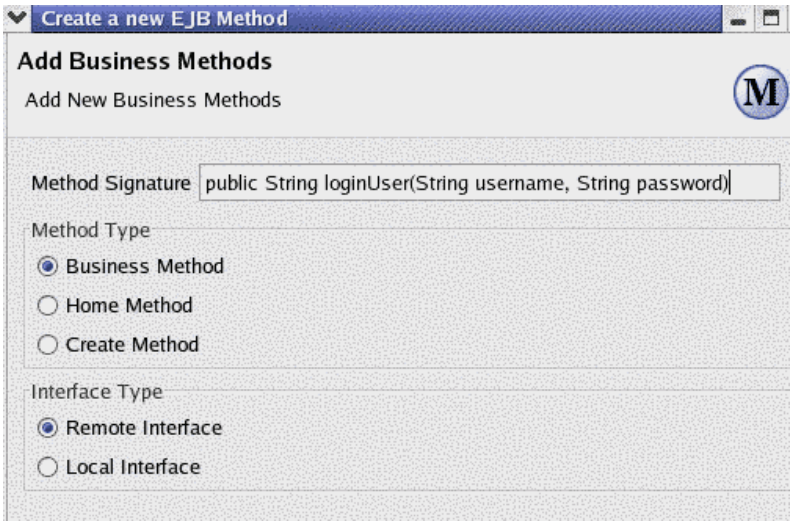
Next step is to add a business method in the bean.

Go to StoreAccesBean > Right click > Select New on pop up menu > Select Lomboz Ejb Method Wizard.



Add a business method with the following signature: public String loginUser (String username, String password).

Select Method Type as Business and Interface as Remote as shown in the figure below..



This wizard generates a loginUser method in our bean class, with the method level tag '@ejb.interface' shown below.

```
/**
 * @ejb.interface-method
 *  view-type="remote"
 *
 **/
public String loginUser(String username, String password ) {
```

This tag is responsible for generating this method in the Remote Interface (in this case it is StoreAccess which will be created once you generate your classes). This tag is covered later on in this chapter.

Now, This business method needs to invoke a method on the DAO, which will communicate with the database.

Therefore we add another tag on this method, so that a method with this signature is generated in DAO interface which we can implement in the DAOImpl class. Then this business method can invoke the method in DAOImpl class to get the desired result.

@dao.call name="loginUser"

So add this tag at the method level as shown in the figure at right.

Now generate your EJB classes again as shown in the steps we went through earlier.

Note: OK, OK! For reference these are the steps you have to follow.

Expand 'MyStoreMgr' node under 'MyStore' Project in Package Explorer.

Right click and a pop up menu will appear.

Go to Lomboz J2EE > Generate EJB Classes.

MyStore/src/au/com/tusc/session/StoreAccessBean.java

```
package au.com.tusc.session;

import javax.ejb.SessionBean;

/**
 * @ejb.bean name="StoreAccess"
 *  jndi-name="StoreAccessBean"
 *  type="Stateless"
 *
 * @ejb.dao class="au.com.tusc.session.StoreAccessDAO"
 *  impl-class="au.com.tusc.dao.StoreAccessDAOImpl"
 **/

public abstract class StoreAccessBean implements SessionBean {

/**
 * @ejb.interface-method
 *  view-type="remote"
 * @dao.call name="loginUser"
 **/
    public String loginUser(String username, String password){
        return null;
    }

}
```

After generating the classes, we look at first the generated DAO interface and then the generated Session Class.

In StoreAcessDAO two methods are generated.

1. init() by default.

2. loginUser(), generated by tag shown below.

@dao.call name="loginUser"

StoreAccessBean.j... | StoreAccessDAOI... | StoreAccessDA... | ejb-jar.xml

```
/*
 * Generated by XDoclet - Do not edit!
 */
package au.com.tusc.session;

/**
 * Data Access Object interface for StoreAccess.
 * @lomboz generated
 */
public interface StoreAccessDAO
{
    public void init();

    public java.lang.String loginUser(java.lang.String username,java.lang.String password) ;

}
```

Note: Please do not edit any class generated by Xdoclect.

In StoreAcessSession two methods of our interest are

1. getDAO() creates instance of DAOImpl calss.

2. loginUser(), calls loginUser method in DAOImpl class, which we have to implement.

Code snippet from 'StoreAccessSession'.

```
*StoreAccessSession.java ×
  }

  private static au.com.tusc.session.StoreAccessDAO dao = null;

protected au.com.tusc.session.StoreAccessDAO getDao()
  {
    if (dao != null) {
      return dao;
    } else {

      dao = (au.com.tusc.session.StoreAccessDAO) new au.com.tusc.dao.StoreAccessDA

      dao.init();
      return dao;
    }
  }

 public java.lang.String loginUser(java.lang.String username,java.lang.String password)
  {

     super.loginUser(username,password);

     return getDao().loginUser(username,password);

  }

}
```

Implement DAO Interface :

Now, we will implement methods in the StoreAccessDAOImpl class.

First import the following packages.

javax.naming.InitialContext;

javax.sql.DataSource;

java.sql.Connection;

java.sql.PreparedStatement;

java.sql.ResultSet;

java.sql.SQLException;

Change your class declaration so that StoreAccessDAOImpl implements StoreAccessDAO.

Add a field to store the JDBC resource factory reference.

private DataSource jdbcFactory;

In init() method, locate the reference "jdbc/DefaultDS" using the JNDI API, and store the reference in variable jdbcFactory.

Lookup string is "java:comp/env/jdbc/DefaultDS".

Code Snippet is shown in the figure on the right.

Now add the required code in loginUser().

```
StoreAccessDAOImpl.java ×  CustomerDAOImpl.java  CustomerBean.ja
package au.com.tusc.dao;
import au.com.tusc.session.StoreAccessDAO;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
/**
 * @author vishal
 *
 * To change the template for this generated type comment go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
public class StoreAccessDAOImpl  implements  StoreAccessDAO{

  private DataSource jdbcFactory;

  public void init() {

     System.out.println (" Entering StoreAccessDAOImpl.init() ");
     InitialContext c = null ;

     if  (this.jdbcFactory == null ) {

        try {
           c = new InitialContext() ;
           this.jdbcFactory= (DataSource) c.lookup("java:comp/env/jdbc/DefaultDS");
        }catch (Exception e) {
           System.out.println ("Error in StoreAccessDAOImpl.init() ");
        }
     }
     System.out.println (" Leaving StoreAccesDAOImpl.init() ");
  }
```

In method loginUser(), first get the connection to the database using the jdbcFactory.

Create a SQL statement which searches for userid in the table StoreAccess where userid and password is provided for each user.

Return userid if successful, else raise SQLException.

Code snippet is shown in the figure on the right.

Go back to your loginUser method in StoreAccessBean class.

```
public String loginUser (String username, String password) {

   System.out.println (" Entering StoreAccessDAOImpl.loginUser()  ");
   Connection conn = null;
   PreparedStatement ps = null;
   ResultSet rs = null;
   String userID = null;
   try {
      conn = jdbcFactory.getConnection();
      String queryString = "select userid from storeaccess where username = ? and password = ?";
      ps = conn.prepareStatement(queryString);
      ps.setString (1, username);
      ps.setString (2, password);
      rs = ps.executeQuery();
      boolean result =rs .next();
      if ( result) {
         userID = rs.getString("userid");
         System.out.println (" Userid is " +  userID );
       }
   }
   catch (SQLException e) {
       e.printStackTrace();
      System.out.println("Inside StoreAccessDAOImpl.loginUser() " +  e);
   }
   finally {
      try {
         rs.close();
         ps.close();
         conn.close();
      }
      catch(Exception e ) {
      }
   }
   System.out.println (" Leaving StoreAccessDAOImpl.loginUser()  " );
   return userID;
```

In StoreAcessBean class under the loginUser method just add some debug statements, as shown below in this code snippet.

```
**/
 public String loginUser(String username, String password ) {

    System.out.println("Entering StoreAccesBean");
    System.out.println("Leaving StoreAccesBean");
    return null;
 }
```

Note : We don't have to call the loginUser method in StoreAccessDAOImpl, as it being invoked by the loginUser method in StoreAccessSession class which inherits StoreAccessBean class, that is the StoreAccessSession class has overridden this method.

Code snippet from StoreAccessSession shown below.

```
public java.lang.String loginUser(java.lang.String username,java.lang.String password)
{

super.loginUser(username,password);

return getDao().loginUser(username,password);

}
```

Add Callback Methods :

Now, add callback methods to complete this bean as shown below.

1. setSessionContext.
2. UnsetSessionContext.

Note : These callback methods are invoked by the EJB container.

Add a field to store sessionContext.

protected SessionContext ctx;

Add method setSessionContext with sessionContext as parameter and assign that to the sessionContext variable as shown below in the code snippet.

```
/**
 * Sets the session context
 * @param javax.ejb.SessionContext the new ctx value
 *
 **/
public void setSessionContext(javax.ejb.SessionContext ctx)
    this.ctx = ctx;
}


/**
 * Unsets the session context
 *
 *
 **/
public void unsetSessionContext()  {
    this.ctx = null;
}
```

Similarly add method unsetSessionContext, assign context variable to null as shown above.

Note : StoreAccessSession class inherits the StoreAccessBean abstract class and implements SessionBean, which will override all methods of interface SessionBean. So after finishing the methods in the bean class, generate your EJB classes again. SessionContext methods will be overridden as shown in figure below. Code snippet from StoreAccessSession shown below.

```
public void setSessionContext(javax.ejb.SessionContext ctx)
{
super.setSessionContext(ctx);
}

public void unsetSessionContext()
{
super.unsetSessionContext();
}
```

Now let's look at the generated Home and Remote interfaces.

In the case of the Remote interface all business methods declared in the bean are also generated with the same signature. This is due to the class level tag declared in the StoreAccess Bean as discussed above after adding business methods. Code snippet for tag is shown below.

```
/**
* @ejb.interface-method
 view-type="remote"
"
**/
public String loginUser(String username, String password ) {
```

So, loginUser is generated in a Remote Interface called StoreAccess as shown below because of this tag.

```
package au.com.tusc.session;

/**
 * Remote interface for StoreAccess.
 * @lomboz generated
 */
public interface StoreAccess
   extends javax.ejb.EJBObject
{

public java.lang.String loginUser( java.lang.String username,java.lang.String password
    throws java.rmi.RemoteException;

}
```

In the case of the Home Interface only one method is created named 'create', which is generated by default because of the <homeinterface/> tag in ejbGenerate.xml as shown below.

```
/*
 * Generated by XDoclet - Do not edit!
 */
package au.com.tusc.session;

/**
 * Home interface for StoreAccess.
 * @lomboz generated
 */
public interface StoreAccessHome
   extends javax.ejb.EJBHome
{
   public static final String COMP_NAME="java:comp/env/ejb/StoreAccess";
   public static final String JNDI_NAME="StoreAccessBean";

   public au.com.tusc.session.StoreAccess create()
      throws javax.ejb.CreateException,java.rmi.RemoteException;

}
```

Also, other then that, it has JNDI_NAME and COMP_NAME (which is the logical name to lookup the component) is also generated, these are generated because of this tag declared at class level in 'StoreAccessBean' class shown below in figure.

```
package au.com.tusc.session;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;



/**
 * @ejb.bean name="StoreAccess"
 * jndi-name="StoreAccessBean"
 * type="Stateless"
```

Note : For further options associated with these tags please refer to the 'ejbdoclet' documentation in Xdoclet.

Now, all the aspects are pretty much covered, and our bean's functionality is complete. Now for the deployment descriptors..

Deploy Bean :

In order to deploy our bean we have to declare a few tags in the StoreAccessBean class as shown below in the code snippet.

```
/**
 * @ejb.bean name="StoreAccess"
 * jndi-name="StoreAccessBean"
 * type="Stateless"
 *
 * @ejb.dao class="au.com.tusc.session.StoreAccessDAO"
 * impl-class="au.com.tusc.dao.StoreAccessDAOImpl"
 *
 * @ejb.resource-ref res-ref-name="jdbc/DefaultDS"
 *     res-type="javax.sql.Datasource"
 *     res-auth="Container"
 *
 * @jboss.resource-ref  res-ref-name="jdbc/DefaultDS"
 *     jndi-name="java:/DefaultDS"
 *
 **/
public abstract class StoreAccessBean implements SessionBean {
```

Add the tag shown below in at the class level (at the top).

```
@ejb.resource-ref res-ref-name="jdbc/DefaultDS"
res-type="javax.sql.Datasource"
res-auth="Container"
```

This tag will generate deployment descriptors in ejb-jar.xml, as the bean has to know which datasource you are going to connect to, what is its type, etc. This will generate these descriptors as shown in code snippet below.

```
ejb-jar.xml   StoreAccessSession.java   CustomerBMP.java
    <ejb-name>StoreAccess</ejb-name>

    <home>au.com.tusc.session.StoreAccessHome</home>
    <remote>au.com.tusc.session.StoreAccess</remote>
    <local-home>au.com.tusc.session.StoreAccessLocalHome</local-home>
    <local>au.com.tusc.session.StoreAccessLocal</local>
    <ejb-class>au.com.tusc.session.StoreAccessSession</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>

    <resource-ref >
      <res-ref-name>jdbc/DefaultDS</res-ref-name>
      <res-type>javax.sql.Datasource</res-type>
      <res-auth>Container</res-auth>
    </resource-ref>

  </session>
```

Add the tag shown below in StoreAccessBean at the class level (at the top).

@jboss.resource-ref res-ref-name="jdbc/DefaultDS" jndi-name="java:/DefaultDS"

This tag will generate deployment descriptors in jboss.xml, as the application server has to know with what jndi-name datasource it has been registered with. This will generate these descriptors as shown in the code snippet below.

```
<session>
  <ejb-name>StoreAccess</ejb-name>
  <jndi-name>StoreAccessBean</jndi-name>
  <local-jndi-name>StoreAccessLocal</local-jndi-name>
  <resource-ref>
    <res-ref-name>jdbc/DefaultDS</res-ref-name>
    <jndi-name>java:/DefaultDS</jndi-name>
  </resource-ref>
</session>
```

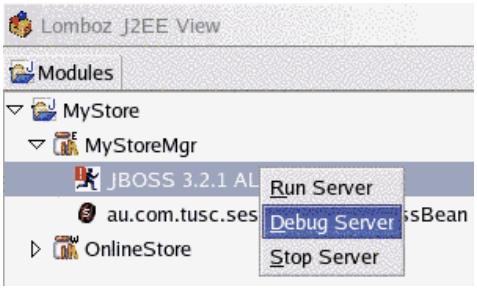
Now, everything is complete, and it's time to deploy the bean.

First, regenerate your EJB classes as shown in the steps above for the final time.

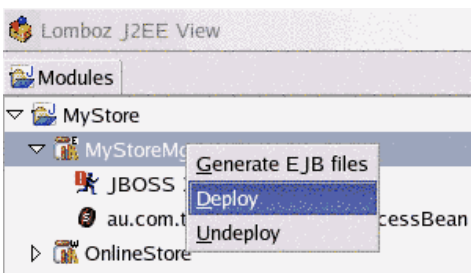
Note : We have regenerated the classes again and again, in order to explain every step and its result. Once you are familiar with these steps you will need much fewer of these iterations. Either way, it doesn't matter, as your implementation always remains untouched by this process.

Go to Lomboz J2EE View > expand node MyStore > expand MyStoreMgr > select 'Jboss 3.2.1 ALL' .

Right click > select Debug Sever on the pop up menu as shown in figure below.



Go to MyStoreMgr node in LombozJ2EE view > right click > select Deploy on the pop up menu as shown in the figure below.



And now wait for your deployment result.

If everything goes fine, you will have this message under your console as shown in the figure below.

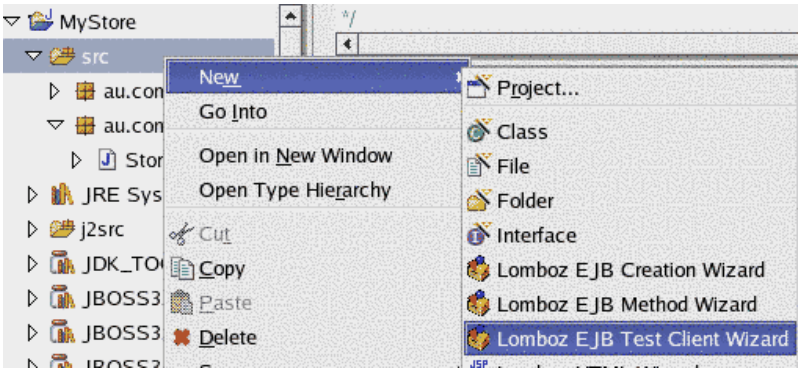
```
14:39:22,080 INFO  [jbossweb] Registered jboss.web:Jetty=0,JBossWebApplicationContext=4,context=/web-console,name=SystemFolder
14:39:22,120 INFO  [jbossweb] Registered jboss.web:Jetty=0,JBossWebApplicationContext=4,context=/web-console,name=UCLs
14:39:22,140 INFO  [jbossweb] Registered jboss.web:Jetty=0,JBossWebApplicationContext=4,context=/web-console,name=HTTP Invocation
14:39:22,195 INFO  [jbossweb] Registered jboss.web:Jetty=0,JBossWebApplicationContext=4,context=/web-console,HashSessionManager=0
14:39:22,196 INFO  [jbossweb] successfully deployed file:/opt/jboss/jboss-3.2.1/server/all/tmp/deploy/server/all/deploy/management/web-cons
14:39:22,293 INFO  [MainDeployer] Deployed package: file:/opt/jboss/jboss-3.2.1/server/all/deploy/management/web-console.war
14:39:22,315 INFO  [URLDeploymentScanner] Started
14:39:22,482 INFO  [MainDeployer] Deployed package: file:/opt/jboss/jboss-3.2.1/server/all/conf/jboss-service.xml
14:39:22,489 INFO  [Server] JBoss (MX MicroKernel) [3.2.1 (build: CVSTag=JBoss_3_2_1 date=200305041533)] Started in 1m:19s:911ms
14:41:08,087 INFO  [MainDeployer] Starting deployment of package: file:/opt/jboss/jboss-3.2.1/server/all/deploy/MyStoreMgr.jar
14:41:09,460 INFO  [EjbModule] Creating
14:41:09,488 INFO  [EjbModule] Deploying StoreAccess
14:41:09,535 INFO  [StatelessSessionContainer] Creating
14:41:09,592 INFO  [StatelessSessionInstancePool] Creating
14:41:09,592 INFO  [StatelessSessionInstancePool] Created
14:41:09,599 INFO  [StatelessSessionContainer] Created
14:41:09,600 INFO  [EjbModule] Created
14:41:09,602 INFO  [EjbModule] Starting
14:41:09,602 INFO  [StatelessSessionContainer] Starting
14:41:10,076 INFO  [StatelessSessionInstancePool] Starting
14:41:10,077 INFO  [StatelessSessionInstancePool] Started
14:41:10,078 INFO  [StatelessSessionContainer] Started
14:41:10,078 INFO  [EjbModule] Started
14:41:10,079 INFO  [EJBDeployer] Deployed: file:/opt/jboss/jboss-3.2.1/server/all/deploy/MyStoreMgr.jar
14:41:10,121 INFO  [MainDeployer] Deployed package: file:/opt/jboss/jboss-3.2.1/server/all/deploy/MyStoreMgr.jar
```

So, now our bean is deployed successfully, let's create our test client, which will invoke the loginUser method on 'StoreAccessBean'.

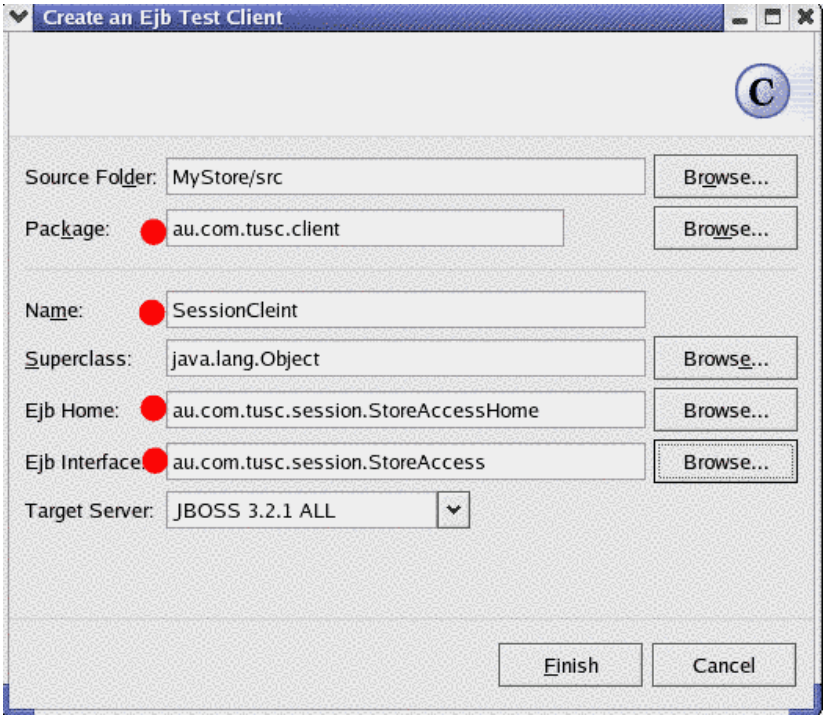
Create your Test Client :

Go to Project MytStore node > select src node > right click.

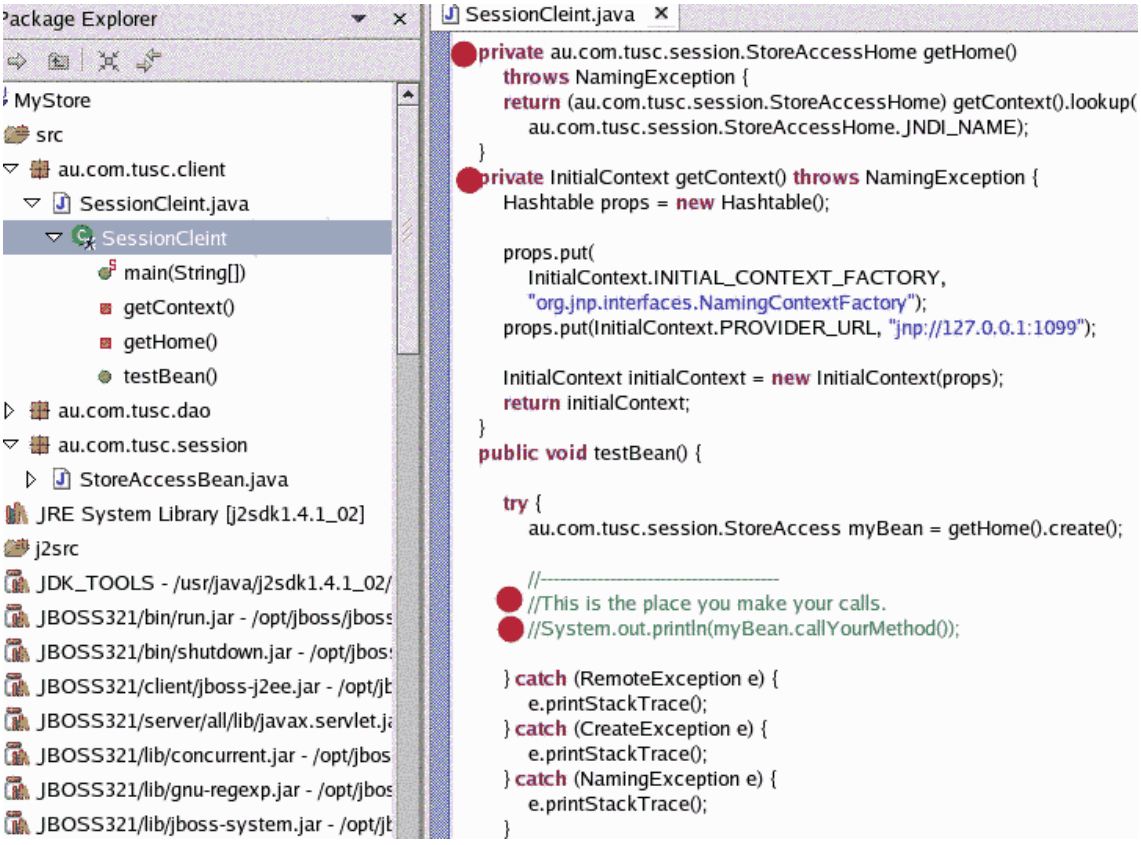
Select New on pop up menu > select Lomboz EJB Test Client Wizard as shown in the figure below.



Select package name au.com.tusc.client, name as SessionClient and select Ejb Home as au.com.tusc.session.StoreAccessHome and Ejb Interface as au.com.tusc.session.StoreAccess as shown in the figure below.



This will generate the required methods for you in your SessionClient class and you have to just invoke the loginUser method on the bean as shown below.



Now the last step is to write code in your client.

So add these lines under the testBean method as shown in figure below**.**

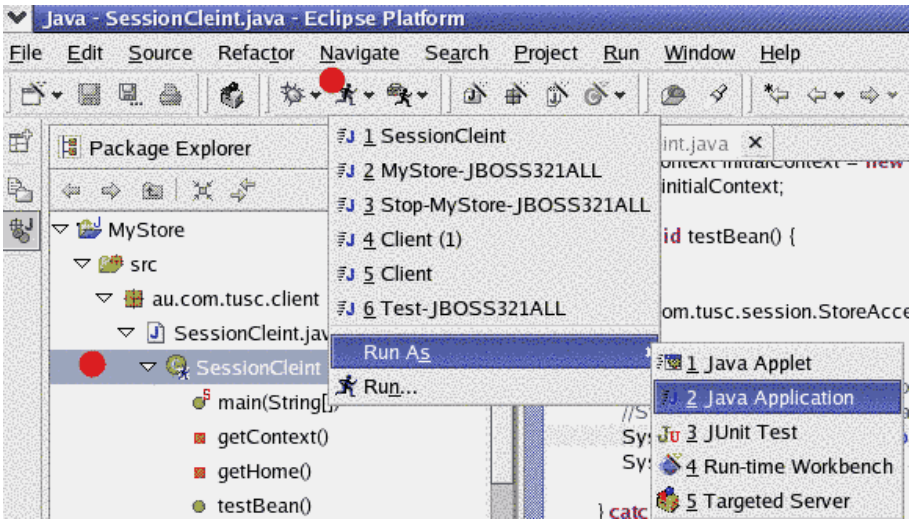
```
System.out.println("Request from client : ");
System.out.println("Reply from Server: Your userid is " +
myBean.loginUser("ANDY","PASSWD"));
```

```
public void testBean() {

    try {
        au.com.tusc.session.StoreAccess myBean = getHome().create();

        //-----------------------------------
        //This is the place you make your calls.
        //System.out.println(myBean.callYourMethod());
        System.out.println("Request from client : ");
        System.out.println("Reply from Server: Your userid is " + myBean.loginUser("ANDY","PASSWD"));

    } catch (RemoteException e) {
        e.printStackTrace();
    } catch (CreateException e) {
        e.printStackTrace();
    } catch (NamingException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    SessionCleint test = new SessionCleint();
    test.testBean();

}
```

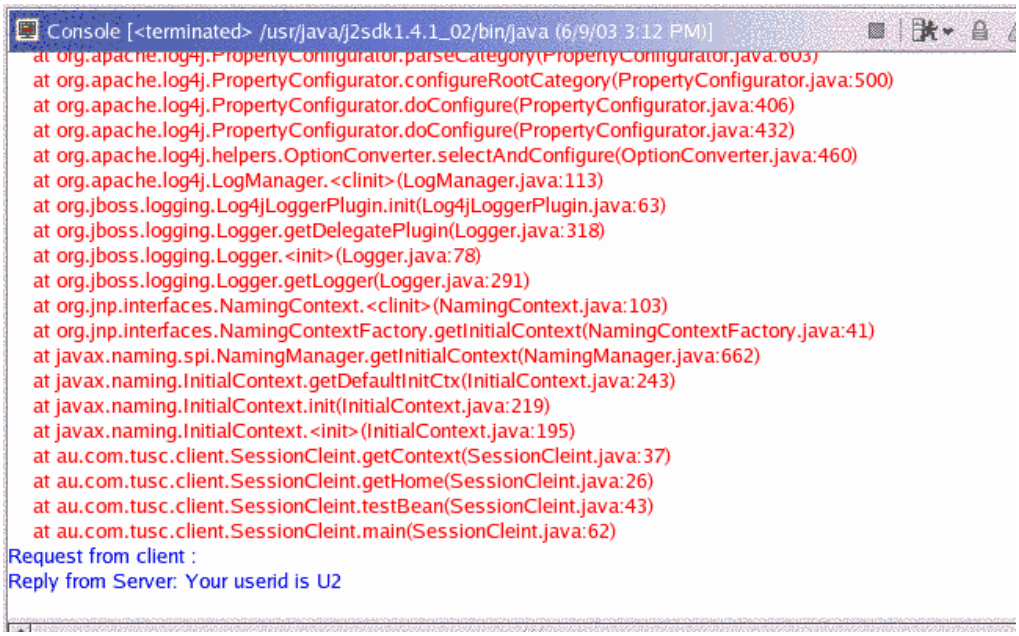
Test your Client :

Now, in order to test your client, Select SessionClient node > Go at top level menu and select the icon with the 'Running Man'.

On that select 'Run as' > select 'Java Application', as shown below.



Now under your console, if you get your reply for 'ANDY' as 'U2', then your call is successful as shown below.



Note : So our Stateless Session Bean is deployed and tested successfully and from now onwards you should be comfortable using Lomboz. In future we will not go into the detail of the steps for using Lomboz and will concentrate more on other aspects of beans.

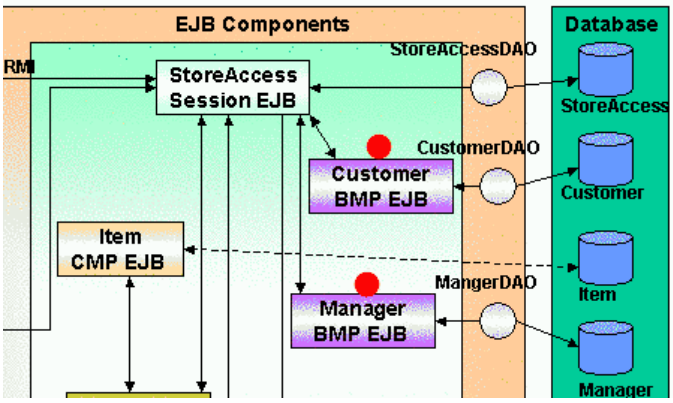
Prev TOC Next

Tutorial for building J2EE Applications using JBOSS and ECLIPSE

Chapter 5.

Creating a BMP Entity Bean

This chapter describes how to create a Bean Managed Persistence (BMP) EJB component. We will create two BMP beans, Customer and Manager, as shown below. The Customer bean will be responsible for storing the details of customers of MyStore. The Manager bean stores details of the manager of MyStore. Both beans communicate with their respective tables in the database using Data Access Objects (DAOs) named CustomerDAO and ManagerDAO respectively.



All customers have been assigned a unique customerID for housekeeping purposes in MyStore in addition to their username for accessing the services of MyStore. Similarly the Manager of MyStore has been assigned a unique ManagerID.

Note : It is the usual practice to access business methods of BMP beans via a session bean, which encapsulates business logic and acts as an interface to further EJB components. In this case Customer and Manager are accessible via by StoreAccess.

This approach comes from a pattern called a Session Facade, whereby enterprise beans encapsulate business logic and business data and expose their interfaces. The session bean acts as a facade to encapsulate the complexity of interactions with the lower-level beans. The session facade is responsible for managing business objects and provides a uniform business service abstraction to presentation layer clients, thereby hiding the business object implementation in the lower-level beans.

This tutorial uses that pattern for business tier implementation.

Tasks :

1. Create a BMP bean named Customer under package au.com.tusc.bmp.
2. Create a DAO class named CustomerDAOImpl under package au.com.tusc.dao.
3. Add all attributes/properties to the CustomerBean, with getter and setter methods for each of the attributes.
4. Add a finder method named ejbFindByPrimaryKey with the signature

   public CustomerPK ejbFindByPrimaryKey (CustomerPK pk) throws FinderException
5. Add a finder method named ejbFindByUserID with the signature

   public CustomerPK ejbFindByUserID (String userID) throws FinderException
6. Add a business method named getCustomerData with the signature

   public CustomerData getCustomerData()
7. Implement required methods in the CustomerDAOImpl class.

8. Deploy the Customer Bean.
9. Add a create method to the StoreAccess Bean.

   public void ejbCreate() throws javax.ejb.CreateException
10. Add a business method to the StoreAccess Bean.

    public CustomerData getCustomerData(String userID)
11. Create a test client named SessionBMPClient under package au.com.tusc.client.
12. Run your client and test the bean.

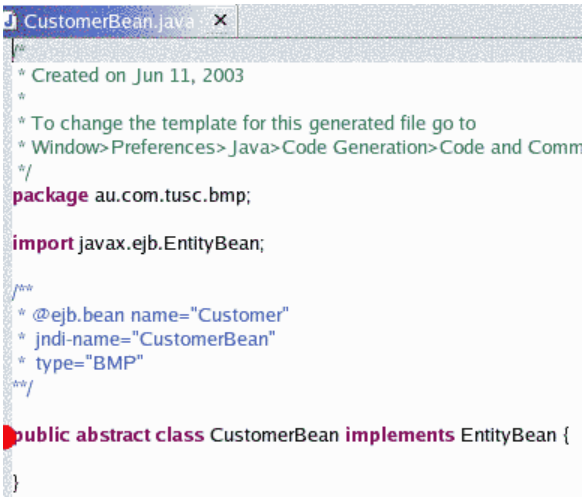
Create the Customer BMP Entity Bean :

Go To Package Explorer > Expand Mystore (project) node > select src, right click and a menu will pop up.

On the pop up menu > New > Lomboz EJB Creation Wizard.

Enter package name au.com.tusc.bmp, bean name Customer and select bean type as Bean Manged Entity > Finish.

This will create a package named au.com.tusc.bmp under src and CustomerBean under that package as shown below.



Note: It will generate the bean name, jndi-name and type of bean in file. Also, the word 'Bean' is appended to the name of the file.

As we can see from the figure above it has created a class level tag **@ejb.bean**, which has assigned the bean type, its name and its JNDI name which will be generated in the Home interface. This tag will also generate deployment descriptors in ejb-jar.xml and jboss.xml file once you generate your EJB classes.

Now we are going to generate all the interfaces including Home, Remote, DAO and other helper classes. We don't need to specify any tags in ejbGenerate.xml as we have already set up that for the MyStoreMgr EJB module in chapter 3.

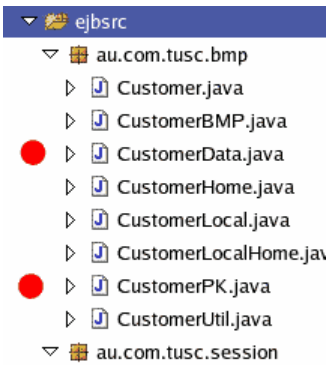
Go to node CustomerBean under au.com.tusc.bmp > LombozJ2EE > Add EJB to Module > Select MyStoreMgr > Ok.

Go to MyStoreMgr node > LombozJ2EE > Generate EJB classes.

Note: All these steps are covered in previous chapters (chapter 3 and 1) so please refer to those sections if you need further details.

Now let's see what files are generated by Xdoclet.

As shown below there are two new files named CustomerData and CustomerPK in addition to what we had for our session beans. We also have a CustomerBMP which extends our CustomerBean class. The remaining files should be familiar from our work with session beans.



As discussed in chapter 3 regading the various ejbDoclet tags used in ejbGenerate.xml, to generate these classes, CustomerData and CutomerPk are generated by following tags as shown below in this snippet from ejbGenerate.xml:

The relevant tags are <dataobject/> and <entitypk/>.

```
<fileset dir="../src" defaultexcludes="yes">
  <patternset includesfile="META-INF/ejbs.x
</fileset>

<dataobject/>
<valueobject/>
<utilobject cacheHomes="true" includeGUI
<dao pattern="{0}" destDir="${ejbsrc.dir}" />

<remoteinterface/>
<localinterface/>
<homeinterface />
<localhomeinterface/>

<entitypk/>
<entitycmp/>
<entitybmp/>
```

Note: There is no CustomerDAO class, as we haven't generated that file by specifying the dao tag in the CustomerBean class.

Create Customer's DAO Interface :

Since we are going to use a DAO to access database for this bean, we have to create a DAOImpl class to provide an implementation for the generated DAO interface.

Go to src > package au.com.tusc.dao > Add a class CustomerDAOImpl in that package.

Now go to your bean class and declare this tag at class level (ie. at the top) as shown below to generate the DAO interface.

```
@ejb.dao class="au.com.tusc.bmp.CustomerDAO"
  impl-class="au.com.tusc.dao.CustomerDAOImpl"
```

```
package au.com.tusc.bmp;

import javax.ejb.EntityBean;

/**
 * @ejb.bean name="Customer"
 * jndi-name="CustomerBean"
 * type="BMP"
 *
 * @ejb.dao class="au.com.tusc.bmp.CustomerDAO"
 * impl-class="au.com.tusc.dao.CustomerDAOImpl"
 *
 **/

public abstract class CustomerBean implements EntityBean {

}
```

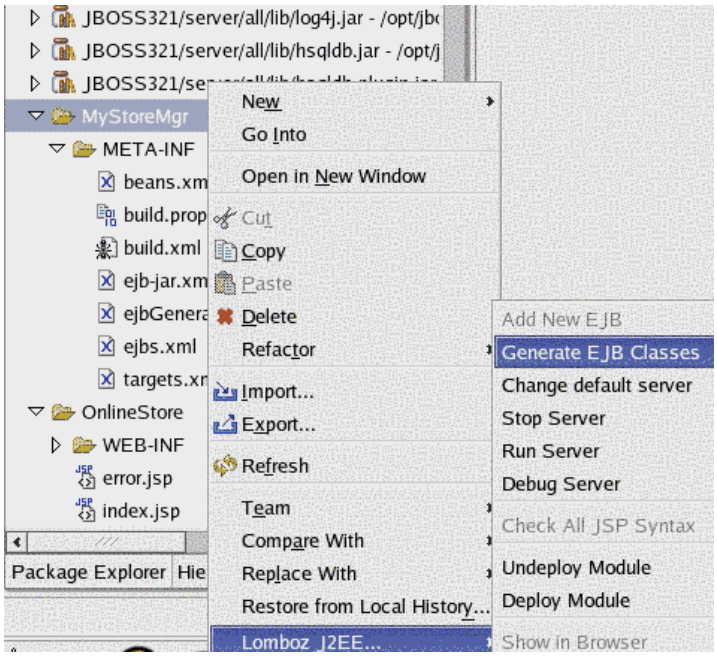
Regenerate your classes and check that your DAO interface is generated.

Note : For reference the steps are as follows:

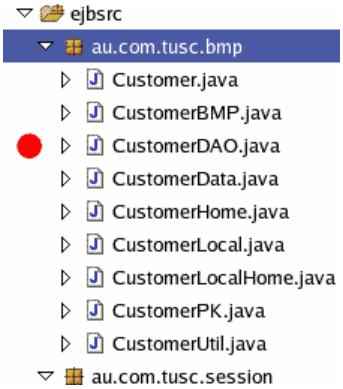
Expand MyStoreMgr node under MyStore Project in Package Explorer.

Right click and a pop up menu will appear.

Go to Lomboz J2EE > Generate EJB Classes as shown on the right.



As shown below, our DAO interface is generated.



If we look at the generated DAO class as shown below, it has four more methods than StoreAccess (which was a Stateless Session Bean) did when we first generated it (ie. prior to adding the method level @dao:call tag).

Note : It's worth mentioning here that BMP Entity beans don't need the @dao:call tag for generating DAO interface methods.

```
package au.com.tusc.bmp;

/**
 * Data Access Object interface for Customer.
 * @lomboz generated
 */
public interface CustomerDAO
{
   public void init();

public void load(au.com.tusc.bmp.CustomerPK pk, au.com.tusc.bmp.CustomerBean ejb) throws javax.ejb.EJBException;
public void store(au.com.tusc.bmp.CustomerBean ejb) throws javax.ejb.EJBException;
public void remove(au.com.tusc.bmp.CustomerPK pk) throws javax.ejb.RemoveException, javax.ejb.EJBException;

public au.com.tusc.bmp.CustomerPK create(au.com.tusc.bmp.CustomerBean ejb) throws javax.ejb.CreateException, javax.ejb.EJBException;

}
```

All these methods are generated by the tag described earlier for DAO interface generation, also shown below.

```
 *
 * @ejb.dao class="au.com.tusc.bmp.CustomerDAO"
 * impl-class="au.com.tusc.dao.CustomerDAOImpl"
 *
```

Also, let's look at the descriptors generated up until this step, in ejb-jar.xml under MyStoreMgr/META-INF.

These descriptors are generated by the following tag in CustomerBean.

```
@ejb.bean name="Customer"
jndi-name="CustomerBean"
type="BMP"
```

This tag generates descriptors in jboss.xml as well, which will be covered later on.

```
<!-- Entity Beans -->
<entity >
  <description><![CDATA[]]></description>

  <ejb-name>Customer</ejb-name>

  <home>au.com.tusc.bmp.CustomerHome</home>
  <remote>au.com.tusc.bmp.Customer</remote>
  <local-home>au.com.tusc.bmp.CustomerLocalHome</local-home>
  <local>au.com.tusc.bmp.CustomerLocal</local>

  <ejb-class>au.com.tusc.bmp.CustomerBMP</ejb-class>
  <persistence-type>Bean</persistence-type>
  <prim-key-class>au.com.tusc.bmp.CustomerPK</prim-key-class>
  <reentrant>False</reentrant>

</entity>
```

The next step is to add attributes/properties to our Customer Bean, which will be accessible to clients through the remote interface using getter and setter methods. These attributes are mapped to corresponding columns in the relevant table in the database.

In order to add these properties/attributes , define all attributes (as private to encapsulate) and their corresponding accessors/mutators (getter and setter methods). Each accessor (getter) method will have two tags or three in the case of a primary key. The mutator (setter) method will have only one tag declared on it.

Add these tags for attributes/properties (in Bean terms) as shown below.

```
/**

        * Returns the customerID
        * @return the customerID
        *
        * @ejb.persistence
        * @ejb.pk-field
        * @ejb.interface-method
        */
        public  java.lang.String getCustomerID() {
                return customerID ;
        }

        /**
          * Sets the customerID
          * @param java.lang.String the new customerID value
          *
          * @ejb.interface-method
          */
        public void setCustomerID (java.lang.String customerID)
                this.customerID = customerID;

}
```

Code snippet from CustomerBean is shown in figure at right.

```
public abstract class CustomerBean implements EntityBean {

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

    /**
     * Returns the customerID
     * @return the customerID
     * @ejb.persistence
     * @ejb.pk-field
     * @ejb.interface-method
     */
     public  java.lang.String getCustomerID(){
       return customerID;
     }

    /**
     * Sets the customerID
     * @param java.lang.String the new customerID value
     * @ejb.interface-method
     *
     */
    public  void setCustomerID (java.lang.String customerID) {
       this.customerID = customerID ;
    }
```

Now, analyzing these tags,

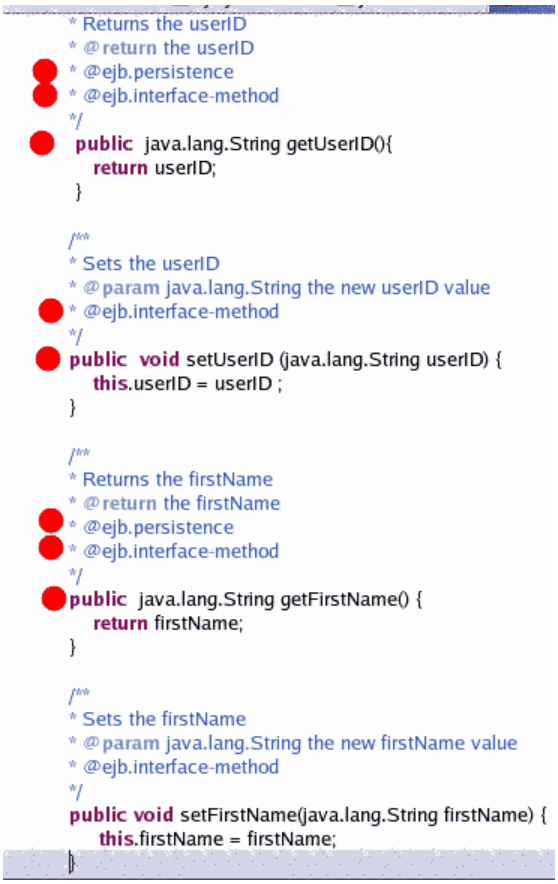
1. **@ejb.persistence** specifies this as a persistent attribute. All accessor and mutator methods will be overridden in generated BMP class, in this case it is CustomerBMP, and all mutator methods will have a dirty flag in them, as persistence is controlled by ejbLoad() and ejbStore().
2. **@ejb.pk-field** specifies that this attribute is mapped to a primary key in database and and is assigned as a primary key in the PrimaryKey class, in this case the CustomerPK class**.**
3. **@ejb.interface** generates these methods in the Remote interface.

Now, in the case of mutator methods (that is 'setCustomerID') the only tag required is @ejb.interface-method for generating this method in the remote interface.

Similarly, add methods and their tags for rest of the persistent fields. There will be no @ejb.pk-field tag as customerID is the primary key.

Note : In the case of a composite primary key you have to specify @ejb.pk-field tags on the other attributes/properties which make up the composite key.

Similarly, Add tags and methods for rest of the persistent fields, which in this case are, firstName, lastName, Address, phone and shareholderStatus as shown in this code snippet from CustomerBean.



```
    * Returns the userID
    * @return the userID
    * @ejb.persistence
    * @ejb.interface-method
    */
     public  java.lang.String getUserID(){
       return userID;
     }

    /**
    * Sets the userID
    * @param java.lang.String the new userID value
    * @ejb.interface-method
    */
    public  void setUserID (java.lang.String userID) {
       this.userID = userID ;
    }

    /**
    * Returns the firstName
    * @return the firstName
    * @ejb.persistence
    * @ejb.interface-method
    */
    public  java.lang.String getFirstName() {
       return firstName;
    }

    /**
    * Sets the firstName
    * @param java.lang.String the new firstName value
    * @ejb.interface-method
    */
    public void setFirstName(java.lang.String firstName) {
       this.firstName = firstName;
    }
```

Generate EJB classes and examine what methods are generated in the various classes, particularly in CustomerBMP and CustomerData.

Add Finder Methods :

Now let's add a finder method to our bean class.

Add a method with this signature:

public CustomerPK ejbFindByPrimaryKey (CustomerPK pk) throws FinderException

Put some debug statements in it and return null as shown below.

```
/**
 *   Finds the primaryKey
 *  @param CustomerPK pk value
 *
 *
 **/
public au.com.tusc.bmp.CustomerPK ejbFindByPrimaryKey (au.com.tusc.bmp.CustomerPK pk) throws FinderException{
    System.out.println ("Entering CustomerBean.ejbFindByPrimatyKey ");
    System.out.println ("Leaving CustomerBean.ejbFindByPrimaryKey ");
     return null;
 }
```

Now when we generate our EJB classes this method will be overridden in CustomerBMP. Also this method will call a corresponding method from CustomerDAO interface as shown below in this code snippet from CustomerBMP.

```
public au.com.tusc.bmp.CustomerPK ejbFindByPrimaryKey(au.com.tusc.bmp.CustomerPK pk) throws javax.ejb.FinderExceptio
{
  super.ejbFindByPrimaryKey(pk);

  return getDao().findByPrimaryKey(pk);
}

public void ejbLoad()
{
  getDao().load((au.com.tusc.bmp.CustomerPK) ctx.getPrimaryKey(), this);
  makeClean();
}

public void ejbStore()
{
  if (isModified())
  {
    getDao().store((au.com.tusc.bmp.CustomerBean) this);
    makeClean();
  }
}
```

This will also create methods in the Home interface and DAO interfaces as shown below.

```
package au.com.tusc.bmp;

/**
 * Home interface for Customer.
 * @lomboz generated
 */
public interface CustomerHome
  extends javax.ejb.EJBHome
{
  public static final String COMP_NAME="java:comp/env/ejb/Customer";
  public static final String JNDI_NAME="CustomerBean";

  public au.com.tusc.bmp.Customer findByPrimaryKey(au.com.tusc.bmp.CustomerPK pk)
    throws javax.ejb.FinderException,java.rmi.RemoteException;

}
```

```
package au.com.tusc.bmp;

/**
 * Data Access Object interface for Customer.
 * @lomboz generated
 */
public interface CustomerDAO
{
  public void init();

  public void load(au.com.tusc.bmp.CustomerPK pk, au.com.tusc.bmp.CustomerBean ejb) throws javax.ejb.EJBException;
  public void store(au.com.tusc.bmp.CustomerBean ejb) throws javax.ejb.EJBException;
  public void remove(au.com.tusc.bmp.CustomerPK pk) throws javax.ejb.RemoveException, javax.ejb.EJBException;

  public au.com.tusc.bmp.CustomerPK create(au.com.tusc.bmp.CustomerBean ejb) throws javax.ejb.CreateException, javax.ejb.EJBException;

  public au.com.tusc.bmp.CustomerPK findByPrimaryKey(au.com.tusc.bmp.CustomerPK pk) throws javax.ejb.FinderException;

}
```

Add another finder method to CustomerBean, with the signature

public CustomerPK ejbFindByUserID (String userID) throws FinderException

Put some debug statements in it and return null as shown below.

```
  /**
   * Finds the Primary Key
   *  @return CustomerPk object
   *
   *  @param String userID value
   *
   *
   **/
  public au.com.tusc.bmp.CustomerPK ejbFindByUserID (String userID) throws FinderException{
    System.out.println ("Entering CustomerBean.ejbFindByUserID ");
    System.out.println ("Leaving CustomerBean.ejbFindByUserID ");
    return null;
  }
```

Note : As stated in the EJB spec 12.8.1 all finder methods should return the Primary Key.

Again, regenerate your EJB classes, and there will be methods created in the CustomerHome interface, CustomerBMP and CustomerDAO, similar to those that were created for ejbFindByPrimaryKey.

A code snippet from the CustomerDAO class, after generating the EJB classes.

```
/
public interface CustomerDAO
{
   public void init();

   public void load(au.com.tusc.bmp.CustomerPK pk, au.com.tusc.bmp.CustomerBean ejb) throws javax.ejb.EJBException;
   public void store(au.com.tusc.bmp.CustomerBean ejb) throws javax.ejb.EJBException;
   public void remove(au.com.tusc.bmp.CustomerPK pk) throws javax.ejb.RemoveException, javax.ejb.EJBException;

   public au.com.tusc.bmp.CustomerPK create(au.com.tusc.bmp.CustomerBean ejb) throws javax.ejb.CreateException, javax.ejb.EJBException;

   public au.com.tusc.bmp.CustomerPK findByPrimaryKey(au.com.tusc.bmp.CustomerPK pk) throws javax.ejb.FinderException;

   public au.com.tusc.bmp.CustomerPK findByUserID(java.lang.String userID) throws javax.ejb.FinderException;

}
```

Add Business Methods :

Now, add a business method with the signature

public CustomerData getCustomerData() with Interface type as local.

Note : The steps to add a business method are covered in previous chapters (1 and 3), so please refer to them. Also, we have chosen the interface type to be local because these methods will be invoked in the same Java Virtual Machine. In this case they will be invoked by the stateless bean StoreAccess.

This will provide the details of an individual customer. Add some debug statements and return an instance of CustomerData as shown below in this code snippet from CustomerBean.

```
/**
 *  Returns the Customer Data Object
 * @return the CustomerData
 * @ejb.interface-method
 *  tview-type="local"
 **/
public CustomerData getCustomerData(){
        System.out.println ("Entering CustomerBean.getCustomerData() ");
        System.out.println ("Leaving CustomerBean.getCustomerData() ");

        return new CustomerData ( getCustomerID(), getUserID(), getFirstName(), getLastName(),
                        getAddress(), getPhone(), getShareholderStatus());
    }
```

Make sure you generate your EJB classes again before you start implementing Customer's DAO interface.

Implement Customer's DAO Interface :

Now let's implement our methods in CustomerDAOImpl class.

CustomerDOAImp class under package au.com.tusc.dao implements methods generated in CustomerDAO class under package au.com.tusc.bmp.

First import the following packages.

javax.naming.InitialContext;
javax.sql.DataSource;
java.sql.Connection;
java.sql.PreparedStatement;
java.sql.ResultSet;
java.sql.SQLException;

Change your class declaration so that CustomerDAOImpl implements CustomerDAO.

Add a field to store the JDBC resource factory reference.

private DataSource jdbcFactory;

In the init() method, locate the reference jdbc/DefaultDS using the JNDI API, and store the reference in variable jdbcFactory.

The lookup string is "java:comp/env/jdbc/DefaultDS".

Code snippet is shown below.

```
package au.com.tusc.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.ejb.EJBException;
import javax.ejb.FinderException;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import au.com.tusc.bmp.CustomerDAO;
import au.com.tusc.bmp.CustomerPK;

public class CustomerDAOImpl  implements CustomerDAO{

  private DataSource jdbcFactory;

    public CustomerDAOImpl () {
       super();
    }

    public void init() {
       System.out.println (" Entering CustomerDAOImpl.init() ");

       InitialContext c = null ;
       if  (this.jdbcFactory == null ){
          try {
             c = new InitialContext() ;
             this.jdbcFactory= (DataSource) c.lookup("java:comp/env/jdbc/DefaultDS");
          }catch (Exception e) {
             System.out.println ("Error in CustomerDAOImpl.init() ");
          }
       }
       System.out.println (" Leaving CustomerDAOImpl.init() ");
    }
```

In method load(), first get the connection to the database using field jdbcFactory. Create a SQL statement which searches for a record corresponding to customerid in table Customer, where customerid is the primary key. Code snippet is shown below.

```
  public void load(au.com.tusc.bmp.CustomerPK pk, au.com.tusc.bmp.CustomerBean ejb) throw
     System.out.println (" Entering CustomerDAOImpl.load()  ");
     Connection conn = null;
     PreparedStatement ps = null;
     ResultSet rs = null;
     try {
        conn = jdbcFactory.getConnection();
        String queryString = "select customerid, userid, firstname, lastname, address, phone,"+
                             "shareholder_stat from customer where customerid = ?";
        ps = conn.prepareStatement(queryString);
        ps.setString(1,pk.getCustomerID());
        rs = ps.executeQuery();
        System.out.println ("QueryString is " + queryString) ;
        if ( rs.next() )  {
           int count =1;
           ejb.setCustomerID((rs.getString(count++)).trim());
           ejb.setUserID((rs.getString(count++)).trim());
           ejb.setFirstName((rs.getString(count++)).trim());
           ejb.setLastName((rs.getString(count++)).trim());
           ejb.setAddress((rs.getString(count++)).trim());
           ejb.setPhone((rs.getString(count++)).trim());
           ejb.setShareholderStatus((rs.getString(count++)).trim());
        }
     }
     catch (SQLException e) {
        throw new EJBException("Row for id " + pk.customerID + " not found in database" + e);
     }
     finally  {
        try {
           rs.close();
           ps.close();
           conn.close();
        } catch (Exception e) {}
     }
     System.out.println (" Leaving CustomerDAOImpl.load() ") ;
  }
```

In method store(), first get the connection to database using field jdbcFactory. Create an SQL statement which updates a record corresponding to customerid in table Customer, where customerid is the primary key. Code snippet is shown below.

```
public void store(au.com.tusc.bmp.CustomerBean ejb) throws javax.ejb.EJBException {

   System.out.println (" Entering CustomerDAOImpl.store()  ");
   Connection conn = null;
   PreparedStatement ps = null;
   ResultSet rs = null;
   try {

      conn = jdbcFactory.getConnection();
      String updateString = "update customer set userid = ?, firstname = ?,lastname = ?," +"" +
                            "address = ?, phone = ?, shareholder_stat = ? where customerid = ?";
      ps = conn.prepareStatement(updateString);
      ps.setString(1,ejb.getUserID().trim());
      ps.setString(2,ejb.getFirstName().trim());
      ps.setString(3,ejb.getLastName().trim());
      ps.setString(4,ejb.getAddress().trim());
      ps.setString(5,ejb.getPhone().trim());
      ps.setString(6,ejb.getShareholderStatus().trim());
      int count  = ps.executeUpdate();
      System.out.println ("Update String is " + updateString) ;
   }
   catch (Exception e ) {
      e.printStackTrace();
   }
   finally  {
      try {
         ps.close();
         rs.close();
         conn.close();
      }catch (Exception e) {
      }
   }
   System.out.println (" Leaving CustomerDAOImpl.store()  ");
}
```

In method ejbFindByUserID(), first get the connection to database using field jdbcFactory. Create an SQL statement which searches for the customerid corresponding to a given userid in table Customer, where customerid is primary key. Code snippet is shown below**.**

```
public au.com.tusc.bmp.CustomerPK findByUserID (java.lang.String userID) throws javax.ejb.FinderException {

   System.out.println (" Entering CustomerDAOImpl.findByUserID()  ");
   Connection conn = null;
   PreparedStatement ps = null;
   ResultSet rs = null;
   CustomerPK pk = new CustomerPK() ;
   try {
      conn = jdbcFactory.getConnection();
      String queryString = "select customerid from customer where userid = ?";
      ps = conn.prepareStatement(queryString);
      ps.setString (1, userID);
      rs = ps.executeQuery();
      boolean result =rs .next();
      if ( result) {
         pk.setCustomerID( rs.getString(1) );
         System.out.println (" Primary Key found :" + pk.getCustomerID() );
      }
   }
   catch (Exception e) {
      e.printStackTrace();
           throw new FinderException("Inside CustomerDAOImpl.findbyPrimaryKey()" + e);
      }
   finally {
      try {
         rs.close();
         ps.close();
         conn.close();
      }
      catch(Exception e ) {
      }
   }
   System.out.println (" Leaving CustomerDAOImpl.findByUserID() with key " + pk.getCustomerID());
   return pk;
}
```

In method ejbfindByPrimaryKeystore(), first get the connection to database using field jdbcFactory. Create an SQL statement which searches for customerid in table Customer, where customerid is the primary key. Return the primary key. Code snippet is shown below.

```
public CustomerPK findByPrimaryKey(CustomerPK pk) throws javax.ejb.FinderException {

        System.out.println (" Entering CustomerDAOImpl.findByPrimaryKey()  ");
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            conn = jdbcFactory.getConnection();
            String queryString = "select customerid from customer where customerid = ?";
            ps = conn.prepareStatement(queryString);
            String key = pk.getCustomerID();
            ps.setString (1, key);
            rs = ps.executeQuery();
            boolean result =rs .next();
            if ( result) {
                System.out.println (" Primary Key found");
            }
        }
        catch (Exception e) {
            e.printStackTrace();
                throw new FinderException("Inside CustomerDAOImpl.findbyPrimaryKey()" +
                " following primarykey " + pk.getCustomerID() + "notfound ");
            }
        finally {
            try {
                rs.close();
                ps.close();
                conn.close();
            }
            catch(Exception e ) {
            }
        }
        System.out.println (" Leaving CustomerDAOImpl.findByPrimaryKey()  " + pk.getCustomerID());
        return pk;
}
```

Also, you should implement methods remove and create. We're leaving those as an exercise for you (you've probably got the idea by now!), or you can leave them as stubs as shown below.

```
public void remove(au.com.tusc.bmp.CustomerPK pk) throws javax.ejb.RemoveException, javax.ejb.EJBException {

}

public au.com.tusc.bmp.CustomerPK create(au.com.tusc.bmp.CustomerBean ejb) throws javax.ejb.CreateException, javax.ejb.EJBException {
    return null;
}
```

Now our CustomerDAOImpl class is finished. Generate your EJB classes again.

After generating your EJB classes, let's look at the Home Local Interface and the Remote Local interface.

In the Remote Local Interface (which is CustomerLocal in this case), there is one business method exposed, and the rest are all the methods to access the attributes/ properties of the bean (as we also declared these methods as interface methods in CustomerBean), as shown below. These methods are generated by @ejb. interface-method tag.

```
public interface CustomerLocal
  extends javax.ejb.EJBLocalObject
{
  /**
   * Returns the address
   * @return the address
   */
  public java.lang.String getAddress( ) ;

  /**
   * Returns the Customer Data Object
   * @return the CustomerData
   */
  public CustomerData getCustomerData( ) ;

  /**
   * Returns the customerID
   * @return the customerID
   */
  public java.lang.String getCustomerID( ) ;
```

In the Home Local Interface, which is CustomerLocalHome in this case, there are two finder methods as shown below. It has also generated JNDI_NAME and COMP_NAME (logical name to lookup the component).

```
 * Generated by XDoclet - Do not edit!
 */
package au.com.tusc.bmp;

/**
 * Local home interface for Customer.
 * @lomboz generated
 */
public interface CustomerLocalHome
   extends javax.ejb.EJBLocalHome
{
   public static final String COMP_NAME="java:comp/env/ejb/CustomerLocal";
   public static final String JNDI_NAME="CustomerLocal";

   public au.com.tusc.bmp.CustomerLocal findByPrimaryKey(au.com.tusc.bmp.CustomerPK pk)
      throws javax.ejb.FinderException;

   public au.com.tusc.bmp.CustomerLocal findByUserID(java.lang.String userID)
      throws javax.ejb.FinderException;

}
```

These names are generated by the following tag declared in CustomerBean as shown below.

```
package au.com.tusc.bmp;

import javax.ejb.EntityBean;
import javax.ejb.FinderException;

/**
 * @ejb.bean name="Customer"
 *  jndi-name="CustomerBean"
 *  type="BMP"
 *
 *
```

Note : We are not currently interested in the Customer interface (which is a Remote interface) and CustomerHome (which is a Remote Home interface), because we are accessing bean methods in the same Java Virtual Machine, so we need only Local interfaces.

Note : In CustomerBean we haven't implemented any methods for the setting and unsetting of context, because it is generated by Xdoclet in the CustomerBMP class, as shown below.

```
  private javax.ejb.EntityContext ctx = null;

  public void setEntityContext(javax.ejb.EntityContext ctx)
  {
    this.ctx = ctx;
  }

  public void unsetEntityContext()
  {
    this.ctx = null;
  }
```

Now Customer Bean and its DAO implementation is complete and we can deploy this bean.

Deploy Customer Bean :

In order to deploy this bean we have to add a few deployment descriptors to it. We will add the two tags shown below.

```
package au.com.tusc.bmp;

import javax.ejb.EntityBean;
import javax.ejb.FinderException;

/**
 * @ejb.bean name="Customer"
 *  jndi-name="CustomerBean"
 *  type="BMP"
 *
 *  @ejb.dao class="au.com.tusc.bmp.CustomerDAO"
 *  impl-class="au.com.tusc.dao.CustomerDAOImpl"
 *
 *  @ejb.resource-ref res-ref-name="jdbc/DefaultDS"
 *  res-type="javax.sql.Datasource"
 *  res-auth="Container"
 *
 *  @jboss.resource-ref  res-ref-name="jdbc/DefaultDS"
 *  jndi-name="java:/DefaultDS"
 **/

public abstract class CustomerBean implements EntityBean {
```

First add the tag shown below at class level in CustomerBean.

@ejb.resource-ref res-ref-name="jdbc/DefaultDS"
res-type="javax.sql.Datasource"
res-auth="Container"

This tag will generate deployment descriptors in ejb-jar.xml, as the bean has to know which datasource you are going to connect to, and what is its ts type, etc. This will generate these descriptors as shown below.

```
<!-- Entity Beans -->
<entity >
  <description><![CDATA[]]></description>

  <ejb-name>Customer</ejb-name>

  <home>au.com.tusc.bmp.CustomerHome</home>
  <remote>au.com.tusc.bmp.Customer</remote>
  <local-home>au.com.tusc.bmp.CustomerLocalHome</local-home>
  <local>au.com.tusc.bmp.CustomerLocal</local>

  <ejb-class>au.com.tusc.bmp.CustomerBMP</ejb-class>
  <persistence-type>Bean</persistence-type>
  <prim-key-class>au.com.tusc.bmp.CustomerPK</prim-key-class>
  <reentrant>False</reentrant>

  <resource-ref >
    <res-ref-name>jdbc/DefaultDS</res-ref-name>
    <res-type>javax.sql.Datasource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>

</entity>
```

Add this second tag required for the JBOSS application server.

@jboss.resource-ref res-ref-name="jdbc/DefaultDS"
jndi-name="java:/DefaultDS"

This tag will generate deployment descriptors in jboss.xml, as the application server has to know with what jndi-name the datasource has been registered with. This will generate these descriptors as shown below.

```
<entity>
  <ejb-name>Customer</ejb-name>
  <jndi-name>CustomerBean</jndi-name>
  <local-jndi-name>CustomerLocal</local-jndi-name>
  <resource-ref>
    <res-ref-name>jdbc/DefaultDS</res-ref-name>
    <jndi-name>java:/DefaultDS</jndi-name>
  </resource-ref>

</entity>
```

Now everything is complete, and it's time to deploy the bean. So, regenerate your EJB classes.

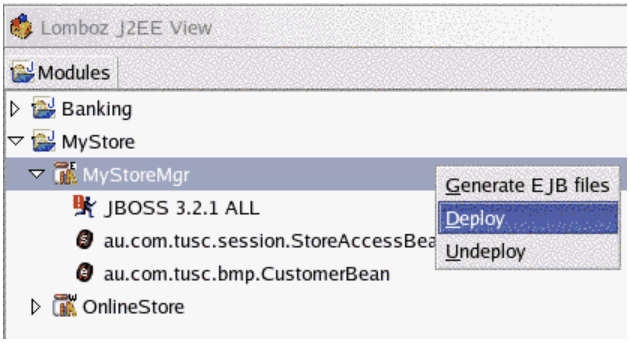
Note : As noted in earlier chapters we are regenerating classes over and over to properly illustrate each step and its result. Once you are proficient you will be able to forgo most of this.

Go to Lomboz J2EE View > expand node MyStore > expand MyStoreMgr > select 'Jboss 3.2.1 ALL' .

Right click > select 'Debug Sever' on pop up menu.

Note : This is to start your server, if you are already running your server then skip these steps and go to the next one.

Go to MyStoreMgr node in LombozJ2EE view > right click > select 'Deploy' on pop up menu as shown in figure below.



Note : All these techniques have been shown in previous chapters (1 and 3) so please refer to them.

The messages in the console will show whether your bean has been successfully deployed or not.

Now our Customer Bean is complete, and in order to create a client to invoke operations on this bean we have make some modifications to our StoreAccess Bean.

Note : As shown in the diagram at the beginning of this chapter that client will invoke operations on Customer Bean through Store AccesBean, as it is a good practice to access Entity Beans in this manner. As a result we need a Local view of Customer Bean rather than Remote, because both are in the same Java Virtual Machine.

Add Create Method in StoreAccess :

In StoreAccess Bean add an ejbCreate method, which will create a BMP Entity Bean (in this case a Customer Bean) with the following signature

public void ejbCreate () throws javax.ejb.CreateException

Now, add a field to store the reference obtained by locating Customer in JNDI.

private CustomerLocalHome customerLocalHome;

In ejbCreate method store the reference in the customerLocalHome variable by invoking the getLocalHome static method in CustomerUtil class as shown in the code snippet below from StoreAccess Bean.

```
/**
 * The  ejbCreate method.
 * @ejb.create-method
 **/
public void ejbCreate () throws  javax.ejb.CreateException {

    System.out.println (" Entering  StoreAccessBean.ejbCreate() ");
    try {
       customerLocalHome = CustomerUtil.getLocalHome();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    System.out.println (" Leaving  StoreAccesBean.ejbCreate() ");
}
```

Add Business Method in StoreAccess :

Add another business method in StoreAccess Bean which will invoke the corresponding business method on Customer Bean.

Now, add a business method with this signature **public CustomerData getCustomerData(String userID)** with Interface type as Remote. As customers will log on to MyStore with username, once they are authenticated they will be identified by userid and they can retrieve their account details from MyStore using this userid.

Note : Steps to create business methods are covered in previous chapters.

Now invoke one of the finder methods of Customer on the reference variable we have created in the ejbCreate method.

CustomerLocal myCustomer = customerLocalHome.findByUserID(userID)

Now invoke the business method on Customer using the myCustomer reference variable.

CustomerData cd = myCustomer.getCustomerData()

Code snippet of this business method is shown below.

```
/**
 * Returns object CustomerData
 * @ejb.interface-method
 *  tview-type="remote"
 **/
 public au.com.tusc.bmp.CustomerData getCustomerData(String userID){

    System.out.println (" Entering StoreAccessBean.getCustomerData() ");
    CustomerData cd = null;
    try {
       CustomerLocal myCustomer =  customerLocalHome.findByUserID(userID);
       if (myCustomer != null ) {
          cd = myCustomer.getCustomerData();
       }
    } catch (Exception e) {
         System.out.println (" Error in StoreAccessBean.getCustomerData() " + e);
    }
    System.out.println (" Leaving StoreAccessBean.getCustomerData() ");
    return cd;
 }
```

All methods in StoreAccess Bean have now been added for the accessing the Customer's business method. All that remains are the deployment descriptors required for linking/referencing the StoreAccess and Customer beans. We will add the two tags shown below.

```
*/
package au.com.tusc.session;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import au.com.tusc.bmp.CustomerData;
import au.com.tusc.bmp.CustomerLocal;
import au.com.tusc.bmp.CustomerLocalHome;
import au.com.tusc.bmp.CustomerUtil;
/**
 * @ejb.bean name="StoreAccess"
 *  jndi-name="StoreAccessBean"
 *  type="Stateless"
 *
 * @ejb.dao class="au.com.tusc.session.StoreAccessDAO"
 *  impl-class="au.com.tusc.dao.StoreAccessDAOImpl"
 *
 * @ejb.resource-ref res-ref-name="jdbc/DefaultDS"
 *      res-type="javax.sql.Datasource"
 *      res-auth="Container"
 *
 * @jboss.resource-ref  res-ref-name="jdbc/DefaultDS"
 *      jndi-name="java:/DefaultDS"
 *
 * @ejb.ejb-ref ejb-name="Customer"
 *    view-type="local"
 *    ref-name="CustomerLocal"
 *
 *  @jboss.ejb-ref-jndi ref-name="CustomerLocal"
 *     jndi-name="CustomerLocal"
 **/


public abstract class StoreAccessBean implements SessionBean {
```

First add this tag at class level in StoreAccess Bean.

```
@ejb.ejb-ref ejb-name="Customer"
  view-type="local"
  ref-name="CustomerLocal"
```

This tag will generate deployment descriptors in ejb-jar.xml, as StoreAccessBean has to know which bean it's referring to, and what is its view-type and ref-name. This will generate these descriptors as shown below.

Note : View type is local as both are in the same Java Virtual Machine, otherwise it would be Remote. Note that ref-name is generated as CustomerLocalHome rather than CustomerHome. Both are generated but we are using Local in this case.

```
    <!-- Session Beans -->
    <session >
      <description><![CDATA[]]></description>

      <ejb-name>StoreAccess</ejb-name>

      <home>au.com.tusc.session.StoreAccessHome</home>
      <remote>au.com.tusc.session.StoreAccess</remote>
      <local-home>au.com.tusc.session.StoreAccessLocalHome</local-home>
      <local>au.com.tusc.session.StoreAccessLocal</local>
      <ejb-class>au.com.tusc.session.StoreAccessSession</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>

      <ejb-local-ref >
        <ejb-ref-name>ejb/CustomerLocal</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <local-home>au.com.tusc.bmp.CustomerLocalHome</local-home>
        <local>au.com.tusc.bmp.CustomerLocal</local>
        <ejb-link>Customer</ejb-link>
      </ejb-local-ref>

      <resource-ref >
        <res-ref-name>jdbc/DefaultDS</res-ref-name>
        <res-type>javax.sql.Datasource</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>

    </session>
```

Now add the second tag shown below at class level in StoreAccess Bean.

```
@jboss.ejb-ref-jndi ref-name="CustomerLocal"
jndi-name="CustomerLocal"
```

This tag will generate deployment descriptors in jboss.xml, as the application server has to know what jndi-name Customer Bean has been registered with. This will generate these descriptors as shown below.

Note : Ref-name and jndi-name are used for bean as local (in same JVM).

```
<session>
  <ejb-name>StoreAccess</ejb-name>
  <jndi-name>StoreAccessBean</jndi-name>
  <local-jndi-name>StoreAccessLocal</local-jndi-name>
  <ejb-ref>
    <ejb-ref-name>ejb/CustomerLocal</ejb-ref-name>
    <jndi-name>CustomerLocal</jndi-name>
  </ejb-ref>
  <resource-ref>
    <res-ref-name>jdbc/DefaultDS</res-ref-name>
    <jndi-name>java:/DefaultDS</jndi-name>
  </resource-ref>
</session>
```

should be <ejb-local-ref>

Note : As we can see from the code snippet above, the deployment descriptor generated by tag @jboss is wrong, because for local referencing of Customer tag <ejb-ref> should be <ejb-local-ref>. There seems to be a bug in this tag, so we will correct this manually by changing the tag in the jboss.xml file as shown below.

```
<session>
  <ejb-name>StoreAccess</ejb-name>
  <jndi-name>StoreAccessBean</jndi-name>
  <local-jndi-name>StoreAccessLocal</local-jndi-name>
  <ejb-local-ref>
    <ejb-ref-name>ejb/CustomerLocal</ejb-ref-name>
    <jndi-name>CustomerLocal</jndi-name>
  </ejb-local-ref>
  <resource-ref>
    <res-ref-name>jdbc/DefaultDS</res-ref-name>
    <jndi-name>java:/DefaultDS</jndi-name>
  </resource-ref>
</session>
```

Caution here: *Make sure that you do this change manually after you finish regenerating your EJB classes, because every time you regenerate your classes, 'jboss.xml' will initially have the wrong descriptors generated by this tag.*

Now our changes to StoreAccess Bean are complete, so deploy your bean again now from the Lomboz J2EE View. The steps to do that are shown above and in previous chapters. Messages will appear in the console showing the status of deployment.

Once your bean is deployed successfully, create a test client which will invoke the loginUser method on StoreAccessBean and getCustomerData on CustomerBean.

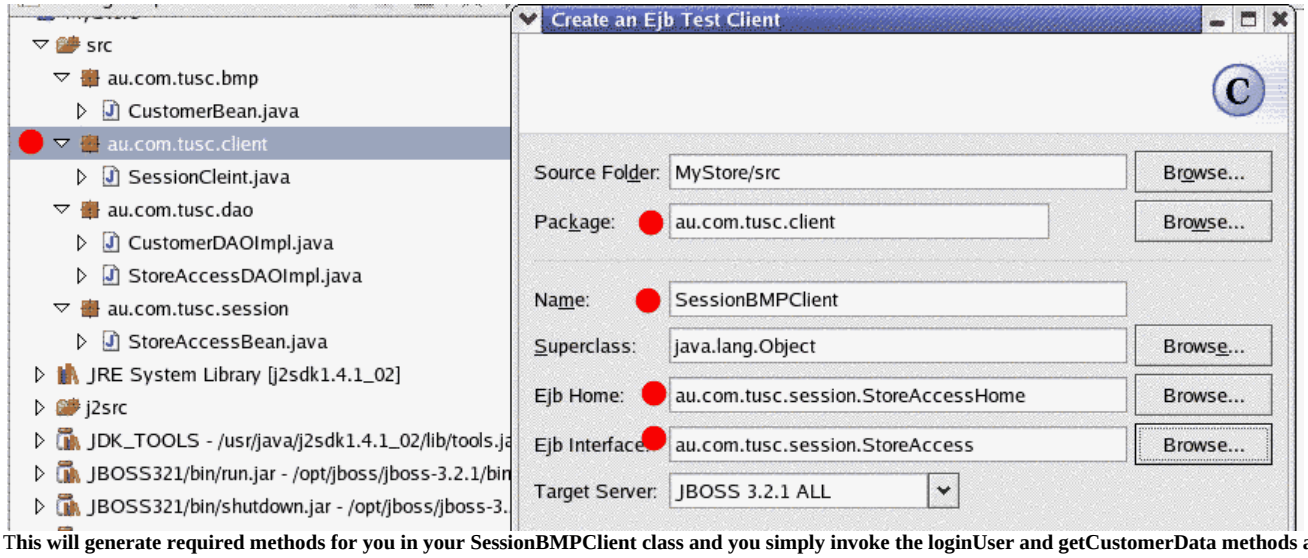
Create your Test Client :

Go to Project MytStore node > select src node abd expand it > select au.com.tusc.client package > right click.

Select New on pop up menu > select Lomboz EJB Test Client Wizard.

Note : These steps are covered in previous chapters.

Select package name au.com.tusc.client, name as SessionBMPClient and select Ejb Home as au.com.tusc.session.StoreAccessHome and Ejb Interface as au.com.tusc.session.StoreAccess as shown in figure below.



This will generate required methods for you in your SessionBMPClient class and you simply invoke the loginUser and getCustomerData methods as shown.

```
package au.com.tusc.client;

import java.rmi.RemoteException;
import java.util.Hashtable;
import javax.ejb.CreateException;
import javax.naming.InitialContext;
import javax.naming.NamingException;


public class SessionBMPClient {

   private au.com.tusc.session.StoreAccessHome getHome()
      throws NamingException {
      return (au.com.tusc.session.StoreAccessHome) getContext().lookup(
         au.com.tusc.session.StoreAccessHome.JNDI_NAME);
   }
   private InitialContext getContext() throws NamingException {
      Hashtable props = new Hashtable();
      props.put(
         InitialContext.INITIAL_CONTEXT_FACTORY,
         "org.jnp.interfaces.NamingContextFactory");
      props.put(InitialContext.PROVIDER_URL, "jnp://127.0.0.1:1099");
      InitialContext initialContext = new InitialContext(props);
      return initialContext;
   }
   public void testBean() {

      try {
         au.com.tusc.session.StoreAccess myBean = getHome().create();
         //-------------------------------------
         //This is the place you make your calls.
         //System.out.println(myBean.callYourMethod());

      } catch (RemoteException e) {
         e.printStackTrace();
      } catch (CreateException e) {
```

Now to add some code to your client.

Add these lines under the testBean method as shown below.

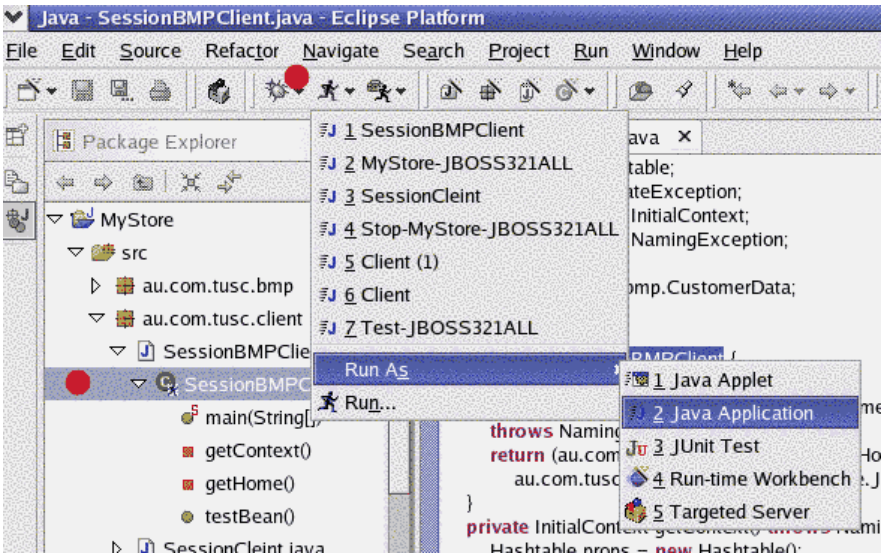
```
System.out.println("Request from client : ");
String userID = myBean.loginUser("ANDY","PASSWD");
System.out.println("Reply from Server: Your userid is " + userID );
CustomerData cd = myBean.getCustomerData(userID);
System.out.println ("Andy your details with MyStore are " + cd );
```

```
   public void testBean() {

      try {
         au.com.tusc.session.StoreAccess myBean = getHome().create();
         //-------------------------------------
         //This is the place you make your calls.
         //System.out.println(myBean.callYourMethod());
         System.out.println("Request from client : ");
         String userID =   myBean.loginUser("ANDY","PASSWD");
         System.out.println("Reply from Server: Your userid is " + userID );
         CustomerData cd = myBean.getCustomerData(userID);
         System.out.println ("Andy your details with MyStore are  " + cd );

      } catch (RemoteException e) {
         e.printStackTrace();
      } catch (CreateException e) {
         e.printStackTrace();
      } catch (NamingException e) {
         e.printStackTrace();
      }
   }

   public static void main(String[] args) {
      SessionBMPClient test = new SessionBMPClient();
      test.testBean();

   }
}
```

Test your Client :

Now, in order to test your client, Select SessionBMPClient node > Go at top level menu and select the 'Man running' icon.

On that select 'Run as' > select Java Application, as shown below.



Now under your console, if your reply for 'ANDY' is 'U2', and his details for CustomerID are 'C2' as well, then your call is successful (see below).

```
Request from client :
Reply from Server: Your userid is U2
Andy your details with MyStore are  {customerID=C2 userID=U2 firstName=ANDREW lastName=HENDRY add
```

Exercise :

Here is an exercise for you. To progress further, implement Manager as a BMP Entity bean similar to Customer, with the same behaviours. The detailed tasks are given below.

1. Create a BMP Bean named Manager under package au.com.tusc.bmp.
2. Create a DAO class named ManagerDAOImpl under package au.com.tusc.dao.
3. Add all attribute/properties in ManagerBean, add accessor and mutator methods for each attribute.
4. Add a find method named ejbFindByPrimaryKey with signature

   public ManagerPK ejbFindByPrimaryKey (MangerPK pk) throws FinderException.

5. Add a find method named ejbFindByUserID with signature

   public MangerPK ejbFindByUserID (String userID) throws FinderException

6. Add a business method named getManagerData with signature

   public ManagerData getManagerData()

7. Implement methods in ManagerDAOImpl class. Lookup string required for JNDI API is "java:comp/env/jdbc/DefaultDS".

8. Deploy Manager Bean.

9. Add a field in StoreAccess Bean to store reference after lookup in JNDI for Manager

   private ManagerLocalHome managerLocalHome;

10. In ejbCreate method of StoreAccess Bean store reference in managerLocalHome variable by invoking getLocalHome static method in ManagerUtil.

11. Add a business method in StoreAccess Bean.

    public MangerData getManagerData(String userID)

12. Add the following tags for deployment at class level for linking/referencing Manager.

    1. @ejb.ejb-ref ejb-name="Manager"
       view-type="local"
       ref-name="ManagerLocal"

    2. @jboss.ejb-ref-jndi ref-name="ManagerLocal"
       jndi-name="ManagerLocal"

13. Test your Manager Bean by running your Test Client created for Customer named SessionBMPClient.

Note : Don't forget to fix the generated descriptors in jboss.xml as mentioned above in relation to Customer Bean.

Note : All these steps are the same as those for Customer. Implement this Bean which will be used in subsequent chapters.

In case you're unable to do that, we have provided ManagerBean, ManagerDAOImpl, modified StoreAccessBean and modified SessionBMPClient classes. You can download these files from downloads below.

Downloads :

ManagerBean

ManagerDAOImpl

StoreAccessBean

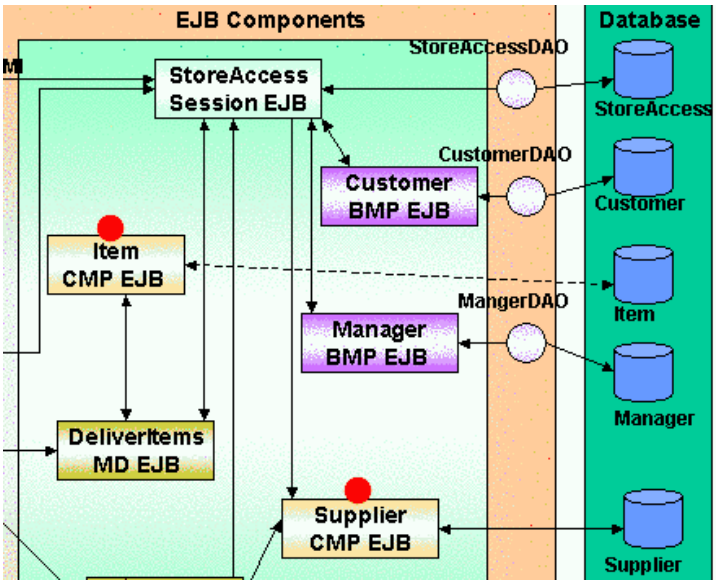
SessionBMPClient

Prev TOC Next

Chapter 6.

Creating a CMP Entity Bean

This chapter covers how to create a Container Managed Persistence (CMP) EJB component. We will create two CMP beans, Item and Supplier as shown below. The Item bean will be responsible for storing the details of items, such as their availabability and their prices, for MyStore. The Supplier Bean stores details of Suppliers to MyStore. Both beans interact with corresponding tables in the database. In CMP this interaction is controlled by the container, in this case the JBOSS CMP container.



All Items have been assigned a unique itemId for housekeeping purposes within MyStore, and all suppliers have been assigned a unique supplierID in addition to their username which is what they use in accessing the services of MyStore.

Note : It is normal practice to access the business methods of CMP beans via a Session bean, that encapsulates the business logic and acts as an interface to the lower-level EJB components. In this case Supplier and Items are accessed via StoreAccess.

Tasks :

1. Create a CMP bean named Items under package au.com.tusc.cmp.
2. Implement the ejbCreate method, with the values of all attributes being passed as arguments and then assigned to the attributes using mutator (setter) methods.
3. Add a finder method named findBySupplierID with the following query and signature:

   query "SELECT OBJECT(b) FROM MyStoreItem as b where b.supplierID = ?1"

   method "java.util.Collection findBySupplierID(java.lang.String supplierID)"
4. Add a finder method named findByOutOfStock with the following query and signature:

   query "SELECT OBJECT(c) FROM MyStoreItem as c where c.quantity = 0"

method "java.util.Collection findByOutOfStock()"

5. Add a business method to get item details with the signature:

   public ItemData getItemData()

6. Add another business method to register delivery of items with the signature:

   public void fillStock(java.lang.Integer quantity)

7. Add callback methods, required for getting/setting bean context for bean with signatures:

   public void setEntityContext(EntityContext ctx)

   public void unsetEntityContext()

8. Deploy the Item Bean.

9. Add a field to the StoreAccess bean to store the Item reference (obtained from JNDI lookup):

   private ItemLocalHome itemLocalHome

10. In the ejbCreate method of the StoreAccess bean store this reference in the itemLocalHome variable by invoking the getLocalHome static method in ItemUtil.

11. Add a business method to StoreAccess Bean with the signature:

    public ItemData getItemData(String itemID)

12. Add another business method to StoreAccess Bean with the signature:

    public java.util.ArrayList getOutOfStockItems()

13. Add another business method to StoreAccess Bean with the signature:

    public java.util.ArrayList getItemBySupplier(String supplierID)

14. Create a test client named SessionCMPClient under package au.com.tusc.client.

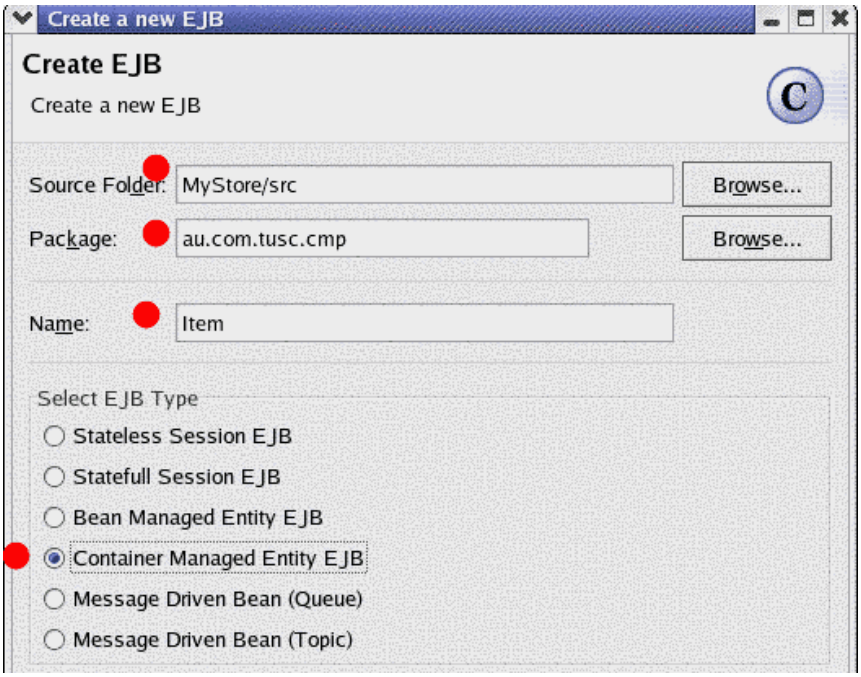
15. Run your client and test the bean.

Create Items CMP Entity Bean :

Go To Package Explorer > Expand Mystore (project) node > select src, right click and a menu will pop up.

On the pop up menu > New > Lomboz EJB Creation Wizard.

Enter the package name au.com.tusc.cmp, the bean name Item and select the bean type as Container Manged Entity as shown below.



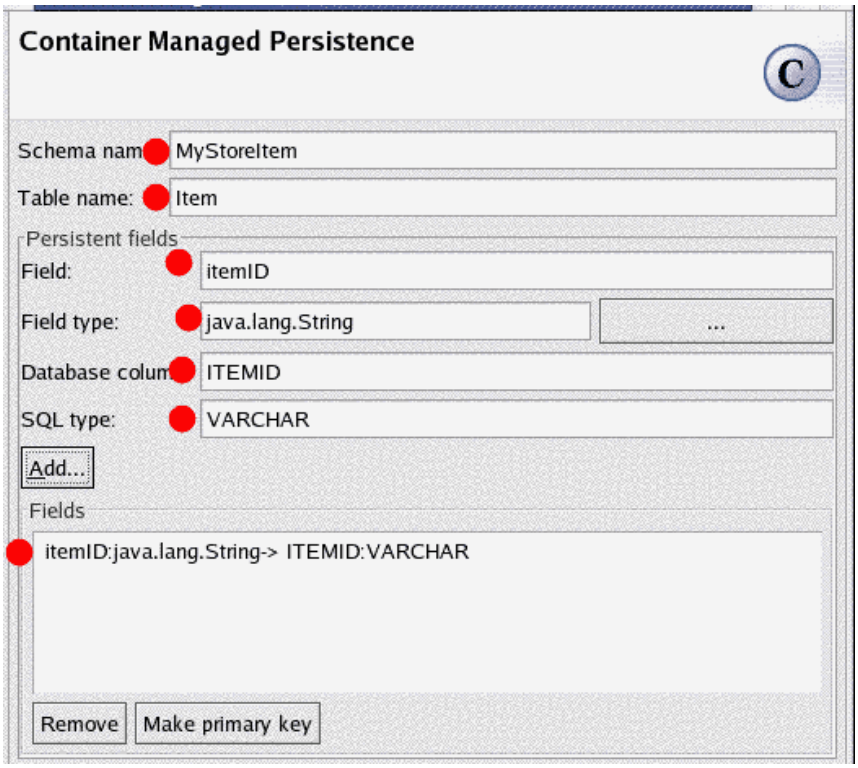
Go to Next and a new screen will pop up as shown below.

Enter MyStoreItem as the Schema Name.

Enter Item as the Table name.

Under Persistent Fields first enter itemID as the Field, with a Field Type of java.lang.String, ITEMID as its Database column, and VARCHAR for its SQL Type.

Press Add .. > It will add this field in Fields section, select this new field > Press Make Primary Key.



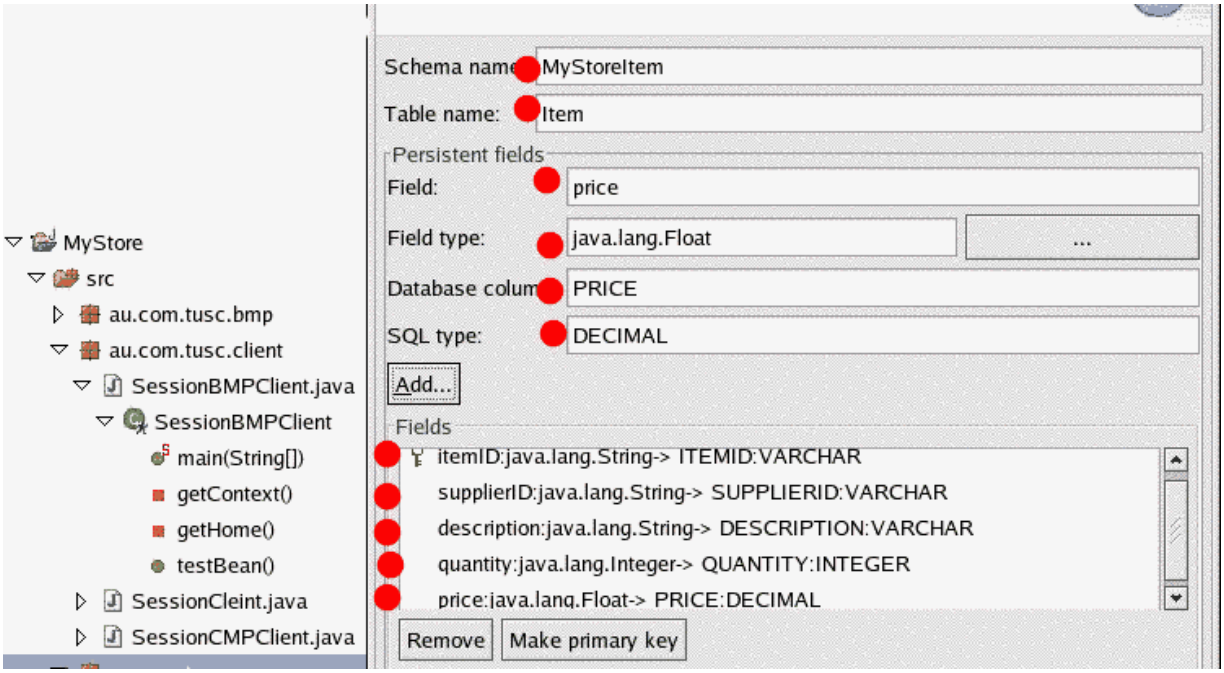
Similarly, add all the rest of the fields of the items table as shown below.

Add .. Field: supplierID, Field Type: java.lang.String, Database Column: SUPPLIERID, SQL Type: VARCHAR.

Add .. Field: description, Field Type: java.lang.String, Database Column: DESCRIPTION, SQL Type: VARCHAR.

Add .. Field: quantity, Field Type: java.lang.Integer, Database Column: QUANTITY, SQL Type: INTEGER.

Add .. Field: price, Field Type: java.lang.Float, Database Column: PRICE, SQL Type: DECIMAL.



After adding all these fields, press Finish.

This will create a package named au.com.tusc.cmp under src, and ItemBean will be be created within that package as shown below.

```
package au.com.tusc.cmp;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
/**
 * @ejb.bean name="Item"
 * jndi-name="ItemBean"
 * type="CMP"
 * primkey-field="itemID"
 * schema="MyStoreItem"
 * cmp-version="2.x"
 *
 * @ejb.persistence
 *  table-name="Item"
 *
 * @ejb.finder
 *   query="SELECT OBJECT(a) FROM MyStoreItem as a"
 *   signature="java.util.Collection findAll()"
 *
 **/

public abstract class ItemBean implements EntityBean {

   /**
    * The  ejbCreate method.
    *
    * @ejb.create-method
    */
   public java.lang.String ejbCreate(String itemID, String supplierID, String description,
                   Integer qunatity, Float price) throws javax.ejb.CreateException {
      // EJB 2.0 spec says return null for CMP ejbCreate methods.
      // TODO: YOU MUST INITIALIZE THE FIELDS FOR THE BEAN HERE.
      // setMyField("Something");

      return null;
   }
```

Note: In comparison with our earlier BMP Entity Beans (Customer & Manager), more tags have been generated at class level. Note also that CMP doesn't require a Data Access Object (DAO) interface, as communication between database and bean is controlled by the container.

Let's first the generate EJB classes and then we will examine these tags.

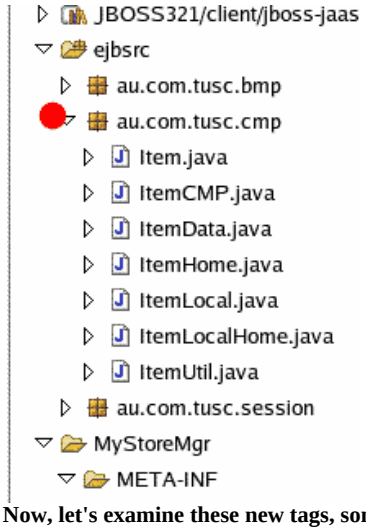
Go to node ItemBean under au.com.tusc.cmp > LombozJ2EE > Add EJB to Module > Select MyStoreMgr > Ok.

Go to MyStoreMgr node > LombozJ2EE > Generate EJB classes.

Note: All these steps are covered in previous chapters (chapters 3 and 1) in detail, so please refer to these if you have any queries.

Now, let's see what files have been generated by Xdoclet.

As shown below, the files generated are nearly the same as for BMP, except there are no Primary Key or DAO classes, and there is now ItemCMP which extends the ItemBean class. The remainder are the same as for BMP entity beans.



Now, let's examine these new tags, some of which have been covered in previous chapters.

1. **@ejb.bean** tag provides information about the EJB. It is the one compulsory tag for all EJBs**.**
2. **@ejb.persistence tag is being used at two levels, at class level and method level.** At class level it provides information about the persistence of a CMP entity bean, that is which database table this bean is going to interact with, which will provide that persistence. At method level it provides information about the mapping of the bean's persistent attributes to columns in that database table.
3. **@ejb.finder** tag defines a finder method for the home interface. This requires the EJB QL query to fetch the data and a signature for the method. This tag can be used for multiple finder methods.

4. **@ejb.persistence-field** method level tag is being deprecated in favour of @ejb.persistence tag, it provided information about persistent fields.
5. **@ejb.pk-field** tag defines the primary key.

The code snippet below shows how persistent attributes are declared in a CMP Entity Bean.

```
    * Returns the itemID
    * @return the itemID
    *
    @ejb.persistent-field
    @ejb.persistence
    *    column-name="ITEMID"
    *     sql-type="VARCHAR"
    @ejb.pk-field
    @ejb.interface-method
    */
   public abstract java.lang.String getItemID();

   /**
    * Sets the itemID
    *
    * @param java.lang.String the new itemID value
    *
    * @ejb.interface-method
    */
   public abstract void setItemID(java.lang.String itemID);

   /**
    * Returns the supplierID
    * @return the supplierID
    *
    * @ejb.persistent-field
    * @ejb.persistence
    *    column-name="SUPPLIERID"
    *     sql-type="VARCHAR"
    *
    * @ejb.interface-method
    */
   public abstract java.lang.String getSupplierID();

   /**
```

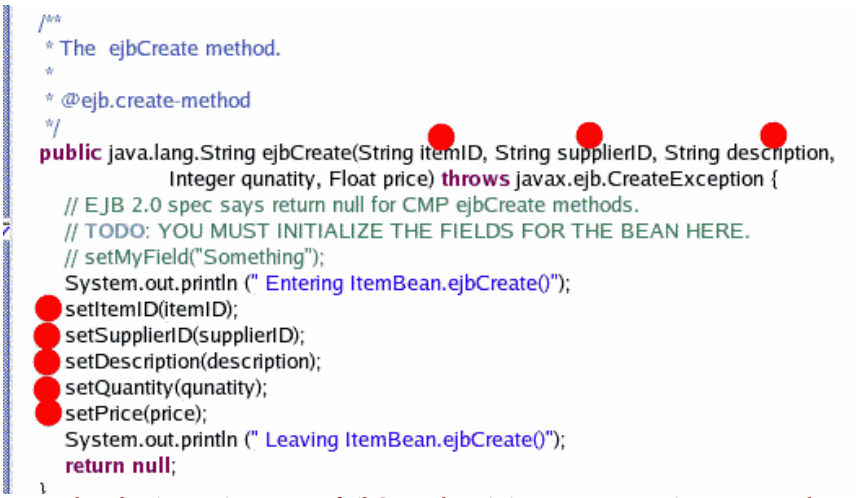
Note: All persistent attributes are declared with abstract accessor and mutator methods in ItemBean. Note also that in the case of a composite primary key

you have to specify the @ejb.pk-field tag on all other attributes/properties which combine to form the overall key as well.

Implement ejbCreate Method :

The ejbCreate method is created by method is created by the Lomboz Bean wizard, but we still have to add some code to complete it. Modify the signature of ejbCreate, passing all the attributes as parameters and then setting all these attributes using their associated mutator methods, as shown below.

Errata note :- There is an error in the figure shown below. The Integer attribute 'qunatity' should be called 'quantity'.



```
/**
 * The  ejbCreate method.
 *
 * @ejb.create-method
 */
public java.lang.String ejbCreate(String itemID, String supplierID, String description,
                Integer qunatity, Float price) throws javax.ejb.CreateException {
    // EJB 2.0 spec says return null for CMP ejbCreate methods.
    // TODO: YOU MUST INITIALIZE THE FIELDS FOR THE BEAN HERE.
    // setMyField("Something");
    System.out.println (" Entering ItemBean.ejbCreate()");
    setItemID(itemID);
    setSupplierID(supplierID);
    setDescription(description);
    setQuantity(qunatity);
    setPrice(price);
    System.out.println (" Leaving ItemBean.ejbCreate()");
    return null;
}
```

Note : The other interesting aspect of ejbCreate here is its return type, as its return type has to be same as the primary key type (e.g. it has to of type String, Integer, Float, or whatever). In this case it is String - the type of itemID, and when implemented it should return null.(Refer to the EJB Spec 10.5.1).

Add Finder Method :

Note: An entity bean's home interface defines one or more finder methods, to find an entity object or collection of entity objects. The name of each finder method starts with the prefix 'find', such as findPrice or findQuantity in our case. Every finder method except findByPrimaryKey(key) must be associated with a query element in the deployment descriptor. The entity bean provider declares the EJB QL finder query and associates it with the finder method in the deployment descriptor. A finder method is normally characterized by an EJB QL query string specified via the query element. Refer to the EJB Spec 10.5.6. This is covered in the Exercise section of this chapter.

Now let's add a finder method to our bean class to find items supplied by a particular supplier.

In order to add this finder method we have to declare a class level tag as discussed above, that is @ejb.finder.

So, add this tag:

```
@ejb.finder
query="SELECT OBJECT(a) FROM MyStoreItem a where a.supplierID = ?1"
signature="java.util.Collection findBySupplierID(java.lang.String supplierID)"
```

Note : In EJB QL, instead of the name of the table, the schema name is used (in this case it is MyStoreItem, rather then specifying Item as you would for an SQL query). Similarly the column names that would be used in SQL are replaced by their corresponding attributes as declared in the bean.

Generate your EJB classes to see what this tag and the previous finder tag have created in the Home Interface as shown below in this code snippet from the ItemLocalHome interface.

```
/**
 * Generated by XDoclet - Do not edit!
 */
package au.com.tusc.cmp;

/**
 * Local home interface for Item.
 * @lomboz generated
 */
public interface ItemLocalHome
   extends javax.ejb.EJBLocalHome
{
   public static final String COMP_NAME="java:comp/env/ejb/ItemLocal";
   public static final String JNDI_NAME="ItemLocal";

   public au.com.tusc.cmp.ItemLocal create(java.lang.String itemID , java.lang.String supplierID ,
                java.lang.String description , java.lang.Integer qunatity , java.lang.Float price)
      throws javax.ejb.CreateException;

   public java.util.Collection findAll()
      throws javax.ejb.FinderException;

   public java.util.Collection findSupplierID(java.lang.String supplierID)
      throws javax.ejb.FinderException;

   public au.com.tusc.cmp.ItemLocal findByPrimaryKey(java.lang.String pk)
      throws javax.ejb.FinderException;
```

It has four methods, including the two finder methods generated by the finder tags declared at class level. The other two, create and findByPrimaryKey are created by Xdoclet(because of the <entitycmp/> tag in ejbGenerate.xml**).**

Add another finder method to find out-of-stock items in MyStore.

Add the following tag.

```
@ejb.finder
query="SELECT OBJECT(c) FROM MyStoreItem c where c.quantity = 0"
signature="java.util.Collection findByOutOfStock()"
```

Code snippet from ItemLocalHome after generating EJB classes for these finder methods.

```
/*
package au.com.tusc.cmp;

/**
 * Local home interface for Item.
 * @lomboz generated
 */
public interface ItemLocalHome
   extends javax.ejb.EJBLocalHome
{
   public static final String COMP_NAME="java:comp/env/ejb/ItemLocal";
   public static final String JNDI_NAME="ItemLocal";

   public au.com.tusc.cmp.ItemLocal create(java.lang.String itemID ,
         java.lang.String supplierID , java.lang.String description ,
         java.lang.Integer qunatity , java.lang.Float price)
      throws javax.ejb.CreateException;

   public java.util.Collection findAll()
      throws javax.ejb.FinderException;

   public java.util.Collection findSupplierID(java.lang.String supplierID)
      throws javax.ejb.FinderException;

   public java.util.Collection findByOutOfStock()
      throws javax.ejb.FinderException;

   public au.com.tusc.cmp.ItemLocal findByPrimaryKey(java.lang.String pk)
      throws javax.ejb.FinderException;

}
```

Now, generate your EJB classes and analyze the relevant classes. All the finder methods are complete. Let's add some business methods now.

Add Business Methods :

Now, add a business method with this signature:

public ItemData getItemData()

.. with Interface type as local.

Note : The steps to add a business method are covered in previous chapters (1 and 3), so please refer to them. Also we have chosen the Interface type as local because these methods will be invoked within the same Java Virtual Machine.

This will provide description of individual items in MyStore. Add some debug statements and return an instance of ItemData as shown below in this code snippet from the Item bean.

```
/**
 * @ejb.interface-method
 *  tview-type="local"
 **/
public ItemData getItemData()  {
    System.out.println ("Entering ItemBean.getItemData() ");
    System.out.println ("Leaving ItemBean.getItemData() ");
    return new ItemData ( getItemID(), getSupplierID(),
    getDescription(), getQuantity(),getPrice());
 }
```

Add another business method with the following signature:

public void fillStock (Integer quantity)

.. with Interface type as local. This will increment the items available to MyStore after the delivery of further items. Add some debug statements and following lines to complete the method.

Integer qty   = new Integer((quantity.intValue() + getQuantity().intValue())) ;
setQuantity (qty);

Code snippet of fillStock in ItemBean shown below.

```
    /**
     * @ejb.interface-method
     *  tview-type="local"
     **/
    public  void  fillStock(java.lang.Integer quantity) {

        System.out.println (" Entering ItemBean.fillStock() with quantity " + quantity.intValue() );
        Integer qty   = new Integer(  (quantity.intValue() + getQuantity().intValue()) ) ;
        System.out.println (" Quantity of items after delivery of items " + qty.intValue());
        setQuantity ( qty );
        System.out.println (" Leaving ItemBean.fillStock()");
    }
}
```

Add Callback Method

Unlike BMP (where they were generated) we have to add callback methods which will be overridden in the ItemCMP class.

First import the following package: **javax.ejb.EntityContext**

Add a field to store the entity context.

protected EntityContext eContext;

Add a method setEntityContext with entityContext as its parameter and assign that to your entityContext variable.

Similarly add a method unsetEntityContext, which sets the entityContext variable to null.

Code snippet for both methods is shown below.

```
protected EntityContext eContext;

/**
 * Sets the entity context
 * @param javax.ejb.EntityContext the new eContext value
 *
**/
public void setEntityContext(EntityContext context) {
    eContext = context;
}

/**
 * Unsets the enity context
 *  @param javax.ejb.EntityContext eContext value
 *
**/
public void unsetEntityContext()    {
    eContext = null;
}
```

Now all business and finder methods are complete, so it's time to generate EJB classes.

Let's examine the generated ItemCMP class, which is of most interest .

Unlike our BMP bean all persistent attribute behavior is being overridden by abstract methods. This is because the EJB container is responsible for maintaining their persistence.

ItemCMP.java

```
public void unsetEntityContext()
{
  super.unsetEntityContext();
}

public void ejbRemove() throws javax.ejb.RemoveException
{

}

/* Value Objects BEGIN */

/* Value Objects END */

public abstract java.lang.String getItemID() ;

public abstract void setItemID( java.lang.String itemID ) ;

public abstract java.lang.String getSupplierID() ;

public abstract void setSupplierID( java.lang.String supplierID ) ;

public abstract java.lang.String getDescription() ;

public abstract void setDescription( java.lang.String description ) ;

public abstract java.lang.Integer getQuantity() ;

public abstract void setQuantity( java.lang.Integer quantity ) ;

public abstract java.lang.Float getPrice() ;

public abstract void setPrice( java.lang.Float price ) ;

}
```

All the callback methods we have implemented are being overridden as shown below.

```
public void setEntityContext(javax.ejb.EntityContext ctx)
{
  super.setEntityContext(ctx);
}

public void unsetEntityContext()
{
  super.unsetEntityContext();
}
```

Note : There are no ejbFinder methods in this class as with BMP beans, as all this is controlled by the container. Also as pointed out before, there is no

PrimaryKey class generated and there is no need for a DAO class, as this too is controlled by the container.

Now, before we deploy our bean, we will just have a look at ejb-jar.xml and jboss.xml to see what descriptors are generated.

Note : We don't have to write any descriptors for the data sources as we did with Session and BMP beans, as the EJB container is responsible for that.

```
<entity >
  <description><![CDATA[]]></description>
  <ejb-name>Item</ejb-name>
  <home>au.com.tusc.cmp.ItemHome</home>
  <remote>au.com.tusc.cmp.Item</remote>
  <local-home>au.com.tusc.cmp.ItemLocalHome</local-home>
  <local>au.com.tusc.cmp.ItemLocal</local>
  <ejb-class>au.com.tusc.cmp.ItemCMP</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.String</prim-key-class>
  <reentrant>False</reentrant>
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>MyStoreItem</abstract-schema-name>
  <cmp-field >
    <description><![CDATA[Returns the itemID]]></description>
    <field-name>itemID</field-name>
  </cmp-field>
  <cmp-field >
    <description><![CDATA[Returns the supplierID]]></description>
    <field-name>supplierID</field-name>
  </cmp-field>
  <cmp-field >
    <description><![CDATA[Returns the description]]></description>
    <field-name>description</field-name>
  </cmp-field>
  <cmp-field >
    <description><![CDATA[Returns the quantity]]></description>
    <field-name>quantity</field-name>
  </cmp-field>
  <cmp-field >
    <description><![CDATA[Returns the price]]></description>
    <field-name>price</field-name>
  </cmp-field>
  <primkey-field>itemID</primkey-field>

  <query>
```

As shown in the code snippet above from the ejb-jar.xml file, all abstract methods are generated as persistent fields under the tag <cmp-field> because of the tag @persistence-field declared at each accessor method. Also the primary key class descriptor is generated under the <primary-key-class> tag and the primary key field is generated under the <primarykey-field> because of the tag @field-pk declared for the relevant attribute(s).

Descriptors for finder methods are generated along with the query defined to fetch the data as shown below. These finder tags are generated by the @ejb.finder tag declared at class level.

```
    </cmp-field>
    <primkey-field>itemID</primkey-field>

    <query>
      <query-method>
        <method-name>findAll</method-name>
        <method-params>
        </method-params>
      </query-method>
      <ejb-ql><![CDATA[SELECT OBJECT(a) FROM MyStoreItem as a]]></ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findBySupplierID</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
        </method-params>
      </query-method>
      <ejb-ql><![CDATA[SELECT OBJECT(b) FROM MyStoreItem b where b.supplierID = ?1]]></ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findByOutOfStock</method-name>
        <method-params>
        </method-params>
      </query-method>
      <ejb-ql><![CDATA[SELECT OBJECT(c) FROM MyStoreItem c where c.quantity = 0]]></ejb-ql>
    </query>
  <!-- Write a file named ejb-finders-ItemBean.xml if you want to define extra finders. -->
  </entity>
```

```
    </query>
  <!-- Write a file named ejb-finders-ItemBean.xml if you want to define extra finders. -->
  </entity>
```

And in the jboss.xml file, the following descriptors are generated due to the tag @ejb.bean declared at class level, as shown below.

```
<entity>
  <ejb-name>Item</ejb-name>
  <jndi-name>ItemBean</jndi-name>
  <local-jndi-name>ItemLocal</local-jndi-name>
```

Note : @ejb.bean tag has been covered in previous chapters.

Item Bean functionality is complete in and ready for deployment.

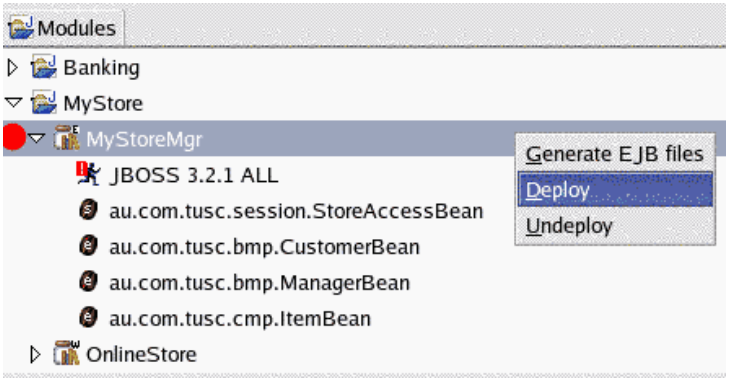
Deploy Item Bean :

Go to Lomboz J2EE View > expand node MyStore > expand MyStoreMgr > select Jboss 3.2.1 ALL.

Right click > select Debug Sever on the pop up menu.

Note : This is to start your server, if you are already running your server then skip these steps and go to the next one.

Go to MyStoreMgr node in LombozJ2EE view > right click > select Deploy on the pop up menu as shown below.



Note : All these steps have been detailed in previous chapters (1 and 3), so, please refer to them.

Messages in the console will indicate whether deployment of your bean has been successful or not.

Now, lets modify StoreAccessBean to invoke methods on ItemBean.

Add a field to store our Item reference (obtained from JNDI lookup).

private ItemLocalHome itemLocalHome;

In the ejbCreate method store this reference in the itemLocalHome variable by invoking the getLocalHome static method in the ItemUtil class as shown in this code snippet below from StoreAccess Bean.

```
/**
 * The  ejbCreate method.
 * @ejb.create-method
 **/
public void ejbCreate () throws  javax.ejb.CreateException {

    System.out.println (" Entering  StoreAccessBean.ejbCreate() ");
    try {
        customerLocalHome = CustomerUtil.getLocalHome();
        managerLocalHome = ManagerUtil.getLocalHome();
        itemLocalHome = ItemUtil.getLocalHome();
    }
    catch (Exception e) {
         e.printStackTrace();
    }
    System.out.println (" Leaving  StoreAccesBean.ejbCreate() ");
}
```

Add a Business Method to StoreAccess :

Add another business method to the StoreAccess Bean which will invoke this business method on our Item Bean.

Now, add a business method with this signature:

public ItemData getItemData(String itemID)

.. with Interface type as Remote. As managers will log on to MyStore with username, once authenticated they will be identified by their userid. They can then retrieve their account details from MyStore using that userid. A Manager can invoke methods on the Item Bean to examine the inventory of MyStore.

Note : Steps to create business methods are covered in previous chapters.

Now invoke one of Item's finder methods via the reference variable we have created in the ejbCreate method.

ItemLocal item = itemLocalHome.findByPrimaryKey(itemID)

Now invoke the business method of Customer on this reference.

ItemData myItem = item.getItemData()

Code snippet for this business method is shown below.

```
/**
 * Returns object ItemData
 * @ejb.interface-method
 * tview-type="remote"
 **/
public au.com.tusc.cmp.ItemData getItemData(String itemID){

    System.out.println (" Entering StoreAccessBean.getItemData() ");
    ItemData myItem = null;

    try {
    ItemLocal item =  itemLocalHome.findByPrimaryKey(itemID);

      if (item != null ) {
        myItem = item.getItemData();
       }
    } catch (Exception e) {
         System.out.println (" Error in StoreAccessBean.getItemData() " + e);
    }
    System.out.println (" Leaving StoreAccessBean.getItemData() ");
    return myItem;
}
```

Add another business method to StoreAccess Bean.

Add a business method with this signature:

public java.util.ArrayList getOutOfStockItems()

.. with Interface type as Remote. This will return the items which are out of stock in MyStore.

Create two variables of type Collection and ArrayList respectively, as the finder method for Items returns a Collection and this method will return an ArrayList, after populating items which are out of stock from the returned Collection.

Collection items = null;
ArrayListItemsOutOfStock = null;

Now invoke one of the finder methods of Item on the reference variable we have created in the ejbCreate method.

items = itemLocalHome.findByOutOfStock()

Now iterate through the collection of out of stock items and add to the ArrayList.

ItemLocal myItemLocal = (ItemLocal) iterate.next();
itemsOutOfStock.add(myItemLocal.getItemData());

Code snippet for this business method is shown below.

```
/**
 * Returns ArrayList of Items which are out of Stock.
 * @ejb.interface-method
 *  tview-type="remote"
 **/
public java.util.ArrayList getOutOfStockItems( ){

    System.out.println (" Entering StoreAccessBean.getItemsOutOfStock() ");
    Collection items = null;
    ArrayList itemsOutOfStock = new ArrayList();

    try {
        items  =   itemLocalHome.findByOutOfStock();
        Iterator iterate = items.iterator();
            while (iterate.hasNext()) {
                ItemLocal myItemLocal = (ItemLocal) iterate.next();
                itemsOutOfStock.add(myItemLocal.getItemData());
            }
        } catch (Exception e) {
            System.out.println (" Error in StoreAccessBean.getItemsOutOfStock() " + e);
        }
        System.out.println (" Leaving StoreAccessBean.getItemsOutOfStock() ");
        return itemsOutOfStock;
    }
```

Add another business method to StoreAccess Bean.

Add a business method with this signature:

public java.util.ArrayList getItemBySupplier(String supplierID)

.. with Interface type as Remote. This will return the items which are provided to MyStore by a given supplier.

Create two variables of type Collection and ArrayList respectively as the finder method for Items returns a Collection and this method will return ArrayList, after populating items which are supplied by a particular supplier from the returned Collection.

Collection suppliedItems = null;
ArrayList itemsBySupplier = null;

Now invoke one of the finder methods of Item on the reference variable we have created in the ejbCreate method.

suppliedItems = itemLocalHome.findBySupplierID(supplierID)

Now iterate through the collection of items for this supplier and add to the ArrayList.

ItemLocal myItemsLocal = (ItemLocal) iterate.next();
itemsBySupplier.add(myItemsLocal.getItemData());
Code snippet for this business method is shown below.

```
/**
 * Returns ArrayList of Items supplied by a supplier
 * @ejb.interface-method
 *  tview-type="remote"
 **/
public java.util.ArrayList getItemsBySupplier(String supplierID ){

    System.out.println (" Entering StoreAccessBean.getItemsBySupplier() ");
    Collection suppliedItems = null;
    ArrayList itemsBySupplier = new ArrayList();

    try {
        suppliedItems  =   itemLocalHome.findBySupplierID (supplierID);
        Iterator iterate = suppliedItems.iterator();
            while (iterate.hasNext()) {
                ItemLocal myItemsLocal = (ItemLocal) iterate.next();
                itemsBySupplier.add(myItemsLocal.getItemData());
            }
        } catch (Exception e) {
            System.out.println (" Error in StoreAccessBean.getItemsBySupplier() " + e);
        }
        System.out.println (" Leaving StoreAccessBean.getItemsbySupplier() ");
        return itemsBySupplier;
    }
```

Now all the methods in StoreAccess Bean for accessing Item's business methods have been added. The only remaining bit is the deployment descriptors required for linking/referencing of StoreAccess and Item Bean. So we will the add two tags shown below.

```
/**
 * @ejb.ejb-ref ejb-name="Item"
 *    view-type="local"
 *    ref-name="ItemLocal"
 *
 * @jboss.ejb-ref-jndi ref-name="ItemLocal"
 *    jndi-name="ItemLocal"
 **/

public abstract class StoreAccessBean implements SessionBean {
```

First add the tag shown below at class level in StoreAccess Bean.

@ejb.ejb-ref ejb-name="Item"
view-type="local"
ref-name="ItemLocal"

This tag will generate deployment descriptors in 'ejb-jar.xml', as StoreAccessBean has to know which bean it is referring to, what is its view-type and ref-name. This will generate these descriptors as shown below.

Note : View type is local as both are in the same Java Virtual Machine, otherwise it would be Remote. Secondly ref-name is generated as ItemLocalHome, as we are using that rather than ItemHome (which was also generated, but is used in the Remote case).

```
<ejb-name>StoreAccess</ejb-name>

<home>au.com.tusc.session.StoreAccessHome</home>
<remote>au.com.tusc.session.StoreAccess</remote>
<local-home>au.com.tusc.session.StoreAccessLocalHome</local-home>
<local>au.com.tusc.session.StoreAccessLocal</local>
<ejb-class>au.com.tusc.session.StoreAccessSession</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>

<ejb-local-ref >
  <ejb-ref-name>ejb/CustomerLocal</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>au.com.tusc.bmp.CustomerLocalHome</local-home>
  <local>au.com.tusc.bmp.CustomerLocal</local>
  <ejb-link>Customer</ejb-link>
</ejb-local-ref>
<ejb-local-ref >
  <ejb-ref-name>ejb/ManagerLocal</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>au.com.tusc.bmp.ManagerLocalHome</local-home>
  <local>au.com.tusc.bmp.ManagerLocal</local>
  <ejb-link>Manager</ejb-link>
</ejb-local-ref>
<ejb-local-ref >
  <ejb-ref-name>ejb/ItemLocal</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>au.com.tusc.cmp.ItemLocalHome</local-home>
  <local>au.com.tusc.cmp.ItemLocal</local>
  <ejb-link>Item</ejb-link>
</ejb-local-ref>

<resource-ref >
  <res-ref-name>jdbc/DefaultDS</res-ref-name>
  <res-type>javax.sql.Datasource</res-type>
```

Now add a second tag (shown below) at class level in StoreAccess Bean.

@jboss.ejb-ref-jndi ref-name="ItemLocal"
jndi-name="ItemLocal"

This tag will generate deployment descriptors in 'jboss.xml', as the application server has to know what jndi-name the Item bean has been registered with. This will generate these descriptors as shown below.

Note : Ref-name and jndi-name are used for bean as local (in same JVM).

```
<session>
  <ejb-name>StoreAccess</ejb-name>
  <jndi-name>StoreAccessBean</jndi-name>
  <local-jndi-name>StoreAccessLocal</local-jndi-name>
  <ejb-local-ref>
    <ejb-ref-name>ejb/CustomerLocal</ejb-ref-name>
    <jndi-name>CustomerLocal</jndi-name>
  </ejb-local-ref>
  <ejb-local-ref>
    <ejb-ref-name>ejb/ManagerLocal</ejb-ref-name>
    <jndi-name>ManagerLocal</jndi-name>
  </ejb-local-ref>
  <ejb-local-ref>
    <ejb-ref-name>ejb/ItemLocal</ejb-ref-name>
    <jndi-name>ItemLocal</jndi-name>
  </ejb-local-ref>
  <resource-ref>
    <res-ref-name>jdbc/DefaultDS</res-ref-name>
    <jndi-name>java:/DefaultDS</jndi-name>
  </resource-ref>
</session>
```

Note : We can see in the code snippet above the deployment descriptors generated by tag @jboss. For the view type 'local' it generates incorrect deployment descriptors, as discussed in the previous chapter. So every time we use this tag we have to change the <ejb-ref> to <ejb-local-ref> before deployment. ***Caution here****, that you do this change manually when you finally finish regenerating your EJB classes, because every time you regenerate your classes, 'jboss.xml' will be overwritten.*

Now our Item Bean is complete after these changes, so deploy your bean again now, from the Lomboz J2EE View, as per the steps shown above and in previous chapters. Messages will appear in the console showing the status of deployment.

Once the bean is deployed successfully, create a test client which will invoke the loginUser method on StoreAccess Bean , getCustomerData on Customer Bean, getManagerData on Manager Bean and getOutOfStockItems on Item Beam.

Create your Test Client :

Go to Project MytStore node > select src node and expand it > select au.com.tusc.client package > right click.

Select New on the pop up menu > select Lomboz EJB Test Client Wizard.

Select package name au.com.tusc.client, name as SessionCMPClient and select Ejb Home as au.com.tusc.session.StoreAccessHome and Ejb Interface as au.com.tusc.session.StoreAccess.

This will generate required methods for you in your SessionCMPClient class and you only have to invoke loginUser, getCustomerData, getManagerData (Manager Bean was developed as part of an exercise in the previous chapter) and getOutOfStockItems as shown below.

Now, the last step is to write the code for your client.

In order to access out of stock items we need an Iterator and an ArrayList for items. So, declare two variables of these respective types, and import the packages for these types (which are java.util.ArrayList and java.util.Iterator).

```
Iterator itemsIterator = null;
ArrayList items = null;
```

And now add these lines of code to invoke the methods mentioned above.

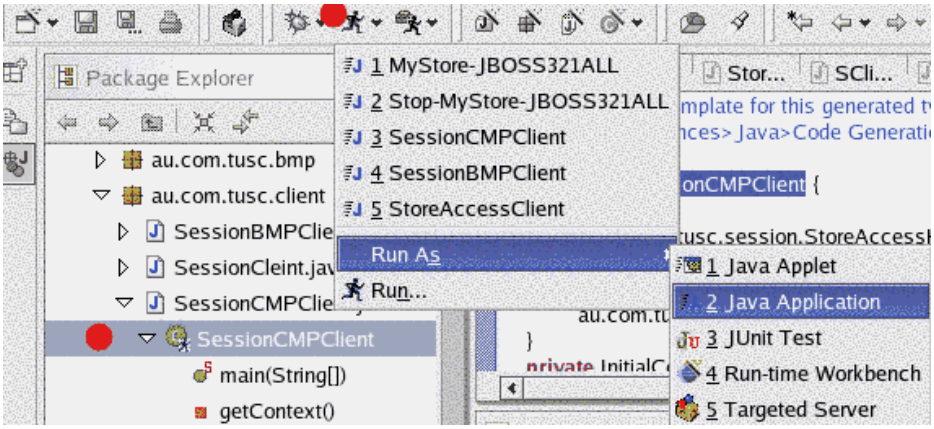
```
System.out.println("Request from client : ");
String userID = myBean.loginUser("ANDY","PASSWD");
System.out.println("Reply from Server: Your userid is " + userID );
CustomerData cd = myBean.getCustomerData(userID);
System.out.println ("Andy your details with MyStore are " + cd );
String mgrID =   myBean.loginUser("RUSTY","PASSWD");
System.out.println("Reply from Server: Your mgrid is " + mgrID );
ManagerData md = myBean.getManagerData(mgrID);
System.out.println ("Rusty your details with MyStore are  " + md );
System.out.println("Manager Request : List items out of stock ");

items = myBean.getOutOfStockItems();
itemsIterator = items.iterator();
System.out.println("List Of Out Of stock Items ");

while ( itemsIterator.hasNext() ) {
    ItemData itemData= ( ItemData ) itemsIterator.next();
    System.out.println ("Item Data " + itemData );
```

}

Test your Client :

Now, in order to test your client, Select SessionCMPClient node > Go at top level menu and select the icon with the 'Running Man'.

On that select 'Run as' > select Java Application, as shown below.



Now on your console, if you get a reply saying two items out of stock which are 'CALCULATOR' and 'CLOCK', then your call is successful as shown below.

```
Request from client :
Reply from Server: Your userid is U2
Andy your details with MyStore are  {customerID=C2 userID=U2 firstName=ANDREW lastName=HENDRY
Reply from Server: Your mgrid is U6
Rusty your details with MyStore are  {managerID=M1 userID=U6 firstName=STEFFI lastName=LINDSAY
Manager Request : List items out of stock
List Of Out Of stock Items
Item Data {itemID=I1 supplierID=S1 description=CALCULATOR quantity=0 price=45.0}
Item Data {itemID=I4 supplierID=S1 description=CLOCK quantity=0 price=65.0}
```

Exercise :

Now, here is an exercise for you. In order to proceed further, implement Supplier as a CMP Entity Bean. The tasks are given below:

1. Create a CMP Bean named Supplier under package au.com.tusc.cmp.
2. Implement the ejbCreate method, with all attributes passed as arguments and then assigned to attributes using mutator methods.
3. Add a find method named findUserID with query and signature:

   query "SELECT OBJECT(b) FROM MyStoreSupplier as b where b.userID = ?1"

   method au.com.tusc.cmp.SupplierLocal findUserID(java.lang.String userID)

   Note : The method signature is find<cmp attribute> instead of findByPrimaryKey, because finder methods for non-key CMP attributes use this convention. Accordingly the return type for finder methods will be either Collection or <entity type>, as specified in the EJB specification, sections 10.5.6 and 10.5.2 respectively.
4. Add a business method to get supplier details with signature:

   public SupplierData getSupplierData()
5. Add a business method to get request items from various suppliers with signature:

   public void requestItem(String itemID, Integer quantity)
6. Add callback methods, required for setting/unsetting bean context with signatures:

   public void setEntityContext(EntityContext ctx)

   public void unsetEntityContext()
7. Deploy the Supplier Bean.

8. Add a field to StoreAcess Bean to store its reference:

   private SupplierLocalHome supplierLocalHome

9. In the ejbCreate method of StoreAccess Bean store a reference in supplierLocalHome variable by invoking the getLocalHome static method in supplierUtil.

12. Add a business method to StoreAccess Bean with signature:

    public ItemData getItemData(String itemID)

13. Add the following tags for deployment at class level for linking/referencing Supplier.

    1. @ejb.ejb-ref ejb-name="Supplier"
       view-type="local"
       ref-name="SupplierLocal"

    2. @jboss.ejb-ref-jndi ref-name="SupplierLocal"
       jndi-name="SupplierLocal"

14. Test your Supplier Bean by running your Test Client created for Item named SessionCMPClient.

Note : All these steps are same as were done for Item. Implement this Bean which will be used in subsequent chapters.

In case you have difficulty, we have provided a SupplierBean class, modified StoreAccessBean class and SesiomCMPClient class. You can download these files under downloads below.

Downloads :

SupplierBean

StoreAccessBean

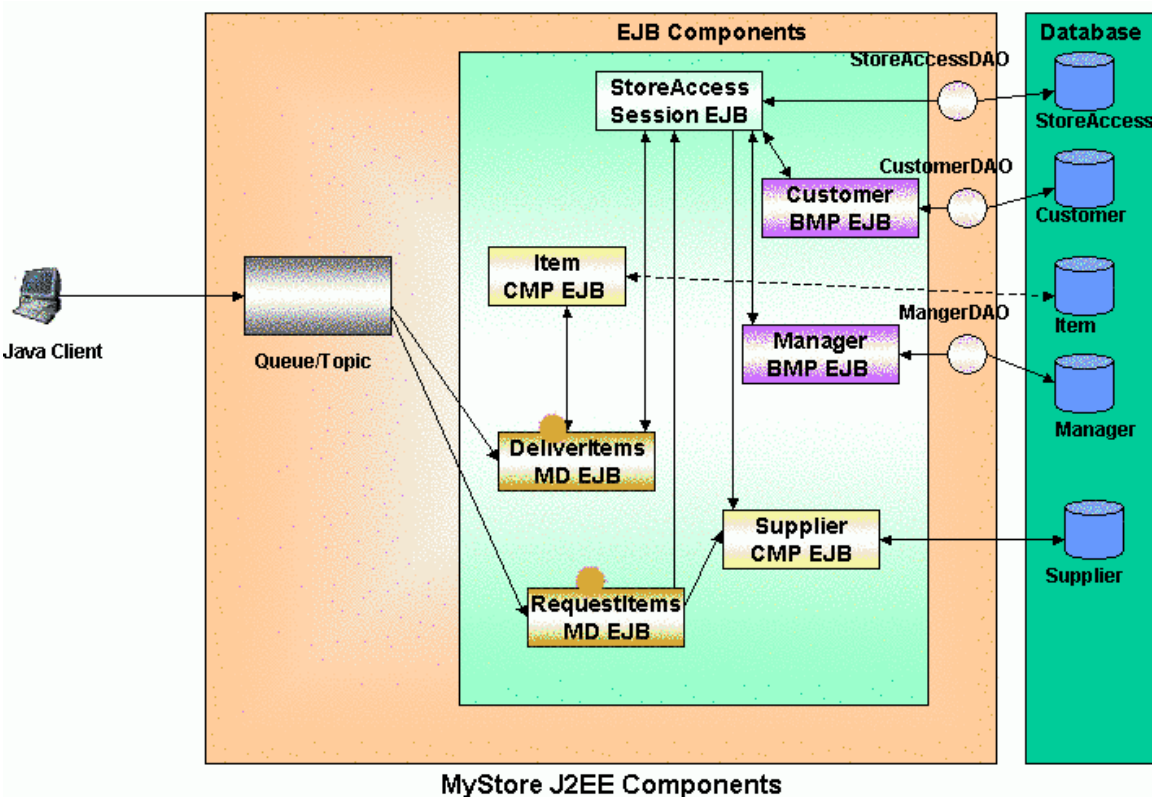
SessionCMPClient

Prev TOC Next

Chapter 7.

Creating a Message Driven Bean

This chapter covers how to create a Message Driven Bean (MDB) EJB component. We will create two MDB beans, DeliverItems and RequestItems as shown below. The DeliverItems bean will replenish the stocks of various items within MyStore, and the RequestItems bean will send requests to various suppliers to deliver items which are out of stock. The MyStore manager will issue/send this request.



Note : Both message-driven beans access the StoreAccess bean through its remote interface, even though the StoreAcessBean is in the same JVM. This is because we have implemented StoreAccessBean as a Remote Bean, so it only exposes its remote interface. However, in accessing the Manager and Item beans, which are also used by these message-driven beans we can use their local interfaces as they are in the same JVM, and we have exposed their local interfaces.

Tasks :

1. Create a MD bean named RequestItems under package au.com.tusc.mdb.
2. Create an Immutable Value Object named RequestItem under package au.com.tusc.mdb. Add attributes and implement their accessor and mutator methods. The attributes are:

   private String username

   private String passwd

   private String itemID

   private int quantity

3. Implement the onMessage method.
4. Deploy the RequestItems Bean.
5. Create your test client named RequestMDBClient under package au.com.tusc.mdb.
6. Add a method named testMDBBean with the following signature and implement it.

   public void testMDBBean

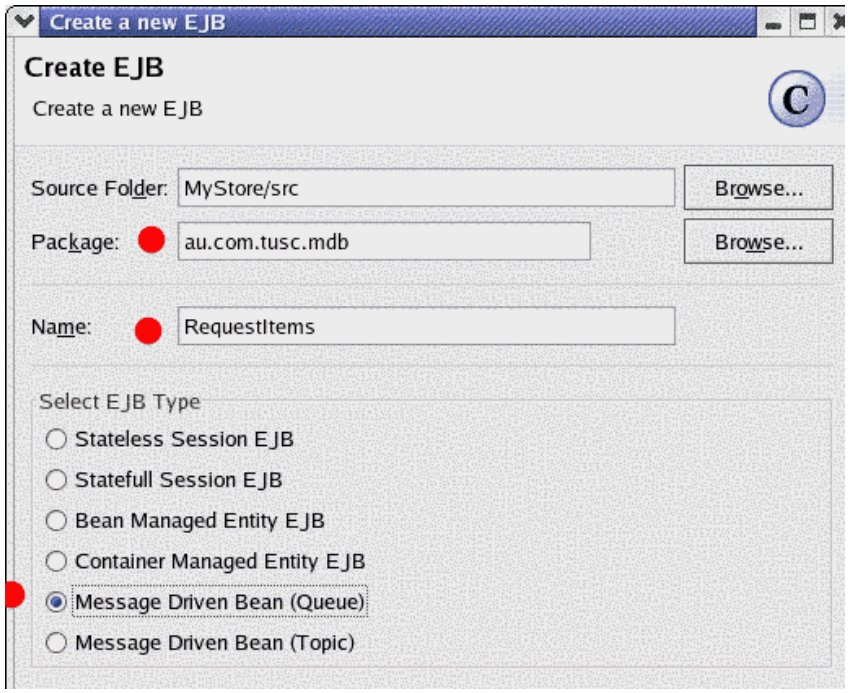
7. Run your client and test the bean.

Create RequestItems MDB Bean :

Go To Package Explorer > Expand Mystore (project) node > select src, right click and a menu will pop up.

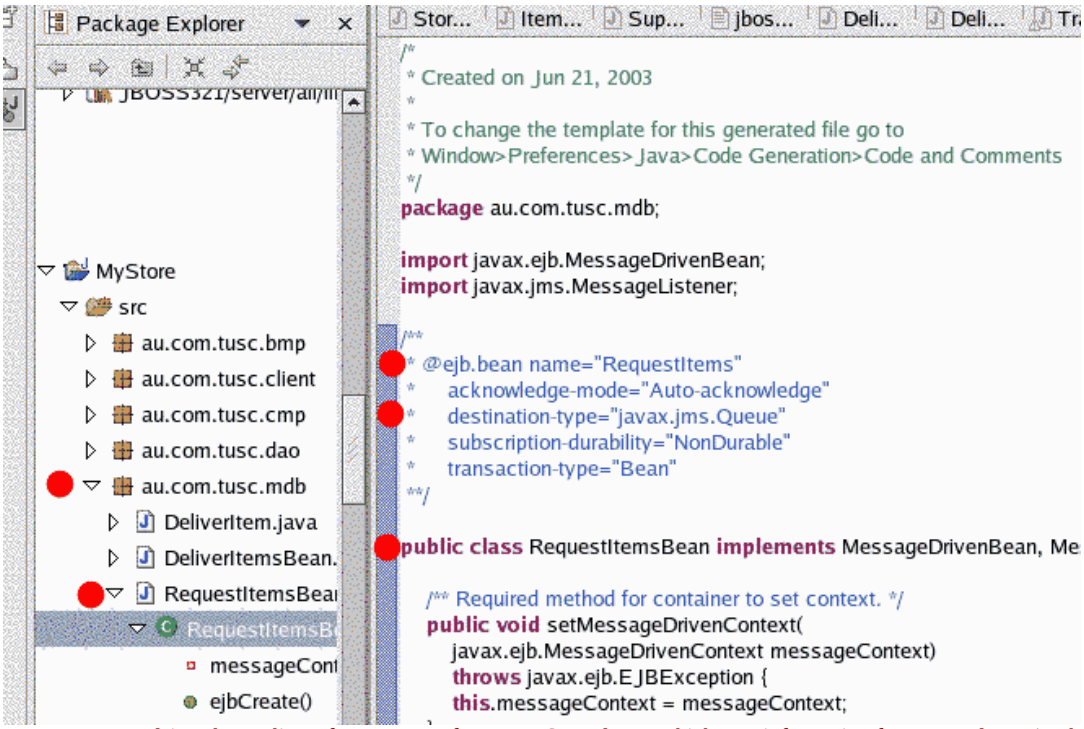
On the pop up menu > New > Lomboz EJB Creation Wizard.

Enter package name au.com.tusc.mdb, bean name RequestItems and select bean type as Message Drive Bean (Queue) as shown below.



Press Finish.

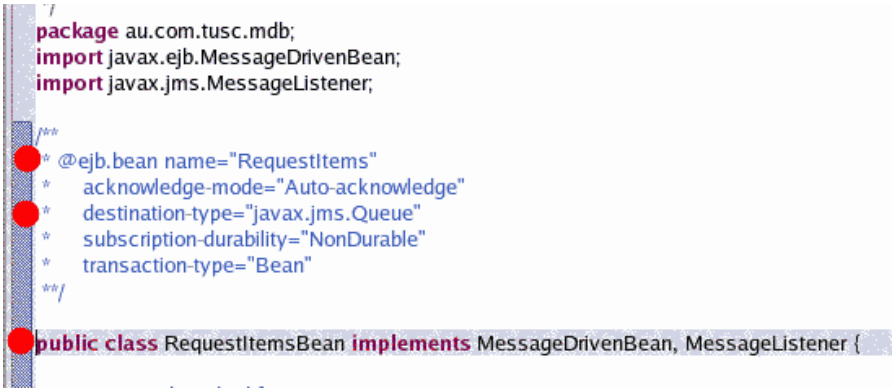
This will create a package named au.com.tusc.mdb under src and RequestItemsBean within that package as shown below.



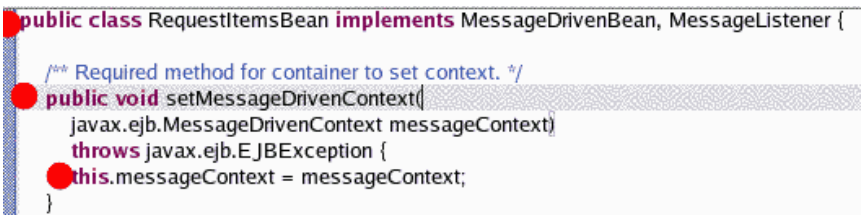
Note : Message-driven beans listen for messages from a JMS Producer, which gets information from a producer (perhaps another bean) and transfers it to the relevant Consumer bean. Since it is only responsible for processing such messages, it doesn't need any helper classes such as Remote and RemoteHome interfaces, Util classes, DAO class, etc. like our previous types of beans. The only helper classes we have to create are immutable value objects, which will be responsible for holding the information extracted from messages and then transferred to the bean(s).

Let's examine what methods and tags are generated by the EJB creation wizard..

It creates one @ejb.bean tag which assigns the name, transaction type, destination type and some other properties as shown below.



Unlike our previous beans it has a setMessageContext method for setting the context.



It has ejbCreate and ejbRemove methods like the other types of beans, as shown below.

```
/**
 * Required creation method for message-driven beans.
 * @ejb.create-method
 */
public void ejbCreate() {
    // no specific action required for message-driven beans
}

/** Required removal method for message-driven beans. */
public void ejbRemove() {
    messageContext = null;
}
```

It has a new method named onMessage which is the one of significance to us, where all the business logic will be written (as shown below).

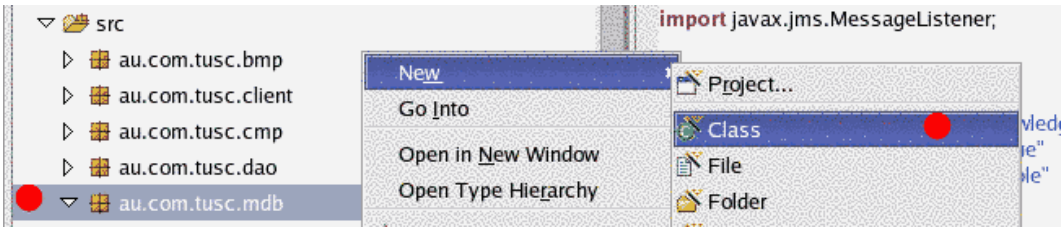
```
/**
 * This method implements the business logic for the EJB.
 * <p>Make sure that the business logic accounts for asynchronous message processing.
 * For example, it cannot be assumed that the EJB receives messages in the order they were
 * sent by the client. Instance pooling within the container 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.
 */
public void onMessage(javax.jms.Message message) {
    System.out.println("Message Driven Bean got message " + message);
    // do business logic here

}
```

Once a message is received from the JMS producer as a Message object, its data is extracted and filled into the immutable value object and then transferred to the relevant bean. This is covered later on in this chapter.

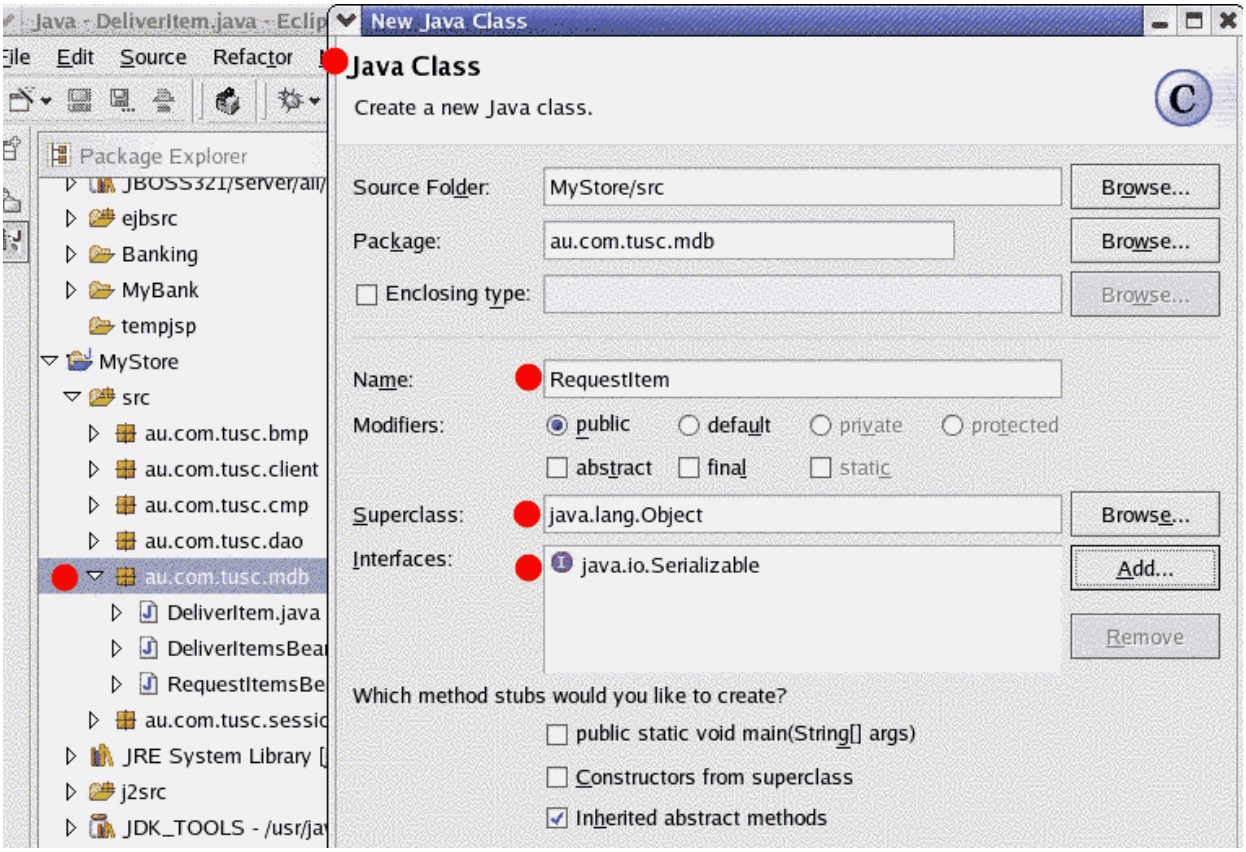
Now, before we add any functionality, create a class or immutable value object for extracting information from the message.

Create Immutable Value Object RequestItem :

Go to src > under package au.com.tusc.mdb > New >Class.



The java class wizard will pop up > Add class name as RequestItem, Superclass as java.lang.Object and Interfaces as Serializable as as shown below in figure.



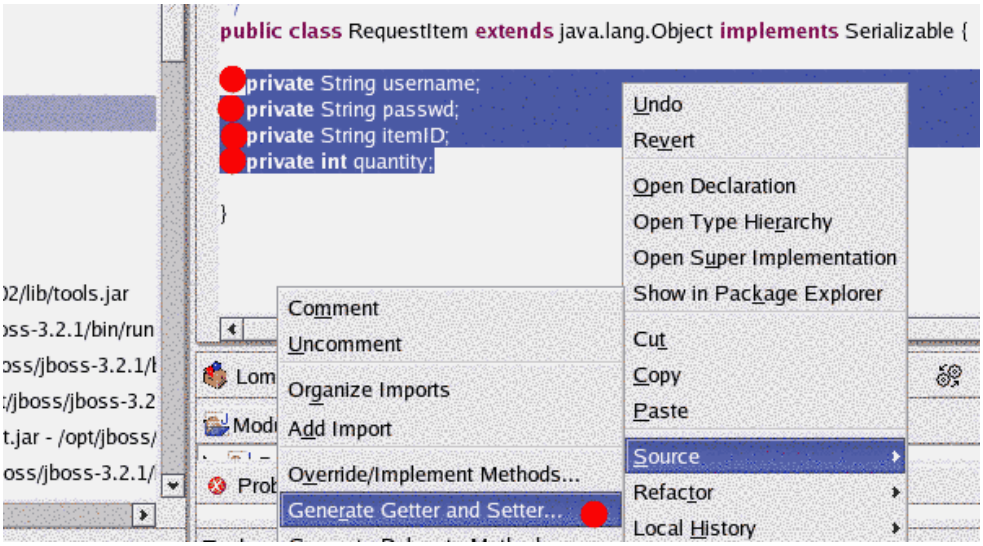
This will generate a RequestItem class under au.com.tusc.mdb as shown below.

Add the following attributes to the RequestItem class as shown below.

private String username;
private String passwd;
private String itemID;
private int quantity;

Add accessor and mutator method for these attributes.

Select all attributes > Right click> Select source > Generate Getter and Setter as shown below.



Add a constructor for the class which has parameters with the same type as the attributes and assigns them to the attributes as shown below.

```
public RequestItem(String username, String passwd, String temID, int quantity) {
    super();
    setUsername(username);
    setPasswd(passwd);
    setItemID(itemID);
    setQuantity(quantity);
}
```

The RequestItem constructor is complete. Now to implement the onMessage method in the RequestItem Bean.

Implement onMessage Method :

This method is responsible for extracting the information from the message and transferring it to the relevant bean.

The MyStore Manager will check for items which are out of stock and generate a request to suppliers specifying the itemID and quantity required.

First import the packages java.util.ArrayList and java.util.Iterator .

Add the following variables to store references.

```
ArrayList outOfStockItems = null;
Iterator itemsIterator = null;
private StoreAccessHome storeAccess = null;
private SupplierLocalHome suppLocalHome = null;
private ItemLocalHome itemLocalHome = null;
```

Extract the data from the message into the immutable value object as shown below.

```
RequestItem ri =  (RequestItem) ((ObjectMessage) message).getObject();
```

Add the lines of code shown below, so that the manager can login.

```
StoreAccess access = StoreAccessUtil.getHome().create();
String mgrAccessID = access.loginUser(ri.getUsername(),ri.getPasswd());
```

Get items with zero quantity (no stock), by invoking getOutOfStockItems() on the StoreAccess Bean (which returns a Collection).

```
outOfStockItems = access.getOutOfStockItems();
```

Now, iterating over each item, get the supplierId associated with it by invoking the finder methods on the Supplier Bean, finally sending the required message to that supplier by invoking the business method requestItem() on the Supplier Bean.

```
itemsIterator = outOfStockItems.iterator();

while ( itemsIterator.hasNext() ) {
        ItemData itemData= ( ItemData ) itemsIterator.next();
        String  suppID = itemData.getSupplierID();
        SupplierLocal  supplier = this.suppLocalHome.findByPrimaryKey(suppID);
        Integer  quantity = new Integer (ri.getQuantity());
        String  itemID = ri.getItemID();
        supplier.requestItem(  itemID, quantity);
}
```

Code snippet of onMessage is shown below.

```
public void onMessage(javax.jms.Message message) {
    System.out.println("Message Driven Bean got message " + message);
    // do business logic here
    System.out.println ("Entering RequestItemsBean.onMessage()");
    ArrayList outOfStockItems = null;
    Iterator itemsIterator = null;
    try {
        if (message instanceof ObjectMessage) {

        RequestItem ri =  (RequestItem) ((ObjectMessage) message).getObject();
        StoreAccess access = StoreAccessUtil.getHome().create();
        String mgrUserID = access.loginUser(ri.getUsername(),ri.getPasswd());
        outOfStockItems = access.getOutOfStockItems();
        itemsIterator = outOfStockItems.iterator();
        this.suppLocalHome = SupplierUtil.getLocalHome();
          System.out.println("List Of Out Of stock Items ");
          while ( itemsIterator.hasNext() ) {
            ItemData itemData= ( ItemData ) itemsIterator.next();
            String suppID = itemData.getSupplierID();
            SupplierLocal supplier = this.suppLocalHome.findByPrimaryKey(suppID);
            Integer quantity = new Integer (ri.getQuantity());
            String itemID = ri.getItemID();
            supplier.requestItem( itemID, quantity);
          }
         }
        }catch (Exception e) {
              System.out.println ("In RequestItems.onMessage  " +  e.getMessage() );
        }
        System.out.println("Leaving RequestItemsBean.onMessage()");
}
```

Now our bean is complete, and all that remains are the deployment descriptors for the bean.

Deploy RequestItems Bean :

In order to deploy this bean we have to add a few deployment descriptors. As shown below, five tags have been added.

```
/**
 * @ejb.bean name="RequestItems"
 *     acknowledge-mode="Auto-acknowledge"
 *     destination-type="javax.jms.Queue"
 *     subscription-durability="NonDurable"
 *     transaction-type="Bean"
 *
 * @ejb.ejb-ref
 *      ejb-name="StoreAccess"
 *      view-type="remote"
 *      ref-name="StoreAccess"
 *
 * @ejb.ejb-ref
 *      ejb-name="Supplier"
 *      view-type="local"
 *      ref-name="SupplierLocal"
 *
 * @jboss.ejb-ref-jndi ref-name="SupplieLocal"
 *      jndi-name="SupplierLocal"
 *
 * @jboss.ejb-ref-jndi ref-name="StoreAccess"
 *      jndi-name="StoreAccessBean"
 *
 * @jboss.destination-jndi-name
 *      name="queue/MdbQueue"
 *
 *
 **/

public class RequestItemsBean implements MessageDrivenBean, MessageListener {
```

First add the tag shown below at class level in the RequestItems Bean, to obtain a reference to StoreAccessBean, so that methods can be invoked on that bean.

```
@ejb.ejb-ref
ejb-name="StoreAccess"
view-type="remote"
```

ref-name="StoreAccess"

This tag will generate deployment descriptors in ejb-jar.xml when you generate your ejb classes, as this message bean has to authenticate, before transferring information to the relevant bean. This will generate these descriptors as shown below.

```
<!-- Message Driven Beans -->
<message-driven >
  <description><![CDATA[]]></description>

<ejb-name>RequestItems</ejb-name>

  <ejb-class>au.com.tusc.mdb.RequestItemsBean</ejb-class>

  <transaction-type>Bean</transaction-type>
  <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
  <message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
    <subscription-durability>NonDurable</subscription-durability>
  </message-driven-destination>

<ejb-ref >
  <ejb-ref-name>ejb/StoreAccess</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>au.com.tusc.session.StoreAccessHome</home>
    <remote>au.com.tusc.session.StoreAccess</remote>
  <ejb-link>StoreAccess</ejb-link>
  </ejb-ref>
```

Add another tag as shown below, to obtain a reference to the Supplier Bean from this bean.

@ejb.ejb-ref
ejb-name="Supplier"
view-type="local"
ref-name="SupplierLocal"

This tag will generate also generate descriptors in ejb-jar.xml when you generate your ejb classes, as this message bean transfers information to the Supplier bean. The following descriptors are generated as shown below.

```
<message-driven >
    <description><![CDATA[]]></description>

    <ejb-name>RequestItems</ejb-name>

  <ejb-class>au.com.tusc.mdb.RequestItemsBean</ejb-class>

    <transaction-type>Bean</transaction-type>
    <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
    <message-driven-destination>
      <destination-type>javax.jms.Queue</destination-type>
      <subscription-durability>NonDurable</subscription-durability>
    </message-driven-destination>

    <ejb-ref >
      <ejb-ref-name>ejb/StoreAccess</ejb-ref-name>
      <ejb-ref-type>Session</ejb-ref-type>
      <home>au.com.tusc.session.StoreAccessHome</home>
      <remote>au.com.tusc.session.StoreAccess</remote>
      <ejb-link>StoreAccess</ejb-link>
    </ejb-ref>

  <ejb-local-ref >
    <ejb-ref-name>ejb/SupplierLocal</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <local-home>au.com.tusc.cmp.SupplierLocalHome</local-home>
      <local>au.com.tusc.cmp.SupplierLocal</local>
    <ejb-link>Supplier</ejb-link>
    </ejb-local-ref>

  </message-driven>
```

Note : Another descriptor which has been generated is <ejb-name>, which is generated by the tag @ejb.bean that was added by the EJB creation wizard.

```
@ejb.bean name="RequestItems"
   acknowledge-mode="Auto-acknowledge"
   destination-type="javax.jms.Queue"
   subscription-durability="NonDurable"
   transaction-type="Bean"
```

This tag generates the following descriptors in ejb-jar.xml:

```
<message-driven >
  <description><![CDATA[]]></description>

  <ejb-name>RequestItems</ejb-name>

  <ejb-class>au.com.tusc.mdb.RequestItemsBean</ejb-class>

  <transaction-type>Bean</transaction-type>
  <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
  <message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
    <subscription-durability>NonDurable</subscription-durability>
  </message-driven-destination>

  <ejb-ref >
    <ejb-ref-name>ejb/StoreAccess</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>au.com.tusc.session.StoreAccessHome</home>
```

Add another tag as shown below. This tag is JBOSS-specific, and is used to register a message-driven bean with a jndi-name, using the format "queue/name".

```
@jboss.destination-jndi-name
name="queue/MdbQueue"
```

This will generate the following deployment descriptors in jboss.xml:

```
<message-driven>
  <ejb-name>RequestItems</ejb-name>
  <destination-jndi-name>queue/MdbQueue</destination-jndi-name>
</message-driven>
```

Note : For further documentation on MDB development with JBOSS, please refer to the JBOSS documentation.

Add another tag as shown below, required by JBOSS, for finding the Supplier bean using its JNDI NAME.

```
@jboss.ejb-ref-jndi ref-name="SupplierLocal"
  jndi-name="SupplierLocal"
```

Note : As discussed in chapter5, this tag generates incorrect descriptors within jboss.xml. For view-type="local" it generates an <ejb-ref> tag rather than an <ejb-local-ref> tag.

Correct the following tags:

```
message-driven>
  <ejb-name>RequestItems</ejb-name>
  <destination-jndi-name>queue/MdbQueue</destination-jndi-name>
  <ejb-ref>
    <ejb-ref-name>ejb/SupplieLocal</ejb-ref-name>
    <jndi-name>SupplierLocal</jndi-name>
  </ejb-ref>
</message-driven>
```

should be <ejb-local-ref>

Find these tags in jboss.xml and change them to <ejb-local-ref> as shown below.

```
<message-driven>
  <ejb-name>RequestItems</ejb-name>
  <destination-jndi-name>queue/MdbQueue</destination-jndi-name>
  <ejb-local-ref>
    <ejb-ref-name>ejb/SupplieLocal</ejb-ref-name>
    <jndi-name>SupplierLocal</jndi-name>
  </ejb-local-ref>
</message-driven>
```

Now add this last tag as shown below for our StoreAccess Bean, to reference the StoreAccess bean using its JNDI NAME .

```
@jboss.ejb-ref-jndi ref-name="StoreAccess"
jndi-name="StoreAccessBean"
```

This tag will generate the following descriptors in jboss.xml:

```
<message-driven>
  <ejb-name>RequestItems</ejb-name>
  <destination-jndi-name>queue/MdbQueue</destination-jndi-name>
  <ejb-local-ref>
    <ejb-ref-name>ejb/SupplieLocal</ejb-ref-name>
    <jndi-name>SupplierLocal</jndi-name>
  </ejb-local-ref>
  <ejb-ref>
    <ejb-ref-name>ejb/StoreAccess</ejb-ref-name>
    <jndi-name>StoreAccessBean</jndi-name>
  </ejb-ref>
</message-driven>
```

Now our RequestItems Bean is complete, so now add your bean and generate your EJB classes.

Go to the RequestItemsBean node under au.com.tusc.mdb > right click > select Lomboz J2EE... > Add EJB to Module > Ok.

Go to the MyStoreMgr node in Package Explorer > right click > select Lomboz J2EE... > Generate EJB classes.

Note : Since you have generated your classes you have to again fix the incorrectly-generated deployment descriptors in jboss.xml, under <message-driven> and <session>.

Now, let's deploy our bean. Go to Lomboz J2EE View, start your server if it is not running, and deploy your bean.

Messages will appear in console showing deployment status.

Note : Steps for deploying beans have been detailed in previous chapters.

Once the bean is deployed successfully, create your test client.

Create Test Client :

Now, to create a test client in this case, the Test Client Wizard can't help us, because it requires a Home interface and EJB interface to be selected, whereas for Message Driven Beans there is no Home interface and EJB interface required (as discussed above).

So, write a class and the necessary methods to invoke operations on the RequestItems Bean.

Add a class named RequestMDBClient under the package au.com.tusc.mdb.

Add a method named getContext with the following signature:

private InitialContext getContext() **throws** NamingException

Add the following lines of code to get the instance of IntialContext as shown below.

```
public class DeliverMDBClient {

  private InitialContext getContext() throws NamingException {
      Hashtable props = new Hashtable();

      props.put(
          InitialContext.INITIAL_CONTEXT_FACTORY,
          "org.jnp.interfaces.NamingContextFactory");
      props.put(InitialContext.PROVIDER_URL, "jnp://127.0.0.1:1099");

      InitialContext initialContext = new InitialContext(props);
      return initialContext;
  }
```

Add a method named testMDBBean with the following signature.

public void testMDBBean()

Now implement this method, in which the following steps are needed:

1. **Add a Data Object which 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/MdbQueue".**
5. **Create the QueueConnection.**
6. **Create the QueueSender.**
7. **Create the QueueSession for the bean.**
8. **Create the Object Message and pass the Data Object in message.**
9. **Send the message.**
10. **Finally, commit the session, and then close both the session and the connection. A code snippet of the testMDBean method is shown below.**

```
public void testMDBBean() {

RequestItem ri = new RequestItem ("RUSTY","PASSWD", "I4", 30);
  try {
     System.out.println("Looking up the factory ");
     InitialContext ctx = getContext();
     QueueConnectionFactory factory = (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
     System.out.println("Looking up the queue ");
     Queue queue = (Queue) ctx.lookup("queue/MdbQueue");
     System.out.println("Creating the connection now...");
     QueueConnection connection = factory.createQueueConnection();
     System.out.println("Creating the session now...");
     QueueSession session = connection.createQueueSession(true, 1);
     System.out.println("Creating the sender now... ");
     QueueSender sender = session.createSender(queue);
     ObjectMessage message = session.createObjectMessage();
     System.out.println ("Setting the object in message now... ");
     message.setObject(ri);
     System.out.println ("Sending the message ");
     sender.send(message);
     System.out.println ("Shuting down");
     session.commit();
     session.close();
     connection.close();
     System.out.println ("Finished ");

  } catch (Exception e) {
     e.printStackTrace();
  }
}
```

Now your test client is complete, so let's test the client.

Test your Client :

To test your client, Select RequestMDBClient node > Go to the top level menu and select the 'Running Man' icon.

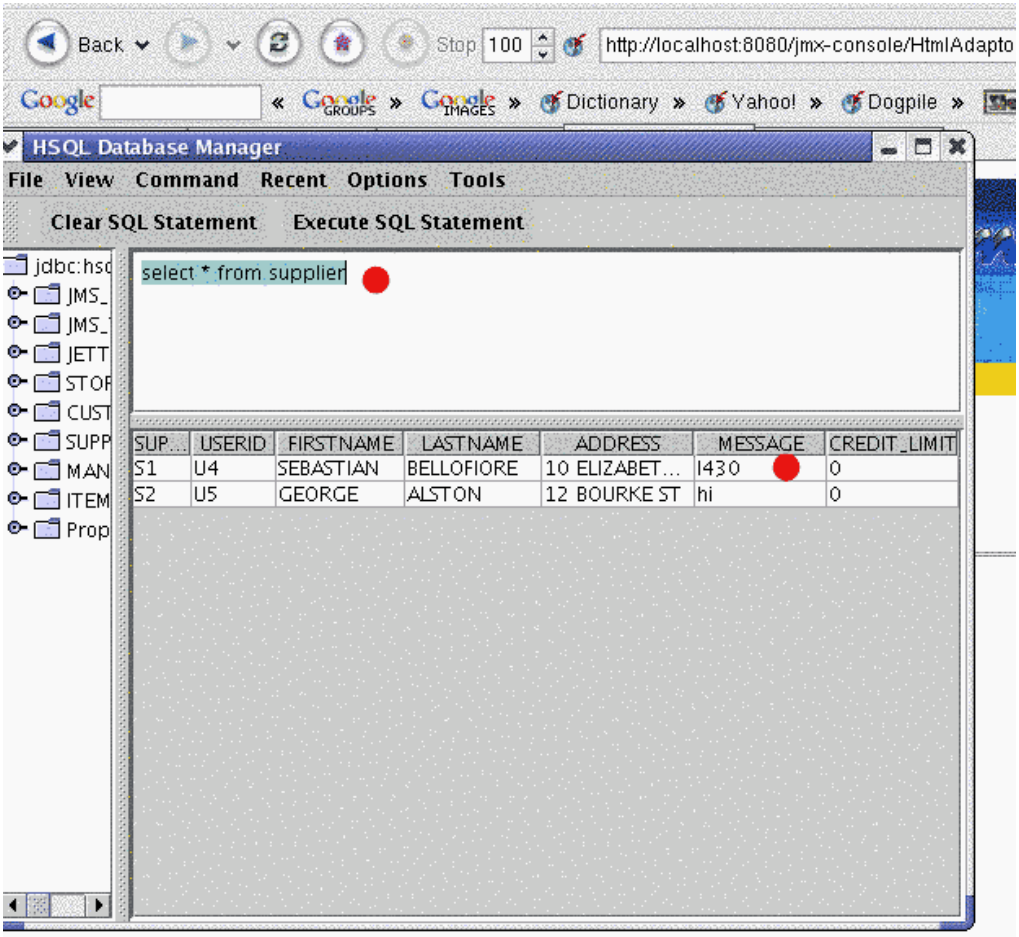
On that select 'Run as' > select Java Application.

Note : The steps to run test clients have been covered in previous chapters.

Now, under your console, you should get the following messages:

```
Looking up the queue
Creating the connection now..
Creating the session now..
Creating the sender
Setting the object in message now..
Sending the message now..
Shuting down...
Finished
```

This message on the console doesn't tell us whether the message has been delivered to relevant beans or not. In order to verify that, go to the database using JMX Management Console View > Hypersonic > Invoke Database Manager and then run a query on table 'supplier' to see if a message has been added for the supplier.



Note: Details on accessing this Database Manager are provided in chapter 1.

Since a supplier named Sebastian has received our message, our message has been transferred successfully.

Exercise :

Now, to progress further, please complete the following exercise. Implement DeliverItems as an MD bean. Detailed tasks are given below.

1. Create an MD Bean named DeliverItems under the package au.com.tusc.mdb.

2. Create an Immutable Value Object named DeliverItem under the package au.com.tusc.mdb. Add some attributes and implement their accessor and mutator methods. The attributes are:

   private String username

   private String passwd

   private String itemID

   private int quantity

3. Implement the onMessage method in your DeliverItems Bean.

   Add these two variables to store references.

   private StoreAccessHome storeAccess = null;
   private ItemLocalHome itemLocalHome = null;

   Extract the data from the message into your immutable value object as shown below.

   DelieverItem di = (DeliverItem) ((ObjectMessage) message).getObject();

   Get the references for the StoreAccess and Item beans.

   StoreAccess access = StoreAccessUtil.getHome().create();
   itemLocalHome = ItemUtil.getLocalHome();

   Invoke the loginUser method for supplier to get the Supplier's userid(accessID) and then find the Supplier's ID by invoking the getSupplierData method.

```
String suppAcessID = access.loginUser(di.getUsername(), di.getPasswd());
SupplierData sd = access.getSupplierData(suppAccessID);
String suppID = sd.getSupplierID();
```

If suppID is not equal to null, then invoke the finder method on the Item bean to get the details of the item to be delivered, by extracting itemID from message.

```
ItemLocal item = this.itemLocalHome.findByPrimaryKey(di.getItemID());
```

Get the supplier ID associated with the item found, so we can fill up the stock.

```
String itemSuppID = item.getSupplierID();
```

Compare itemSuppID and ItemID, if these are the same then fill the stock for that item by invoking the fillStock method in Item bean.

```
if ( suppID.equals(itemSuppID)) {
        System.out.println  ("Delivering items in store now... :");
        Integer  quantity = new Integer (di.getQuantity());
        item.fillStock(quantity);
        System.out.println  ("Stock of iten after dleivery is :" + item.getItemData());
}
```

4. Add the following tags for deployment at the class level for linking/referencing Supplier.

```
1. @ejb.ejb-ref
        ejb-name="StoreAccess"
        view-type="remote"
        ref-name="StoreAccess"

2. @ejb.ejb-ref
        ejb-name="Item"
        view-type="local"
        ref-name="ItemLocal"

3. @jboss.ejb-ref-jndi ref-name="ItemLocal"
        jndi-name="ItemLocal"

4. @jboss.ejb-ref-jndi ref-name="StoreAccess"
        jndi-name="StoreAccessBean"

  5. @jboss.destination-jndi-name
          name="queue/DelMdbQueue"
```

5. Deploy your DeliverItems Bean.

6. Create a test client named DeliverMDBClient under the package au.com.tusc.mdb.

7. Add a method named testMDBBean with the following signature.

   public void testMDBBean

8. Implement testMDBBean, for this a few steps have to be carried out.

   Add a Data Object which is to be sent as the message.

   Create Initial context.

   Create Connection factory.

   Using this context perform the JNDI lookup, where the lookup string is "queue/DelMdbQueue".

   Create QueueConnection.

   Create QueueSender.

   Create QueueSession for the bean.

   Create Object Message and pass the Data Object in the message.

   Send the message.

Finally, commit the session and close both the session and the connection.

5. Run your client and test the bean.

Note : All these steps are same as for the RequestItems Bean. This bean will be used in subsequent chapters.

In the event you run into some difficulty, we have provided the DeliverItemsBean, DeliverItem and DeliverMDBClient classes. You can download these files under downloads.

Downloads :

DeliverItemsBean

DeliverItem

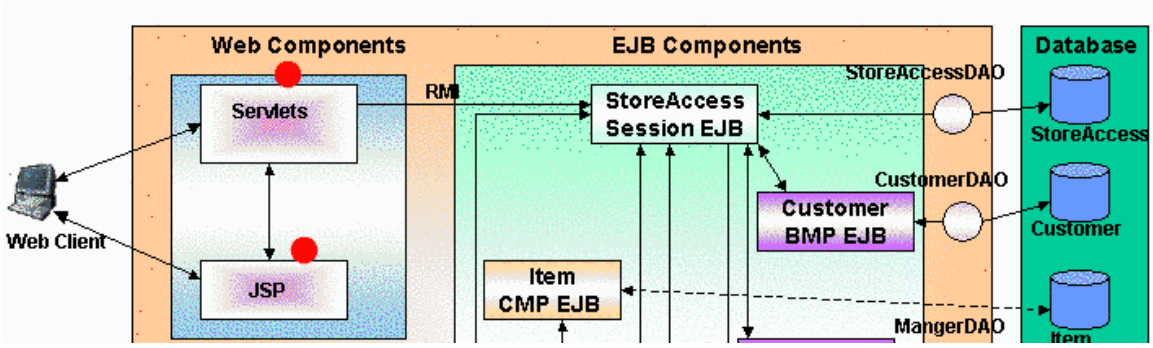
DeliverMDBClient

Prev TOC Next

Chapter 8.

Creating Web Clients

This chapter describes how to create web clients in the Client Tier/Presentation Tier to access or otherwise communicate with the Business Tier. Servlets and JSP pages are the two examples of web clients which will be created. In a typical Model View Controller Pattern, the Servlet acts as the Controller whilst JSP pages act as the View (the Model being the data, of course).



Tasks :

1. Create a Servlet named AccessController under the package au.com.tusc.servlet.
2. Add a business method to StoreAccess Bean named getAllItems() with the following signature

   public java.util.ArrayList getAllItems()
3. Implement the init method.
4. Implement doGet and doPost methods.
5. Add a method processRequest with the following signature

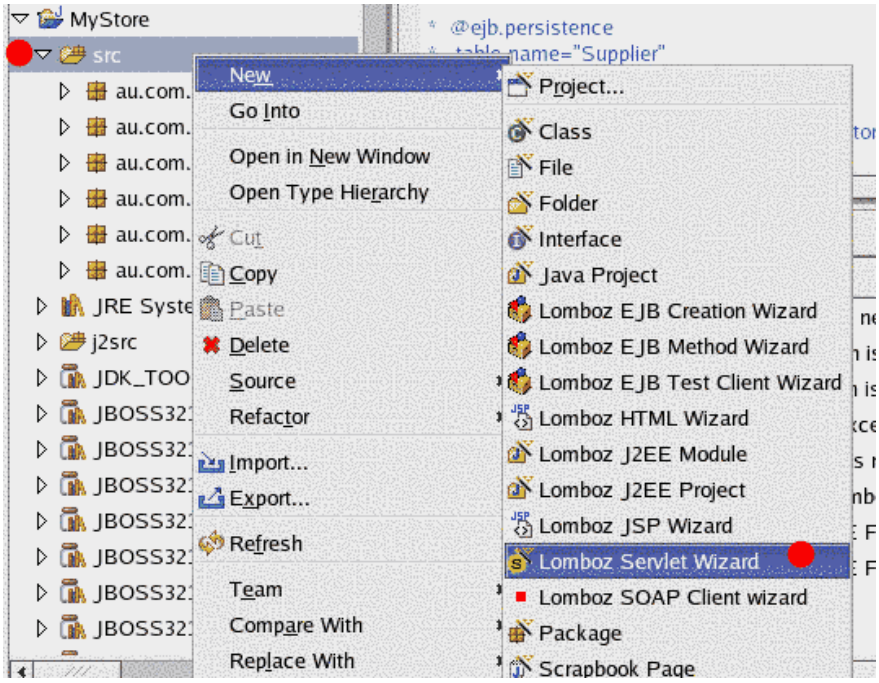
   protected void processRequest (HttpServletRequest request,HttpServletResponseresponse)

   throwsServletException, IOException
6. Implement the processRequest method.
7. Deploy the AccessController Servlet.
8. Test the AccessContoller Servlet.
9. Create a JSP Page named showItems.
10. Modify the method processRequest in Servlet AccessController.
11. Add HTML and JSP tags to display a list of all items in MyStore.
12. Deploy the module OnlineStore.
13. Test the showItems.jsp page.

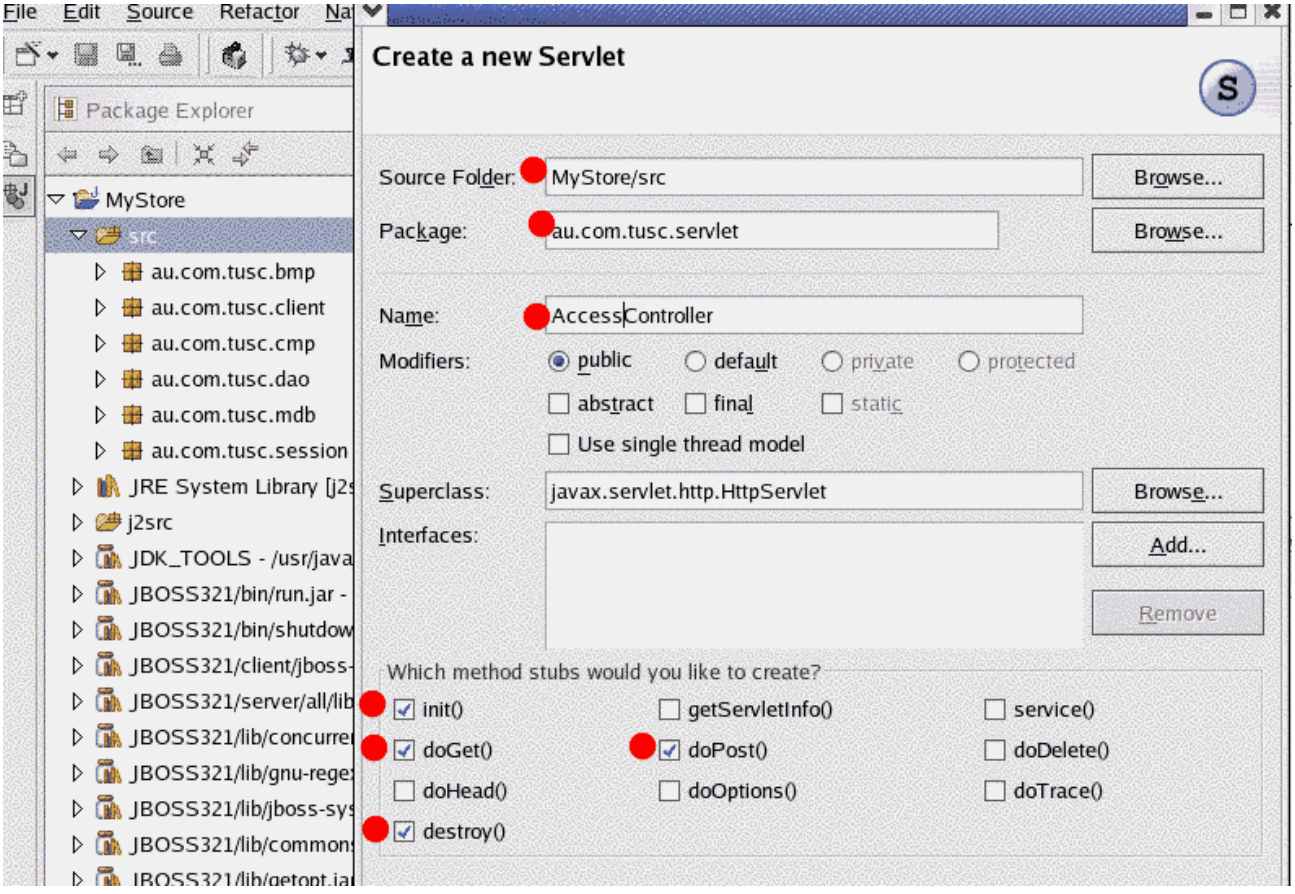
Create AccessController Servlet :

Go To Package Explorer > Expand Mystore (project) node > select src, right click and a menu will pop up.

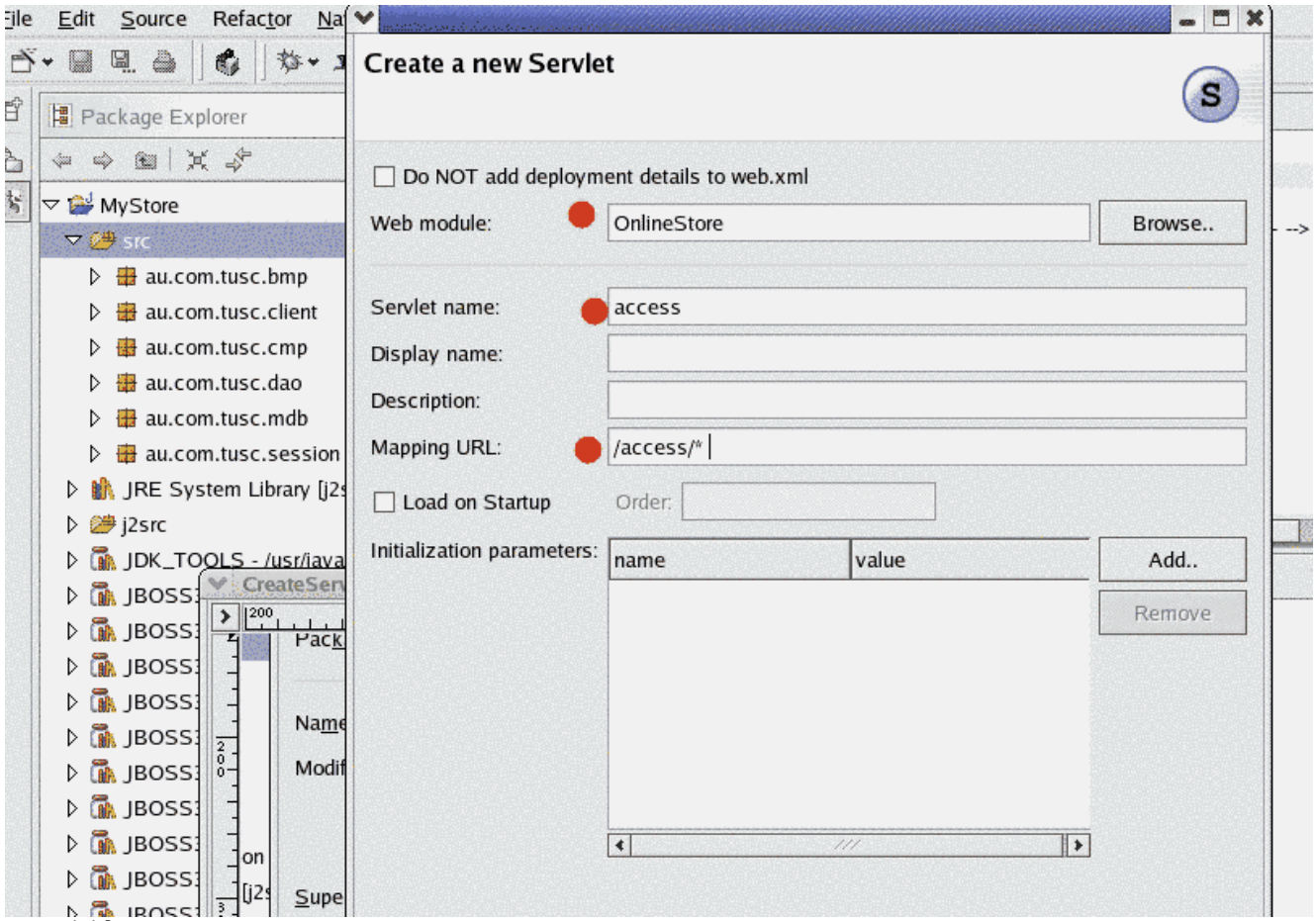
On the pop up menu > New > Lomboz Servlet Wizard as shown below.



Enter the package name au.com.tusc.servlet, with servlet name AccessController and select four methods to be implemented in the servlet as shown.

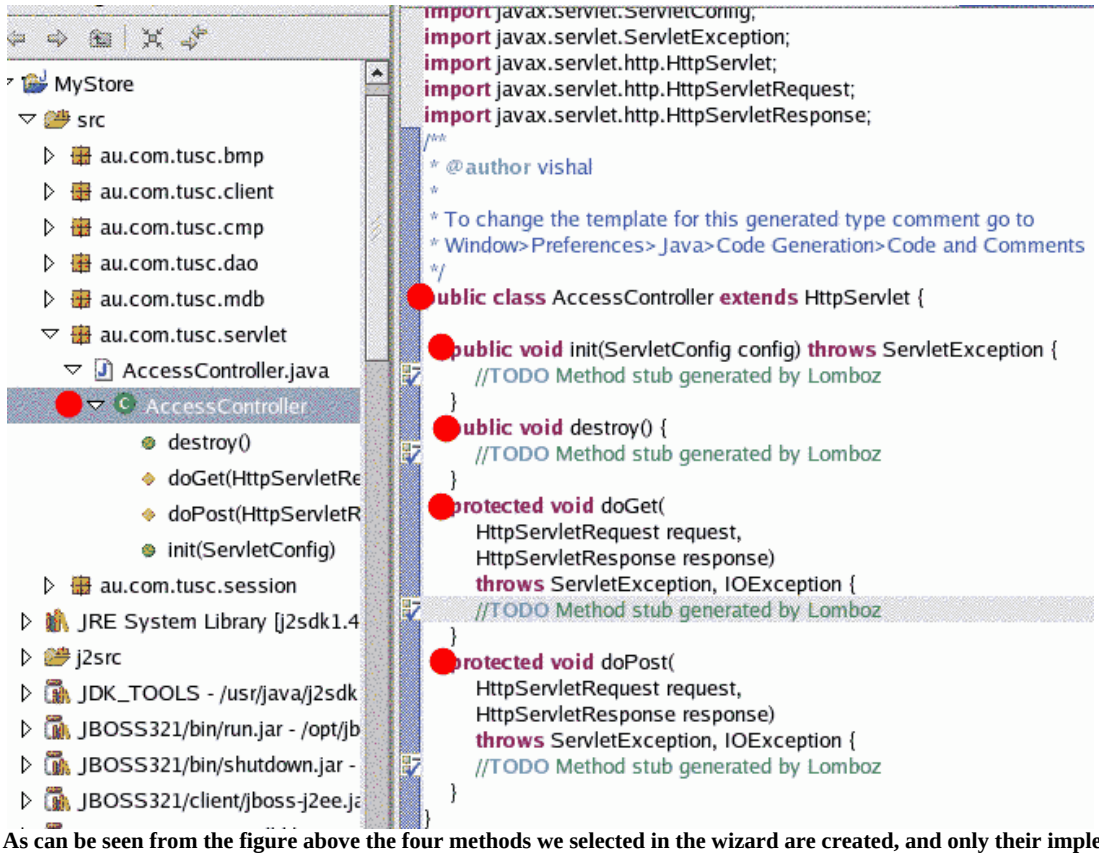


Press Next. A new screen will appear as shown below. Add Web Module (Browse.. web module and it will show list of web modules in this project), in this case it is OnlineStore, so select that. Enter the Servlet name as access, and Mapping URL as ' /access/* '.



Press Finish.

This will create a package named au.com.tusc.servlet under src and AccessController within that package as shown below.



As can be seen from the figure above the four methods we selected in the wizard are created, and only their implementation is required.

Apart from this, some descriptors are generated in web.xml, under Web-Module OnlineStore/WEB-INF as shown below:

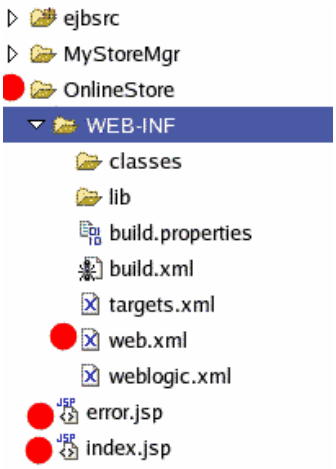
```
<web-app>
   <!-- Remove the comments below to define a servlet.         -->
   <!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++
   <!--    <servlet>                                          -->
   <!--      <servlet-name>MyServlet</servlet-name>           -->
   <!--      <servlet-class>examples.MyServlet</servlet-class> -->
   <!--      <init-param>                                     -->
   <!--        <param-name>myparam</param-name>               -->
   <!--        <param-value>12345</param-value>               -->
   <!--      </init-param>                                    -->
   <!--    </servlet>                                         -->
   <!--                                                       -->
   <!--                                                       -->
   <!--    <servlet-mapping>                                  -->
   <!--        <servlet-name>MyServlet</servlet-name>         -->
   <!--        <url-pattern>/mine/*</url-pattern>             -->
   <!--    </servlet-mapping>                                 -->
   <!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++
   <servlet>
      <servlet-name>access</servlet-name>
      <servlet-class>au.com.tusc.servlet.AccessController</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>access</servlet-name>
      <url-pattern>/access/* </url-pattern>
   </servlet-mapping>
   <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>
   <error-page>
      <error-code>404</error-code>
      <location>/error.jsp</location>
   </error-page>
</web-app>
```

These tags are generated by the Servlet Creation Wizard, where <url-pattern> tag specifies the path name by which the servlet will to be accessed. In this case it will be http://localhost:8080/OnlineStore/access. (N.B. - It's not necessary to have the same <servlet-name> and <url-pattern>.)

Web.xml contains all the deployment descriptors required for deployment of servlets.

Lomboz creates two pages when you create your web module, index.jsp and error.jsp.

Before we go further and start getting our hands dirty with servlets, let's have a look at what directories and files are generated by Lomboz under Web Modules, in this case OnlineStore, as shown below.



Files of interest include web.xml, where all the deployment descriptors will be placed (as discussed above), and targets.xml, which contains information about the server in which it will be deployed (in this case JBOSS). See the code snippet below from targets.xml.

```
<targets>
   <targetServer>JBOSS 3.2.1 ALL</targetServer>
</targets>
```

Add Business Method :

Before we start implementing our servlet, add one more business method to StoreAccess Bean called getAllItems() with the following signature:

public java.util.ArrayList getAllItems()

This method will return all the items in MyStore by invoking the finder method in ItemsLocalHome named findAll, as shown below (code snippet from StoreAccess bean):

```
/**
 * Returns ArrayList of all items in store
 * @ejb.interface-method
 *  tview-type="remote"
 **/
public java.util.ArrayList getAllItems(){

    System.out.println (" Entering StoreAccessBean.getItemsAllItems ");
    Collection items = null;
    ArrayList itemsList = new ArrayList();

    try {
        items =  itemLocalHome.findAll();
        Iterator iterate = items.iterator();
            while (iterate.hasNext()) {
                ItemLocal itemsLocal = (ItemLocal) iterate.next();
                itemsList.add(itemsLocal.getItemData());
            }
        } catch (Exception e) {
          System.out.println (" Error in StoreAccessBean.getAllItems " + e);
        }
        System.out.println (" Leaving StoreAccessBean.getAllItems ");
        return itemsList;
    }
```

Now let's start implementing the generated methods in the servlet.

Implement init method :

This method is responsible for initializing servlets, so it is invoked when the servlet is first created and is not called again for each user request.

We will cache the references for our StoreAccess Bean in this method, as all the client interfaces available are exposed in StoreAccess.

So, first create a context. Then get a reference to a StoreAccess object by looking up the StoreAccess bean via the JNDI API.

Add a variable storeAccessHome of type StoreAccessHome to store the reference obtained by narrowing the object.

Create helper methods for getting Context, Home and assigning the reference to storeAccessHome as shown below.

```
private StoreAccessHome storeAccessHome =null;

public void init(ServletConfig config) throws ServletException {
    super.init(config);
    initStoreAccess();
}

private void initStoreAccess() {

    System.out.println("Entering AccessController.initStoreAccess()");
    try {
        storeAccessHome = getHome();
    }catch (Exception e) {
        System.out.println ("\n Exception in  AccessController.storeAccess() " + e);
    }
    System.out.println("Leaving AccessController.initStoreAccess()");
}

private StoreAccessHome getHome() throws NamingException {

    Object result = getContext().lookup(StoreAccessHome.JNDI_NAME);
    return ( (StoreAccessHome) PortableRemoteObject.narrow(result,StoreAccessHome.class));
}

private InitialContext getContext() throws NamingException {

    Hashtable props = new Hashtable();
    props.put(
            InitialContext.INITIAL_CONTEXT_FACTORY,
            "org.jnp.interfaces.NamingContextFactory");
    props.put(InitialContext.PROVIDER_URL, "jnp://127.0.0.1:1099");
    InitialContext initialContext = new InitialContext(props);
    return initialContext;
}
```

```
    return initialContext;
  }
```

Note : We have to narrow this object because we are accessing a remote interface. If it was a local interface, we wouldn't need to narrow the object.

Implement methods doGet and doPost :

In order to implement these two methods we will create a helper method to provide the functionality for both of them. The request is delegated to this method where all processing of information takes place. Once this business logic processing is complete it dispatches the request to the appropriate view for display, ie. JSP pages.

Note : This approach is based on the Front Controller pattern, where the controller acts as a central point of contact for handling requests, delegating business processing, and coordinating with dispatcher components, whilst dispatchers are responsible for view management and navigation. This pattern suggests centralizing the handling of all requests in this way, but also allows for different handler methods to be used in processing different request types.

Add a method named procesRequest with the following signature:

protected void processRequest (HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException

Delegate request from doGet and doPost method to this method as shown below.

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

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

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

}
```

Now implement the processRequest() method.

This method is structured such that it will check for the parameter useraction within the request object. If the useraction parameter is empty then it will build a url for the login screen and generate a login screen page. If useraction has some value then it examines it. If it finds the request is to display items then it builds the url for that and generates a page displaying all the items. Finally, for errors a page is generated to display the error.

Add following attributes shown below, for building different urls.

```
private StoreAccessHome storeAccessHome =null;
private StoreAccess myStore = null;
private static String  LOGIN_SCREEN="/login";
private static String  LOGIN_ERROR_SCREEN="/loginError";
private static String  ITEMS_SCREEN="/showItems";
```

Create a session object and get value from useraction parameter of request object as shown below in code snippet.

```
protected void processRequest (HttpServletRequest request, HttpServletResponse response)throws ServletException, IOException {
    System.out.println("Entering AccessController.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("In validateLogin");
            String username = request.getParameter("username");
            String password = request.getParameter("password");
            if ( (username == null)  || !(loginUser(username, password, session) )) {
                System.out.println("Error : invalid attempt for username");
                buildUrl = LOGIN_ERROR_SCREEN;
            }
            else {    buildUrl = ITEMS_SCREEN;     }
        }
    }
    if ( buildUrl == ITEMS_SCREEN ) {
        try {
            String userID = (String) session.getAttribute("userID");
            myStore = storeAccessHome.create();
            ArrayList itemsList  = myStore.getAllItems();
            displayAllItems(response, itemsList);
            myStore.remove();
        }catch (Exception e) {
            System.out.println("Exception in AccessController.processRequest() " +e );
        }
    }
    else {
        if (buildUrl ==  LOGIN_SCREEN ) { displayLoginScreen(response);   }
        else if (buildUrl == LOGIN_ERROR_SCREEN) { displayLoginErrorScreen(response);     }
    }
    System.out.println("Leaving AccessController.processRequest()");
  }
```

If useraction is empty, as it will be initially, then build the url for the login screen, and generate the login screen by invoking the method displayLoginScreen. Code snippet for displayLoginScreen is shown below.

```
private void displayLoginScreen (HttpServletResponse response)
            throws  IOException {

    System.out.println("Entering AccessController.displayLoginScreen()");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter ();
    out.println("<html><title>MyStore Login</title>");
    out.println("<body><h2>Welcome to MyStore</h2></body>");
    out.println("<form method=\"get\">");
    displayLoginDataFields(out);
    out.println("</form></html>");
    if ( out != null ) out.close();
    System.out.println("Leaving AccessController.displayLoginScreen()");

}
```

This method calls displayLoginDataFields to generate the data fields for form submission as shown below.

```
private void displayLoginDataFields (PrintWriter out ) {

    out.println("<h3>Please enter your username and password : .");
    out.println("<table><tr><td>Username : <td><input name=\"username\" type=\"text\">");
    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\">");
  }
```

Once this form is submitted, it checks to see whether validation is successful by invoking the method loginUser, which invokes the loginUser method in the StoreAccess Bean as shown below.

```
private boolean loginUser (String username, String passwd, HttpSession session) {
    System.out.println ("\n Entering AccessContoller.loginUser()" );
    String userid = null;
    try {
       StoreAccess sa = storeAccessHome.create();
       userid = sa.loginUser(username,passwd);
       session.setAttribute("userID", userid);
    } catch (Exception e) {
       System.out.println ("\n Exception in AccessContoller.loginUser() " + e);
    }
    System.out.println ("\n Leaving AccessContoller.loginUser()");
    return ( userid != null );
}
```

Once that is finished, as shown above in the processRequest method code snippet, it builds url's for displaying items and error. Code snippet for the method displayAllItems is shown below.

```
private void displayAllItems(HttpServletResponse response, ArrayList itemsList)
        throws IOException {
    System.out.println("Entering AccessController.displayAllItems()");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter ();
    out.println("<html><title>List of MyStore Items</title>");
    out.println("<form method=\"GET\">");
    out.println("<body><h2>MyStore Items Cataloge</h2></body>");
    if (itemsList.isEmpty()) { out.println("<p><h3><b>Items are not availabe</b></h3>");    }
    else {
       out.println("<p><table border=\"1\">");
       out.println("<tr><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();
       ItemData id = null;
       while (items.hasNext()) {
          id = (ItemData)items.next() ;
          out.println("<tr><td>");
          out.println(id.getItemID());
          out.println("</td><td>");
          out.println(id.getDescription());
          out.println("</td><td>");
          out.println(id.getQuantity());
          out.println("</td><td>");
          out.println(id.getPrice());
          out.println("</td></tr>");
       }
       out.println("</table>");
    }
    out.println("</html>");
    if ( out != null ) out.close();
    System.out.println("Leaving AccessController.displayAllItems");
}
```

Code snippet for displayLoginErrorScreen shown below.

```
private void displayLoginErrorScreen (HttpServletResponse response)
        throws IOException {

    System.out.println("Entering AccessController.displayLoginErrorScreen()");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter ();
    out.println("<html><title>MyStore 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 AccessController.displayLoginErrorScreen()");
}
```

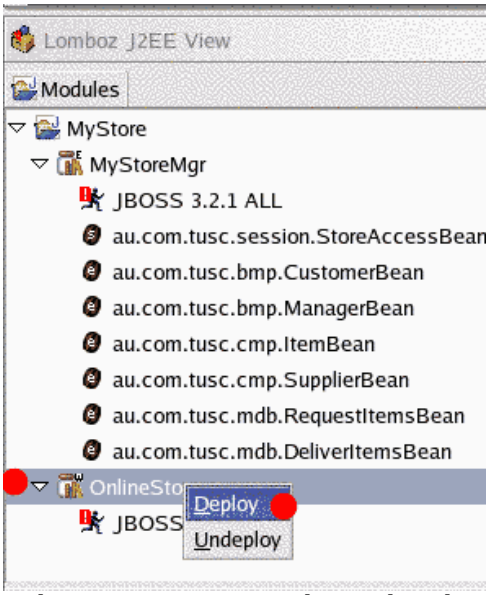
All these helper method are being used by the processRequest method, which is acting as a Controller and dispatches the request to the appropriate view, such as the login screen or display items screen.

Now our servlet is complete, as implementation of the rest of the methods is left as an exercise.

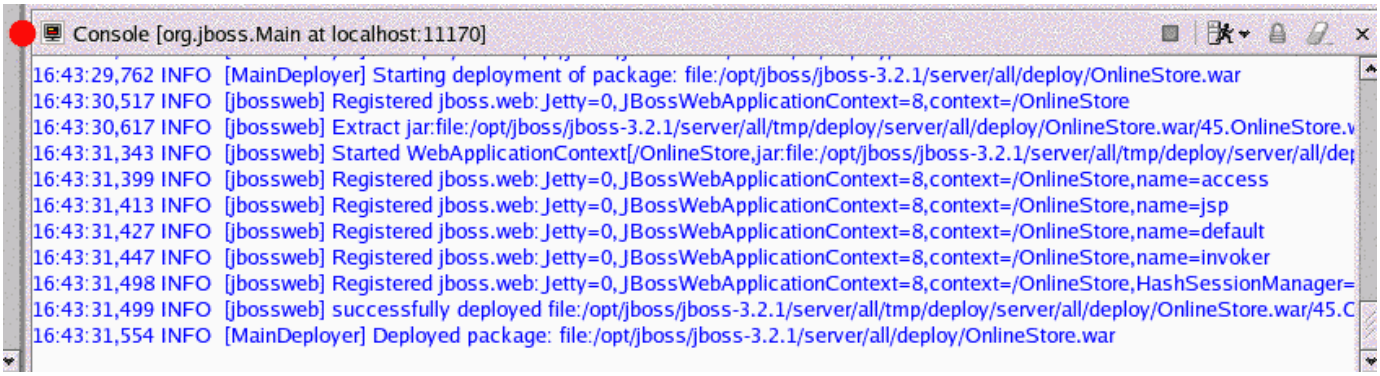
Deploy AccessController Servlet :

In order to deploy your servlet, go to LombozJ2EE view.

Select web module OnlineStore > right click > select Deploy module as shown below , make sure the server is running.



Deployment status appears on the console as shown below. If deployment is successful then test your servlet.

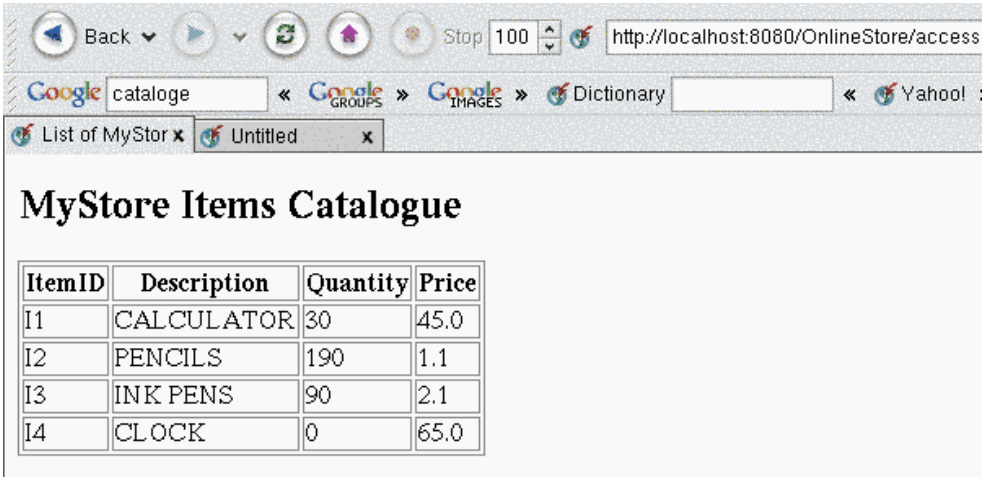


Test your Servlet :

Go to your browser and access the servlet using following URL, 'http://localhost:8080/OnlineStore/access'

where 'access' is the url mapping that was assigned while creating the servlet using the Servlet Creation Wizard, and OnlineStore is the web module, where this servlet resides.

The login screen will be displayed. Enter username as 'ANDY' and password as 'PASSWD'. The next screen will be list of items in MyStore as shown below.

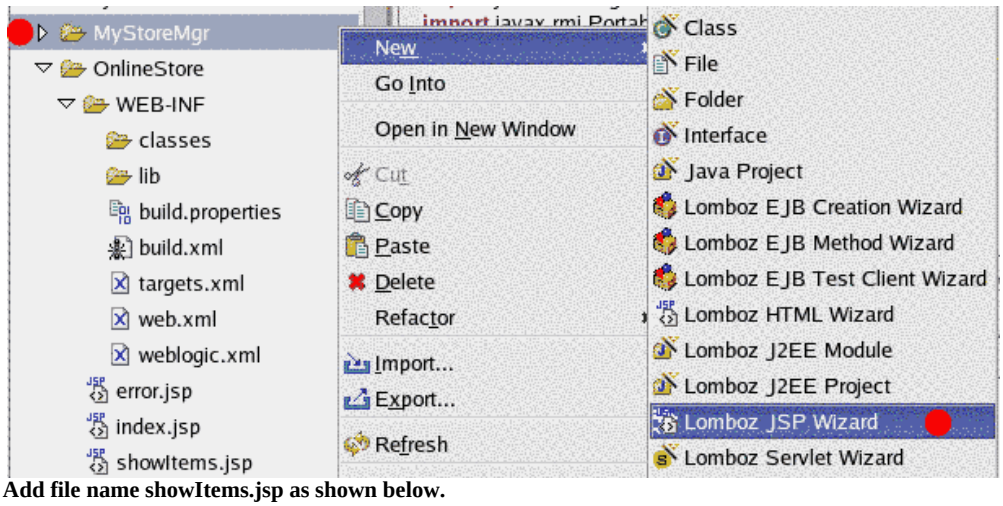


We have successfully created a servlet, now let's access this catalogue via a JSP page.

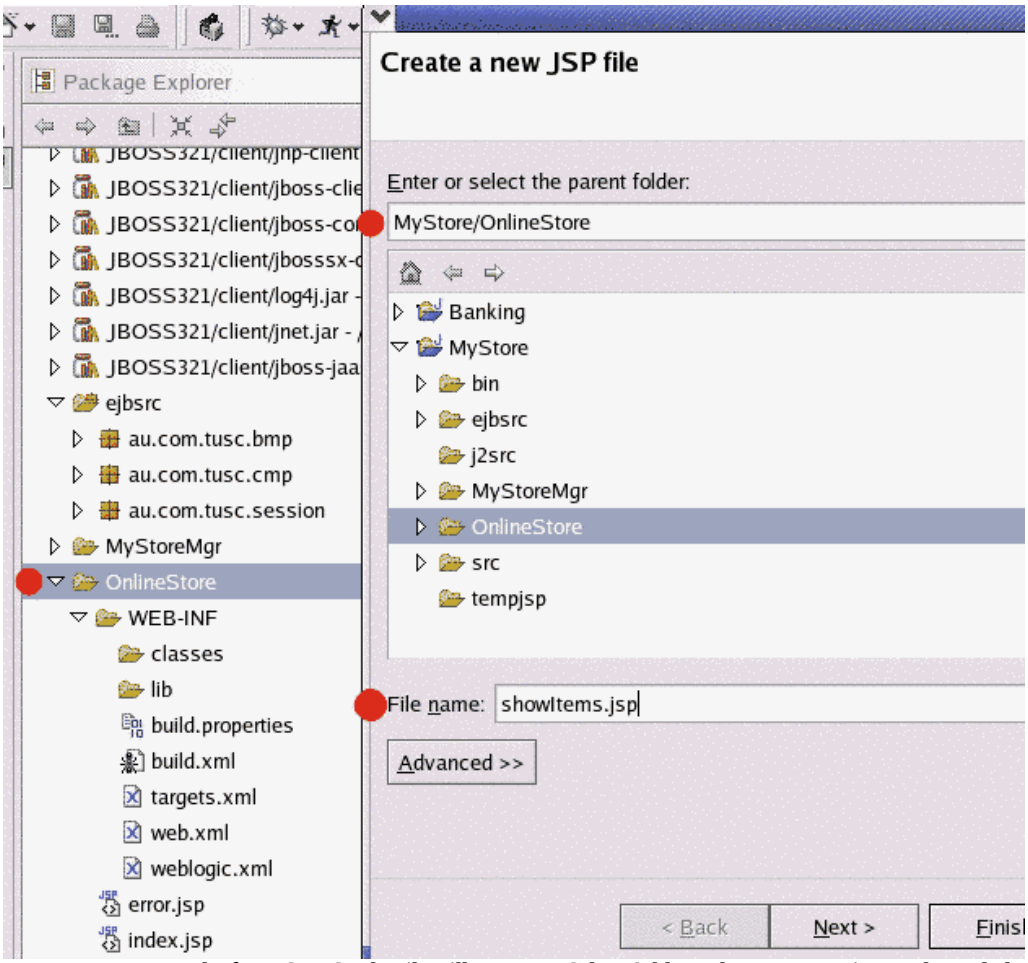
Create JSP Page :

Go To Package Explorer > Expand Web Module node (OnlineStore) > right click and a menu will pop up.

On the pop up menu > New > Select Lomboz JSP Wizard as shown below.

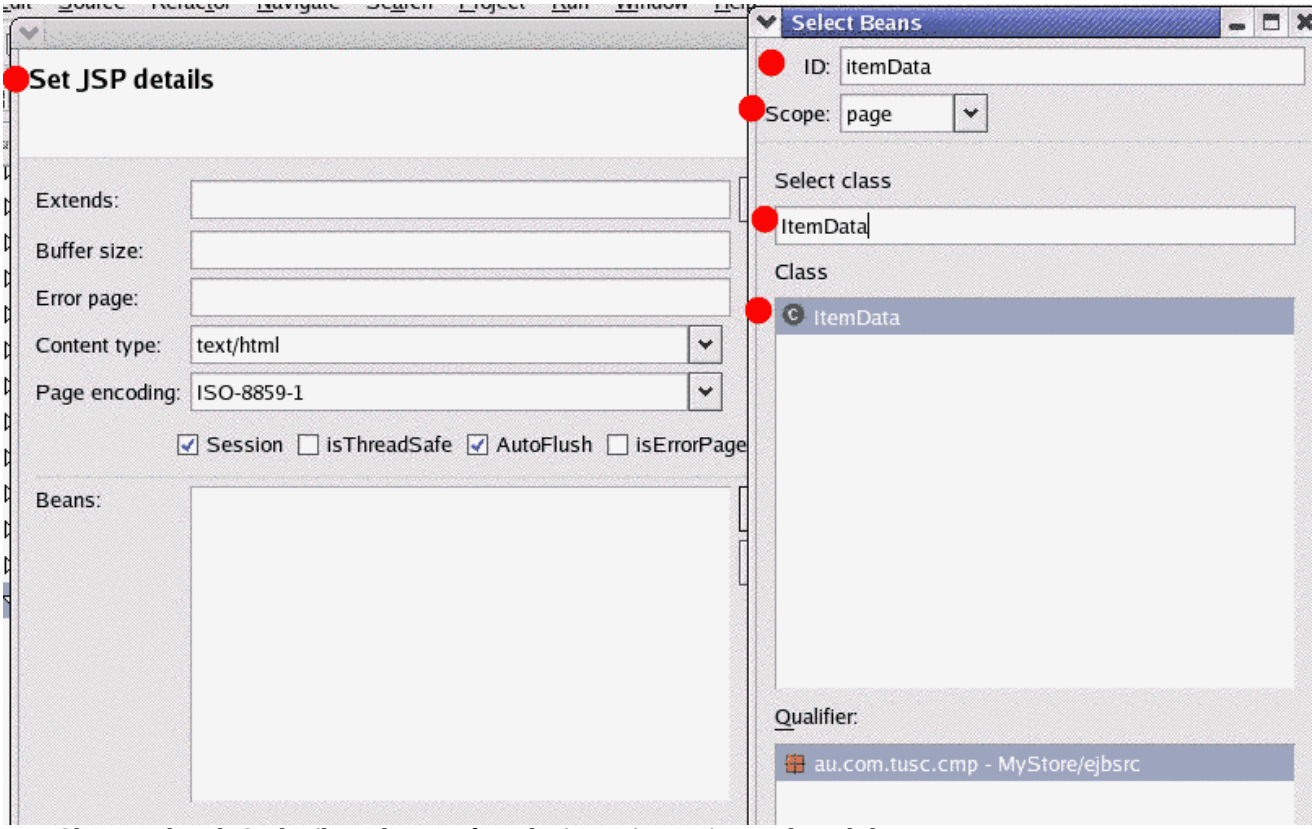


Add file name showItems.jsp as shown below.

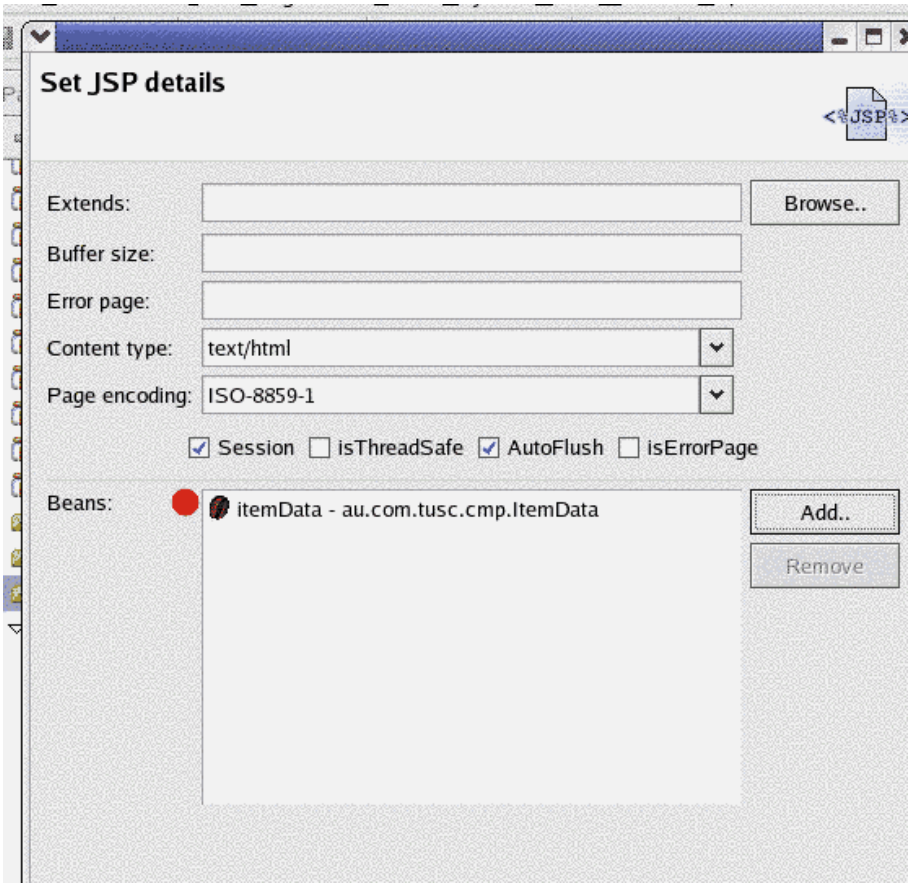


Press Next > Menu, the form Set JSP details will appear > Select Add.. under Beans section as shown below.

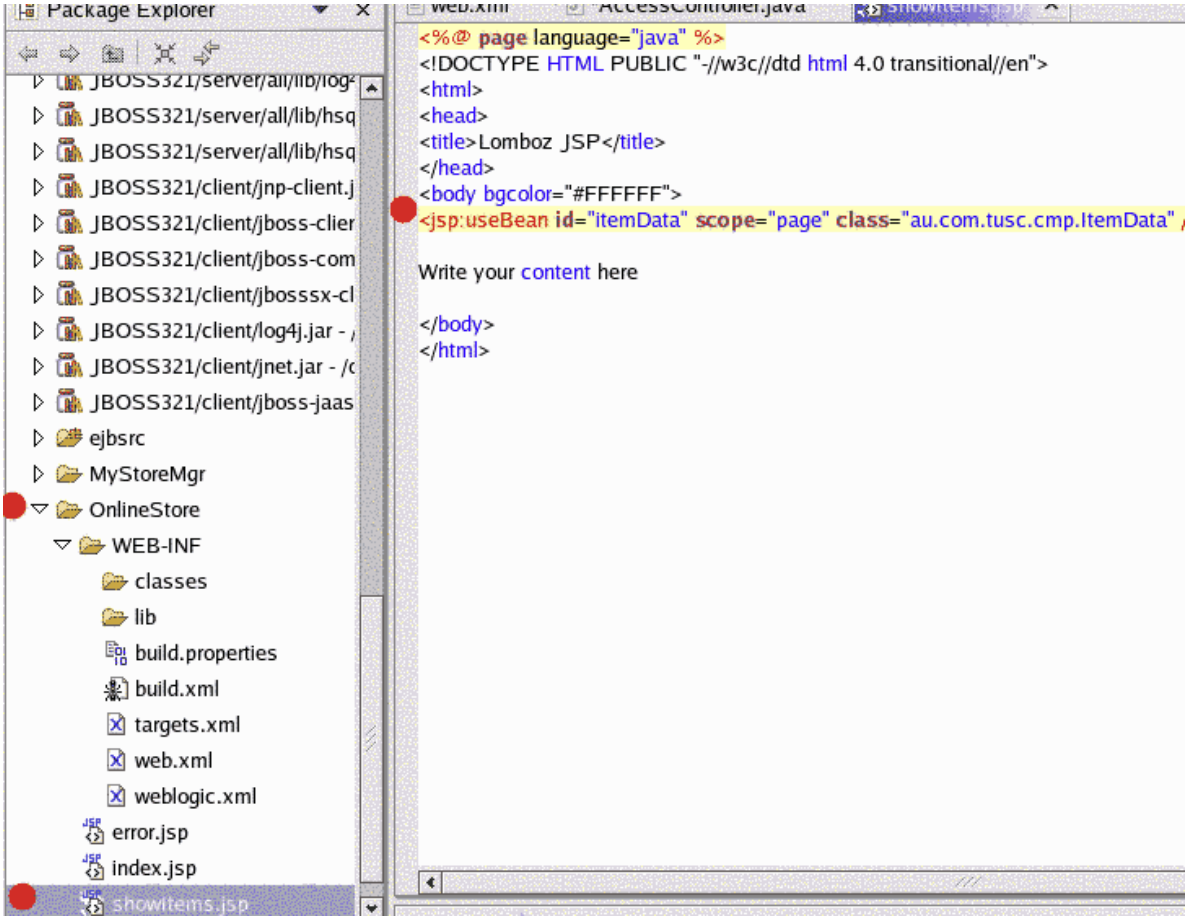
The menu for Select Beans will appear > Add ID as itemData, Scope as page and Class as ItemData.



Press Ok. Now selected JSP details can be seen after selecting various options as shown below.



Press Finish. A new file named showItems.jsp will be created under the web module OnlineStore as shown below.



Modify Method processRequest in Servlet AccessController :

Now, in order to display all items of MyStore from a JSP page, we have to modify the proccessRequest method in servlet AccessController.

Modify String ITEMS_SCREEN to "/showItems.jsp" as shown below.

```
private static String  ITEMS_SCREEN="/showItems.jsp";
```

Comment out the call to method displayAllItems in processRequest.

Add a Session attribute named itemsList with values of listItems.

Add a Request Dispatcher and pass buildUrl as an argument, which has the url for the showItems.jsp page.

Forward to that request dispatcher to draw the JSP page.

Code snippet of the modified processRequest method is shown below.

```
protected void processRequest (HttpServletRequest request, HttpServletResponse re
    System.out.println("Entering AccessController.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("In validateLogin");
            String username = request.getParameter("username");
            String password = request.getParameter("password");
             if ( (username == null)  || !(loginUser(username, password, session) )) {
               System.out.println("Error : invalid attempt for username");
               buildUrl = LOGIN_ERROR_SCREEN;
            }
            else {    buildUrl = ITEMS_SCREEN;     }
        }
    }
  if ( buildUrl == ITEMS_SCREEN ) {
        try {
            String userID = (String) session.getAttribute("userID");
            myStore = storeAccessHome.create();
            ArrayList itemsList  = myStore.getAllItems();
            //displayAllItems(response, itemsList);
          session.setAttribute("itemsList", itemsList);
            System.out.println("BuildUrl is "+ buildUrl);
          RequestDispatcher rd = getServletContext().getRequestDispatcher(buildUrl);
          rd.forward(request,response);
            myStore.remove();
        }catch (Exception e) {
            System.out.println("Exception in AccessController.processRequest() " +e );
        }
    }
    else {
        if (buildUrl ==  LOGIN_SCREEN ) {
```

Add Html and JSP Tags :

First import the following packages: au.com.tusc.cmp.*, java.util.ArrayList and java.util.Iterator.

Now add the necessary HTML and JSP tags to access items of MyStore.

Get itemsList from our session attribute set in servlet AccessController.

Set itemData as a page attribute via pageContext.

Now display the attributes of itemData using JSP property access tags. Code snippet for the showItems page is shown below.

```
<%@ page language="java" import="au.com.tusc.cmp.*, java.util.ArrayList, java.util.Iterator"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>Lomboz JSP</title>
</head> <body bgcolor="#FFFFFF">
<form method="post">
<b><h3> MyStore Items Catalogue </h3></b><br>
<% ArrayList itemsList = (ArrayList) session.getAttribute("itemsList");
  if ( itemsList.isEmpty() ) { %>
    <p><h3><b>Items are not availabe</b></h3>
 <%}
    else { %>
        <p><table border="1">
        <tr><th><b>ItemID</b><th><b>Description</b>
        <th><b>Quantity</b><th><b>Price</b></tr>
        <%
        Iterator items = itemsList.iterator();
        ItemData id = null;
      while (items.hasNext()) {
           id = (ItemData)items.next() ;
        pageContext.setAttribute("itemData", id);
        %>
        <jsp:useBean id="itemData" scope="page" class="au.com.tusc.cmp.ItemData" />
        <tr><td>
        <jsp:getProperty name= "itemData" property="itemID" />
        </td><td>
        <jsp:getProperty name= "itemData" property="description" />
        </td><td>
        <jsp:getProperty name= "itemData" property="quantity" />
        </td><td>
```

```
            </td><td>
            <jsp:getProperty name= "itemData" property="quantity" />
            </td><td>
            <jsp:getProperty name= "itemData" property="price" />    </td></tr>
        <%} %>
        </table>
    <%} %>
</form>
```

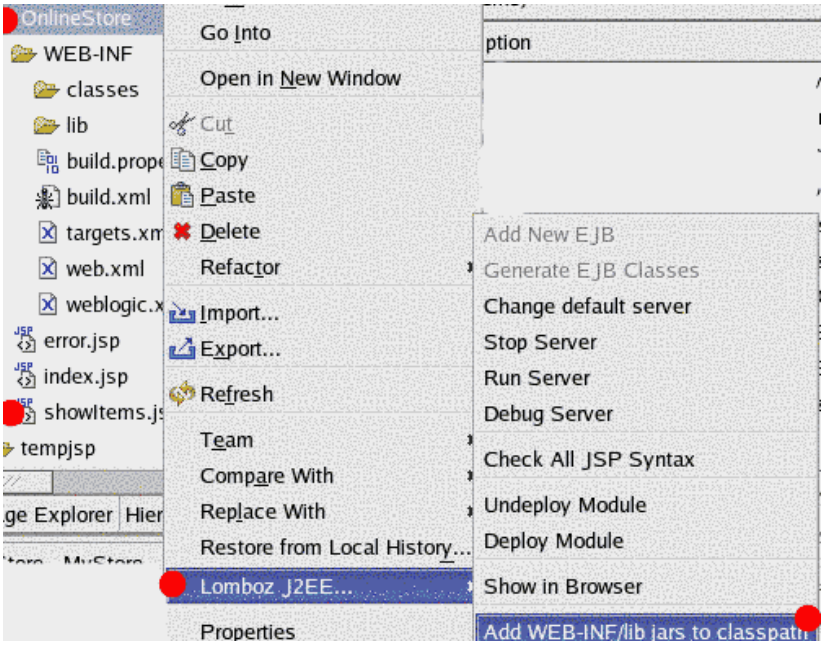
Note : This JSP Page is a view in the Front Controller Pattern discussed above.

Now the JSP page is done.

Go to OnlineStore (web module) node > right click > select Lomboz J2EE.. on pop up menu > Check All JSP Syntax.

See if there are any errors.

Go to OnlineStore node > right click > select Lomboz J2EE.. on pop up menu > Add WEB-INF/lib JARs to the Classpath as shown below.



Deploy Module OnlineStore :

In order to deploy, go to LombozJ2EE view.

Select web module OnlineStore > right click > select Deploy module, make sure the server is running.

Deployment status will be visible on the console. If deployment is successful then test your JSP page.

Test your JSP Page :

Go to your browser and access the servlet using the following web address, 'http://localhost:8080/OnlineStore/access'

where 'access' is the url mapping that was assigned whilst creating the servlet in the Servlet Creation Wizard, and OnlineStore is the web module where the servlet AccessController resides.

The login screen will be displayed. Enter username as 'ANDY' and password as 'PASSWD'. The next screen will be the list of items in MyStore as shown below.

MyStore Items Catalogue

ItemID	Description	Quantity	Price
I1	CALCULATOR	30	45.0
I2	PENCILS	190	1.1
I3	INK PENS	90	2.1
I4	CLOCK	0	65.0

You have successfully created a JSP page using all the J2EE components, and they have been created and deployed successfully. Congratulations!

Prev TOC Next

Chapter 9 .

Creating Web Services :

Web services promises to be the next generation of software development. In essence, a web service is a means of interfacing to web or enterprise applications that allows you to integrate these with other enterprise applications, including those from different vendors and different platforms, using XML as the means of exchanging data.

There is a great deal of hype about web services. In the past there have been a number of initiatives to provide such vendor-and-platform agnostic integration technology, but none of these has been totally succesful. A good example is that of getting an enterprise applicaction running on Windows to communicate with an enterprise application running on Unix. Sure, it's been possible by various means but typically with a mechanism which is not equally well supported by both environments. That has changed with the advent of web services SOAP and XML provide an integration glue that everyone can finally agree on.

The basic mechanism of Web services is to use XML to transport its across different applications using the standard Web HTTP protocol (Hyper Text Transfer Protocol). Specifically SOAP (Simple Object Access Protocol) is used, which is a lightweight XML based RPC (Remote Procedure Call) over HTTP. The protocol has three parts :

1. An envelope (SOAP envelope) that defines a framework for describing what is in a message and how to process it.
2. A set of encoding rules for expressing instances of application defined-datatypes.
3. A set of rules for representing remote procedure calls and responses.

Web Services Standards :

Now lets go through some evolving standards used in in Web Services, which are SOAP, WSDL (Web Services Description Language), UDDI (Universal Description Discovery and Integration), ebXML which is promoted by OASIS (Organization for the Advancement of Structured Information Standards) and UN/CEFACT (United Nations Centre for Trade Facilitation and Electronic Business). We will cover only SOAP and WSDL in this tutorial.

WSDL (Web Services Description Language) describes the interface of a network service similar to IDL (Interface Definition Language) in CORBA (Common Object Request Broker Architecture), specifying what messages an endpoint (i.e. a service instance which processes client requests and returns responses) will receive and send. WSDL is itself an XML metadata format. (In the terms of similar mechanisms like RMI or CORBA, a service is the interface or set of methods that a client will invoke across the wire).

SOAP (Simple Object Access Protocol) as discussed earlier describes the format of the data that is transmitted over the wire.

Both these standards are still evolving under the eye of the W3C, but these standards are regarded as the building block of the next revolution in distributed computing and the IT industry in general. Web services promise to do for machine-to-machine communication what the Web has done for human-to-human and machine-to-human communication.

However, due to the evolving nature of the web services standards together with the various related tools and platforms, the full impact is still far from being realized.

Web Services In Java :

As discussed earlier, Web Services standards are still evolving and their support in Java is still evolving. Even though Web Services were initially promoted by Microsoft and IBM, with the release of J2EE 1.4 Sun has provided full support for Web Services.

In the J2EE environment Web services are built on JAX-RPC (Java API for XML-based RPC). This is an API for building Web services and clients that use remote procedure calls (RPC) and XML.

In JAX-RPC, a remote procedure call is represented by an XML-based protocol such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and convention for representing remote procedure calls and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP. Even though the SOAP messages are a bit complex, but this complexity is transparent to the developer. When developing client and server side implementations, developers don't have to generate SOAP messages these are generated by the JAX-RPC API calls.

On the server side, the developer specifies the remote procedures by defining methods in an interface written in the Java programming language. The developer also codes 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 simply invokes methods on the proxy. With JAX-RPC, the developer does not generate or parse SOAP messages. It is the JAX-RPC runtime system that converts the API calls and responses to and from SOAP messages.

For this tutorial, we will use the Apache Axis library, which is an open source initiative of the Apache Software Foundation (ASF). Axis is a JAX-RPC compliant SOAP engine. It can be integrated with web containers like Tomcat/Jetty, which allows it to use the features of such web containers like security, resource pooling, multi-threading,

etc.

Note: JAX-RPC is based on JSR-101, which took the first step in defining a standard set of Java APIs and a programming model for developing and deploying Web services on the Java platform. Apache Axis is JAX-RPC compliant.

Now another specification which is evolving is JSR-109. This builds upon JAX-RPC. It defines a standard mechanism for deploying a Web service in the J2EE environment, more specifically, in the area of Enterprise JavaBean (EJB) technology and servlet containers.

Both of these specifications will be integrated into the J2EE 1.4 specification. Together they serve as the basis of Web services for J2EE.

Jboss.3.2.x comes with integrated support for Web-services provided by Jboss.Net. Jboss.NET has Apache AXIS as a plugin service for the Jboss-Microkernel. Jboss.Net uses the built-in JBoss deployment system that understands the WSR packaging. [Web Service aRchive (WSR- or .wsr-) files are ordinary .jar files that contain, besides necessary byte code for plugging into the Axis machinery, a slightly extended Axis deployment descriptor in their "META-INF/web-service.xml" entry]. Currently, there are not that many application servers available which support WSR deployment, which can result in portability issues.

Therefore for this tutorial we will use Apache-AXIS 1.1 for creating and deploying web services.

Installing AXIS :

Download the Apache-Axis binary from http://ws.apache.org/axis/.

Ex : Version 1.1 Binary - tar.gz

First create a dir named axis under /opt and then save this file under /opt/axis/.

```
[vishal@vishal axis]$ pwd
/opt/axis
[vishal@vishal axis]$ ls
axis-1_1.tar.gz
```

Now unzip the file. A new directory will be created with the name axis-1_1.

```
[vishal@vishal axis]$ tar -xvzf axis-1_1.tar.gz
axis-1_1/
axis-1_1/docs/
axis-1_1/docs/ant/
axis-1_1/docs/ant/ant.html
axis-1_1/docs/ant/axis-admin.html
------------------------------------------------------------
------------------------------------------------------------
------------------------------------------------------------
axis-1_1/xmls/
axis-1_1/xmls/checkstyle.xml
axis-1_1/xmls/deploy_catalina_local.xml
axis-1_1/xmls/path_refs.xml
axis-1_1/xmls/properties.xml
axis-1_1/xmls/targets.xml
axis-1_1/xmls/taskdefs.xml
axis-1_1/xmls/taskdefs_post_compile.xml.
[vishal@vishal axis]$ ls
axis-1_1 axis-1_1.tar.gz
[vishal@vishal axis]$
```

The directory structure will be as shown below.

```
[vishal@vishal axis]$ pwd
/opt/axis
[vishal@vishal axis]$ ls
axis-1_1 axis-1_1.tar.gz
[vishal@vishal axis]$ cd axis-1_1

[vishal@vishal axis-1_1]$ ls
docs lib LICENSE README release-notes.html samples webapps xmls

[vishal@vishal axis-1_1]$ pwd
/opt/axis/axis-1_1

[vishal@vishal axis-1_1]$ cd lib/

[vishal@vishal lib]$ ls
axis-ant.jar commons-discovery.jar jaxrpc.jar saaj.jar
axis.jar commons-logging.jar log4j-1.2.8.jar wsdl4j.jar
[vishal@vishal lib]$
```

Now the axis binary is installed, let's configure it for use with our application server Jboss-3.2.1.

Configuring AXIS with JBOSS :

In order to configure Jboss with Axis, first go to $JBOSS_HOME/server/all/deploy/all/ as shown below.

```
[vishal@vishal deploy]$ pwd
/opt/jboss/jboss-3.2.1/server/all/deploy
[vishal@vishal deploy]$
```

Create a dir named webapps under this directory as shown below.

```
[vishal@vishal deploy]$ ls
cache-invalidation-service.xml jbossweb-jetty.sar MyStoreMgr.jar
cluster-service.xml jboss-xa-jdbc.rar OnlineStore.war
farm-service.xml jms properties-service.xml
hsqldb-ds.xml jmx-console.war schedule-manager-service.xml
http-invoker.sar jmx-ejb-connector-server.sar scheduler-service.xml
iiop-service.xml jmx-invoker-adaptor-server.sar sqlexception-service.xml
jbossha-httpsession.sar jmx-rmi-adaptor.sar transaction-service.xml
jboss-jca.sar mail-service.xml user-service.xml
jboss-local-jdbc.rar management uuid-key-generator.sar
jboss-net.sar MyFirstBean.jar
jbossweb-ejb.jar MySecondBean.jar

[vishal@vishal deploy]$ mkdir webapps

[vishal@vishal deploy]$ ls
cache-invalidation-service.xml jbossweb-jetty.sar MyStoreMgr.jar
cluster-service.xml jboss-xa-jdbc.rar OnlineStore.war
farm-service.xml jms properties-service.xml
hsqldb-ds.xml jmx-console.war schedule-manager-service.xml
http-invoker.sar jmx-ejb-connector-server.sar scheduler-service.xml
iiop-service.xml jmx-invoker-adaptor-server.sar sqlexception-service.xml
jbossha-httpsession.sar jmx-rmi-adaptor.sar transaction-service.xml
jboss-jca.sar mail-service.xml user-service.xml
jboss-local-jdbc.rar management uuid-key-generator.sar
jboss-net.sar MyFirstBean.jar webapps
jbossweb-ejb.jar MySecondBean.jar
[vishal@vishal deploy]$
```

Now go to /opt/axis/axis-1_1/webapps dir. Copy the axis dir to $JBOSS_HOME/server/all/deploy/all/webapps/ as shown below.

Note : Here JBOSS_HOME is /opt/jboss/jboss-3.2.1

```
[vishal@vishal webapps]$ pwd
/opt/axis/axis-1_1/webapps
[vishal@vishal webapps]$ ls
axis
[vishal@vishal webapps]$ cp -r axis /opt/jboss/jboss-3.2.1/server/all/deploy/webapps/.
[vishal@vishal webapps]$
```

Now go back to the $JBOSS_HOME/server/all/deploy/webapps/ dir and rename the axis directory to axis.war.

```
[vishal@vishal webapps]$ pwd
/opt/jboss/jboss-3.2.1/server/all/deploy/webapps
[vishal@vishal webapps]$ ls
axis
[vishal@vishal webapps]$ mv axis axis.war
[vishal@vishal webapps]$ ls
axis.war
[vishal@vishal webapps]$
```

Note : As shown above, the directory axis has been renamed to axis.war so that JBOSS can recognize this as a web application and hence the web container can deploy it at run-time.

Now from within Eclipse start your Application server (JBOSS) , if it isn't already running. If it is already running then don't restart it as it will hot-deploy the the axis.war.

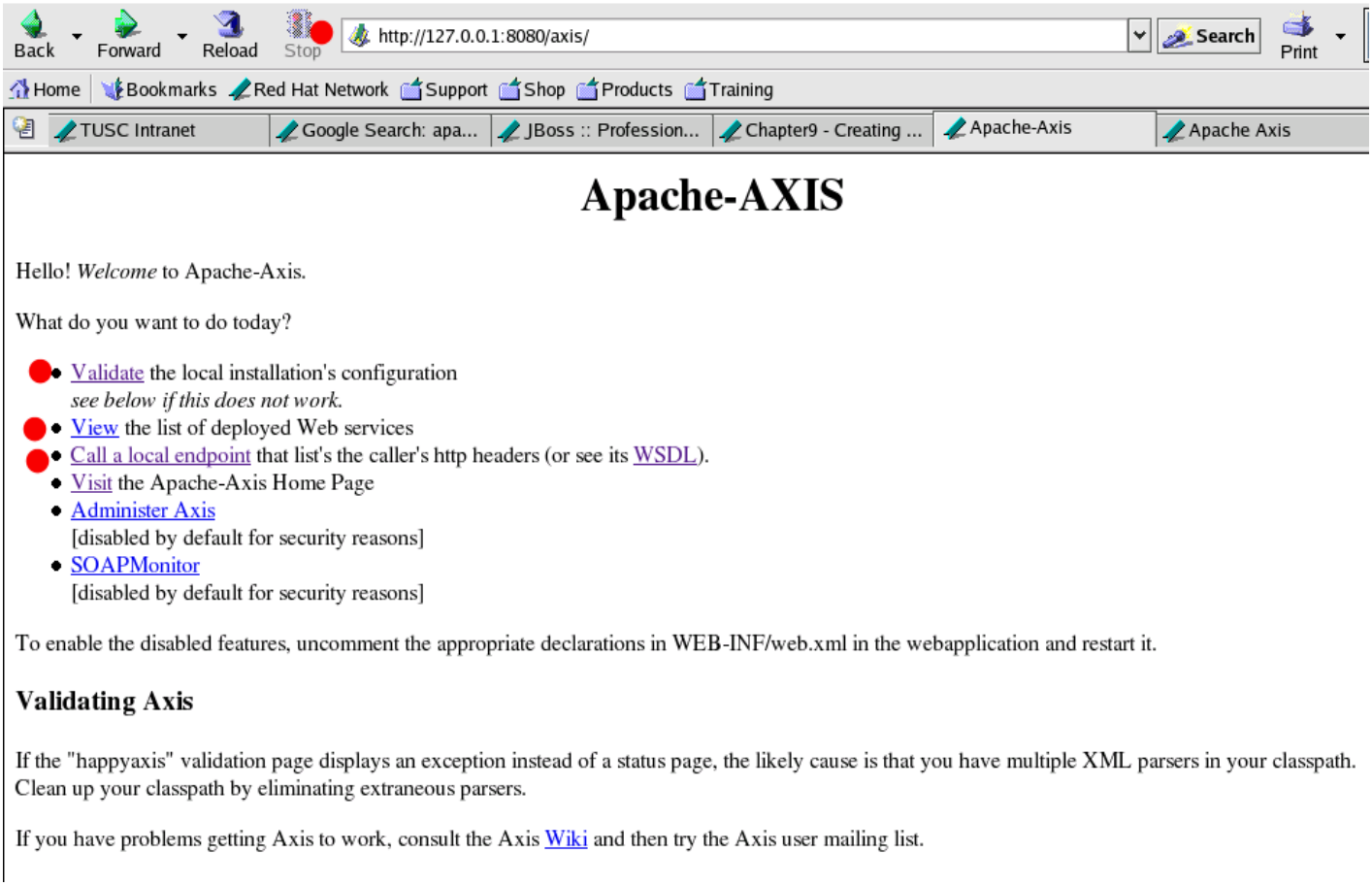
Also make sure all the libraries which come with AXIS are on the classpath ($CLASSPATH or $AXISCLASSPATH) as shown below.

```
[vishal@vishal lib]$ pwd
/opt/jboss/jboss-3.2.1/server/all/deploy/webapps/axis.war/WEB-INF/lib
[vishal@vishal lib]$ ls
axis-ant.jar commons-discovery.jar jaxrpc.jar saaj.jar
axis.jar commons-logging.jar log4j-1.2.8.jar wsdl4j.jar
[vishal@vishal lib]$ echo $AXISCLASSPATH
/opt/axis/axis-1_1/lib/axis.jar:/opt/axis/axis-1_1/lib/jaxrpc.jar:/opt/axis/axis-1_1/lib/saaj.jar:/opt/axis/axis-1_1/lib/wsdl4j.jar:/opt/axis/axis-1_1/lib/commons-discovery.jar:/opt/axis/
axis-1_1/lib/commons-logging.jar:.
[vishal@vishal lib]$
```

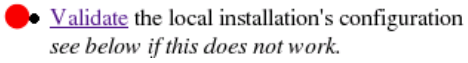
Now to test your configuration go to http://127.0.0.1:8080/axis/.

Note : The port no. will be different if you have customized it from its default settings.

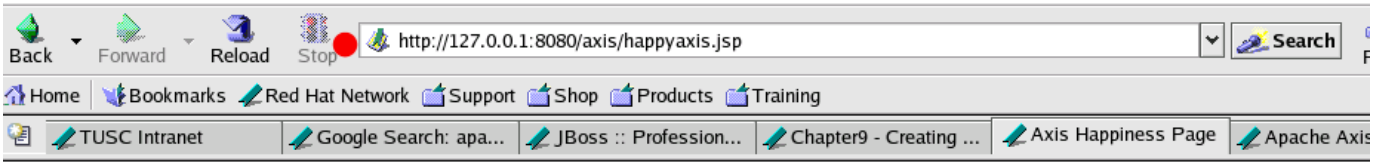
You should now be able to see an Apache-Axis start page as shown below. If you do not, then either the webapp is not actually installed properly, or the appserver is not running.



Now Validate the local installation's configuration by clicking on the link 'Validate the local installation's configuration'.



This will bring you to *happyaxis.jsp*, a test page that verifies whether required and optional libraries are present. The URL for this will be something like http://localhost:8080/axis/happyaxis.jsp as shown below in figure.



Axis Happiness Page

Examining webapp configuration

Needed Components

Found SAAJ API (javax.xml.soap.SOAPMessage) at /opt/jboss/jboss-3.2.1/server/all/tmp/deploy/server/all/deploy/jboss-net.sar/saaj.jar/18.saaj.jar
Found JAX-RPC API (javax.xml.rpc.Service) at /opt/jboss/jboss-3.2.1/server/all/tmp/deploy/server/all/deploy/jboss-net.sar/jaxrpc.jar/15.jaxrpc.jar
Found Apache-Axis (org.apache.axis.transport.http.AxisServlet) at
/opt/jboss/jboss-3.2.1/server/all/tmp/deploy/server/all/deploy/jboss-net.sar/axis.jar/16.axis.jar
Found Jakarta-Commons Discovery (org.apache.commons.discovery.Resource) at
/opt/jboss/jboss-3.2.1/server/all/tmp/deploy/server/all/deploy/jboss-net.sar/commons-discovery.jar/12.commons-discovery.jar
Found Jakarta-Commons Logging (org.apache.commons.logging.Log) at
/opt/jboss/jboss-3.2.1/server/all/tmp/deploy/server/all/deploy/jboss-net.sar/commons-logging.jar/17.commons-logging.jar
Found Log4j (org.apache.log4j.Layout) at /opt/jboss/jboss-3.2.1/client/log4j.jar
Found IBM's WSDL4Java (com.ibm.wsdl.factory.WSDLFactoryImpl) at
/opt/jboss/jboss-3.2.1/server/all/tmp/deploy/server/all/deploy/jboss-net.sar/wsdl4j.jar/14.wsdl4j.jar
Found JAXP implementation (javax.xml.parsers.SAXParserFactory) at an unknown location
Found Activation API (javax.activation.DataHandler) at /opt/jboss/jboss-3.2.1/server/all/lib/activation.jar

Optional Components

Found Mail API (javax.mail.internet.MimeMessage) at /opt/jboss/jboss-3.2.1/server/all/lib/mail.jar

Warning: could not find class org.apache.xml.security.Init from file **xmlsec.jar**
XML Security is not supported
See http://xml.apache.org/security/

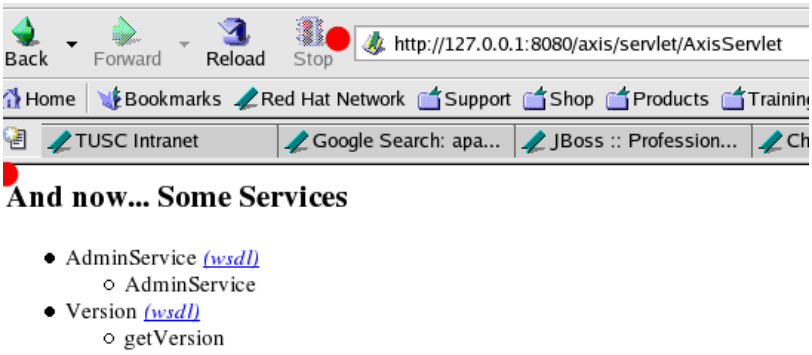
Warning: could not find class org.apache.xml.security.Init from file **xmlsec.jar**
XML Security is not supported
See http://xml.apache.org/security/

If any of the needed libraries are missing, Axis will not work.

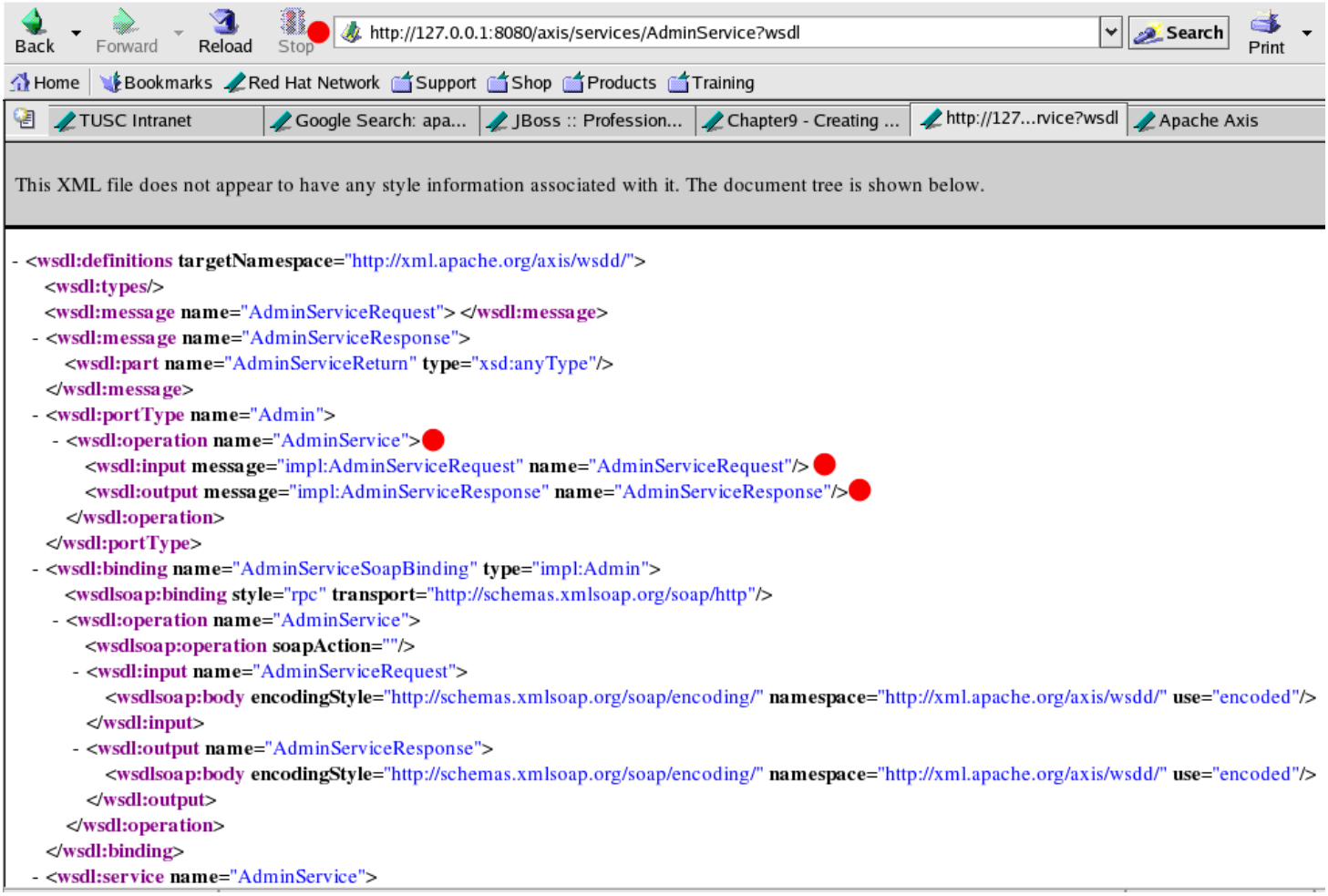
Note : Make sure that all the needed libraries are found, and this validation page is happy.
Optional components are indeed optional; install them as you require. If you see nothing but an internal server error and an exception trace, then you probably have multiple XML parsers on the CLASSPATH (or AXISCLASSPATH), and this is causing version confusion.

Now go back to your Apache AXIS start page and select the list of already deployed Web services. By clicking on '*View the list of deployed Web services*'.

Note : There are a few web services which come with Apache-AXIS bundle to assist us with our learning curve.



In order to make sure that your Web Service is up and running, click on each WSDL (Web Service Description Language) as shown above in figure. Example : If we click on AdminService (WSDL) then the following screen will be available as shown below.



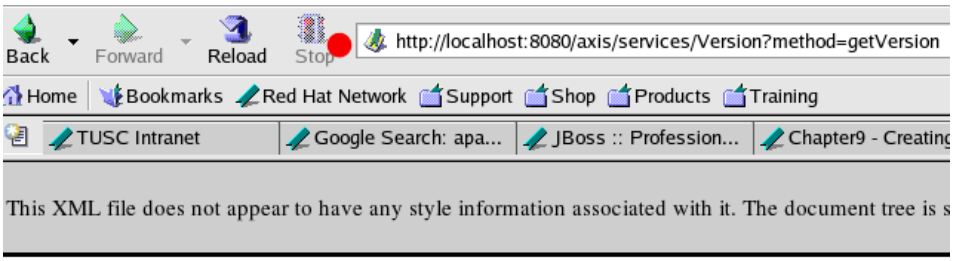
Before we write our first Web Service, let's test one of the working example services which is already deployed as part of the Apache bundle.

Note : For more information look at the documentation for Apache-AXIS under 'Installing and deploying web applications using xml-axis' at http://ws.apache.org/axis/.

Now, SOAP 1.1 uses HTTP POST to submit an XML request to the *endpoint*, but Axis also supports a HTTP GET access mechanism, which is very useful for useful for testing. So to test the service, let's retrieve the version of Axis from the version endpoint, by calling the `getVersion` method:

http://localhost:8080/axis/services/Version?method=getVersion

This will result in the screen shown below.

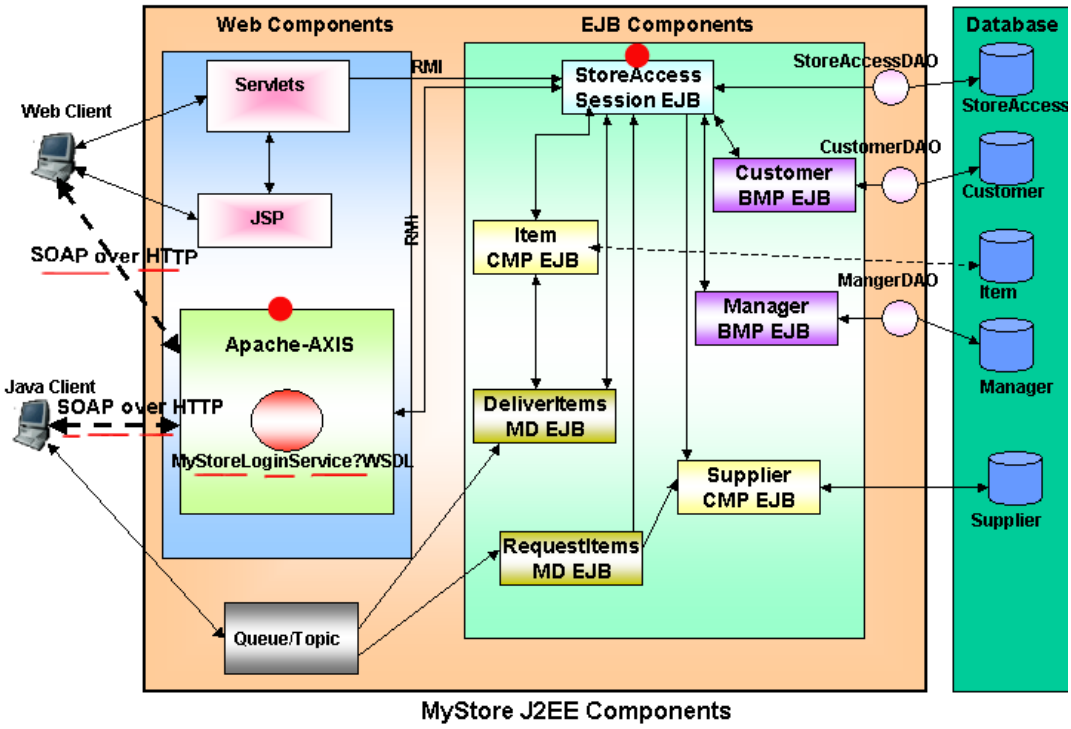



```
- <soapenv:Envelope>
  - <soapenv:Body>
    - <getVersionResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      - <getVersionReturn xsi:type="xsd:string">
          Apache Axis version: 1.0 Built on Oct 07, 2002 (10:43:15 EDT)
        </getVersionReturn>
      </getVersionResponse>
    </soapenv:Body>
  </soapenv:Envelope>
```

After successfully testing the Web Service, let's write our own first Web Service by extending our tutorial application MyStore.

We will create a web service named MyStoreLoginService, which will allow the user to login to MyStore using a Java Client, but using Web service (HTTP) rather then RMI, which is normally the case when we use servlets/JSP or a Java Client. StoreAccessBean will be acting as our endpoint.

Note : Lomboz-2.1_02 witth Eclipse 2.1 and Lomboz 2.1.1 with Eclipse 2.1.1 doesn't have a wizard to create a webservice (that is Web Service Deployment Descriptor [WSDD]) and deploy it. So we will have to do it by hand. Both versions of Lomboz do provide a wizard for creating a SOAP Client.



Create the Web Service :

In order to create a Web Service we need to create an endpoint. In this case, our Stateless Bean StoreAccessBean will be acting as an endpoint. Method loginUser will be invoked from the client side using this Web Service, so we have to create a Web Service Deployment Descriptor (WSDD) file describing the signature of the loginUser method along with some other required information.

Note : Currently, only a Stateless Bean can be used as an endpoint for a Web Service as specified in JSR-109.

WSDD is an AXIS specific file, used to create a skeleton for JAX-RPC. This is transparent to the developer, and once this skeleton is created and deployed, AXIS generates WSDL and publishes the Web-Service.

For more information : http://ws.apache.org/axis/.

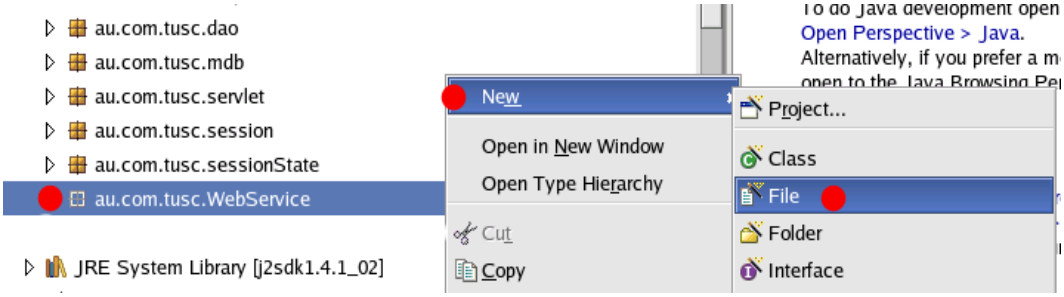
Now, under Eclipse, go to Package Explorer > Expand Mystore (project) node > select src, right click and a menu will pop up.

On the pop up menu > New > Select Package > Add name of package as 'au.com.tusc.WebService' .

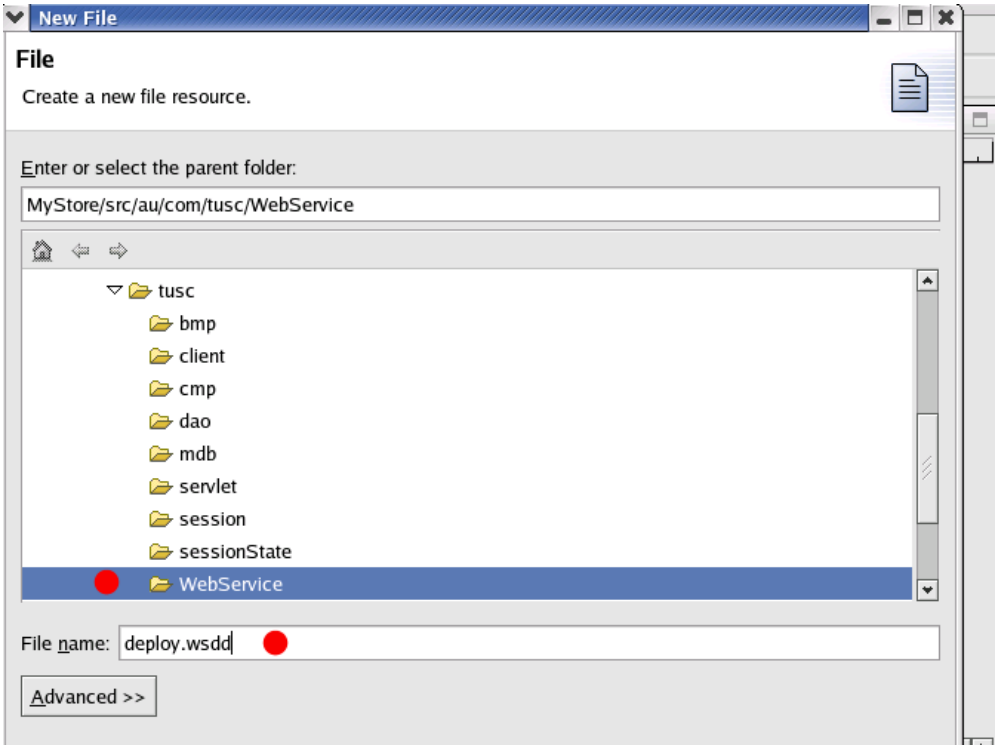
Press Finish.

This will create a new package named au.com.tusc.WebService under 'src'.

Go to src/au.com.tusc.WebService > Select New > File as shown below.



Now create a file named deploy.wsdd under the package au.com.tusc.WebService as shown below.



Add the following XML tags as shown below in the deploy.wsdd file, and then we will examine the contents of the file..

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/" xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="MyStoreLoginService" provider="java:EJB">
    <parameter name="beanJndiName" value="StoreAccessLocal"/>
    <parameter name="homeInterfaceName" value="au.com.tusc.session.StoreAccessLocalHome"/>
    <parameter name="remoteInterfaceName" value="au.com.tusc.session.StoreAccessLocal"/>
    <parameter name="allowedMethods" value="loginUser"/>
    <parameter name="jndiURL" value="jnp://localhost:1099"/>
    <parameter name="jndiContextClass" value="org.jnp.interfaces.NamingContextFactory"/>
  </service>
</deployment>
```

Lets analyze these deployment descriptors.

<service name="MyStoreLoginService" provider="java:EJB">

The service name tag is responsible for giving the name to the service and the provider. Provider states that the web service is an EJB-Web Service. The name of our service is **MyStoreLoginService** and the provider is **java:EJB** .

<parameter name="beanJndiName" value="StoreAccessLocal"/>

The first parameter is **beanJndiName** which specifies where the bean (EJB) is in the JNDI tree.
We are using the local interfaces of StoreAccessBean, so it is **StoreAccessLocal**.

Note : We used non-local interfaces for StoreAccessBean until chapter 8, where the Bean is accessed by Web Components and Java Clients. As all the component are in the same JVM, we are typically using local interfaces as these interfaces have many advantages compared to remote interfaces. These are:

1. *Simpler programming model, no need to catch remote exceptions that can't happen in co-located deployment.*

2. *Since RemoteException is not needed on every method signature, your EJB business method's interface is independent of EJB. This can free callers from dependence on the EJB API.*

3. *As it is running in the same JVM and not in a distributed environment, although coded to call-by-value it is running with call-by-reference.*

<parameter name="homeInterfaceName" value="au.com.tusc.session.StoreAccessLocalHome"/>

The parameter above is called **homeInterfaceName** and it has as it's value the Remote Home Local Interface which is **au.com.tusc.session.StoreAccessLocalHome**.

<parameter name="remoteInterfaceName" value="au.com.tusc.session.StoreAccessLocal"/>

The parameter above is called **remoteInterfaceName** and it has as it's value the Remote Local Interface which is **au.com.tusc.session.StoreAccessLocal.**

<parameter name="allowedMethods" value="loginUser"/>

The **allowedMethods** parameter has the value **loginUser** to indicate that only this method will be made public or can be accesed from client. The rest of the parameters are specific to the application server, in this case JBOSS.

So, now our deploy.wsdd is complete, let's deploy it.

Note : Currently Lomboz doesn't have a wizard for creating the WSDD file and to deploy it. So that's why we have to create it by hand.

Deploy the Web Service :

In order to deploy this we will have to go to our Linux console (If windows then DOS-console).

Now, assuming the application server (JBOSS) is running and our bean has bean deployed (All the beans are in MyStore.jar as covered in previous chapters).

Go into the directory where deploy.wsdd has been created as shown below.

```
[vishal@vishal WebService]$ pwd
/home/vishal/workspace/MyStore/src/au/com/tusc/WebService
[vishal@vishal WebService]$ ls
deploy.wsdd
[vishal@vishal WebService]$
```

Note : Make sure that $AXISCLASSPATH is set properly as mentioned above.

```
[vishal@vishal WebService]$ echo $AXISCLASSPATH

/opt/axis/axis-1_1/lib/axis.jar:/opt/axis/axis-1_1/lib/jaxrpc.jar:/opt/axis/axis-1_1/lib/saaj.jar:
/opt/axis/axis-  1_1/lib/wsdl4j.jar:/opt/axis/axis-1_1/lib/commons-discovery.jar:
/opt/axis/axis-1_1/lib/commons-logging.jar:.

[vishal@vishal WebService]$
```

Now we will be using the AdminClient (org.apache.axis.client.AdminClient) class,which is part of the AXIS API's, to deploy the WSDD.

Note : By default it uses port 8080, but if you have to use different port number then use the -p <port> argument.

Use this command to deploy WSDD as shown below.

```
[vishal@vishal WebService]$ java org.apache.axis.client.AdminClient deploy.wsdd
Processing file deploy.wsdd
<Admin>Done processing</Admin>
[vishal@vishal WebService]$
```

If the message shown above is displayed then your web service is deployed succesfully.

Note : An admin client can be used both from the command line and programmatically.

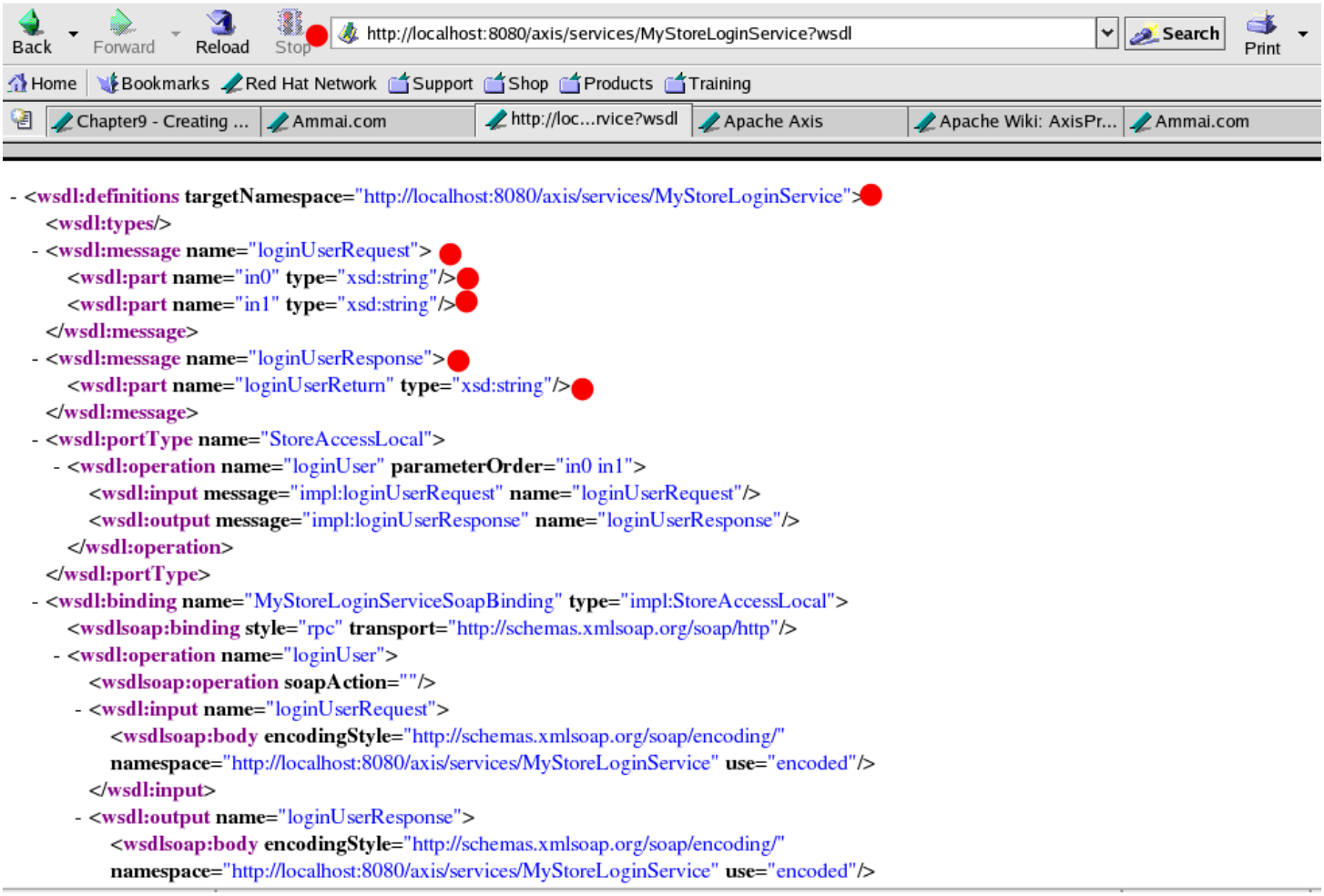
Now, let's access our web service and its corresponding Web Service Description Language (WSDL).

Go to the following URL http://localhost:8080/axis/servlet/AxisServlet to see the all the deployed services as shown below.

And now... Some Services

- AdminService *(wsdl)*
  - AdminService
- Version *(wsdl)*
  - getVersion
- MyStoreLoginService *(wsdl)*
  - loginUser

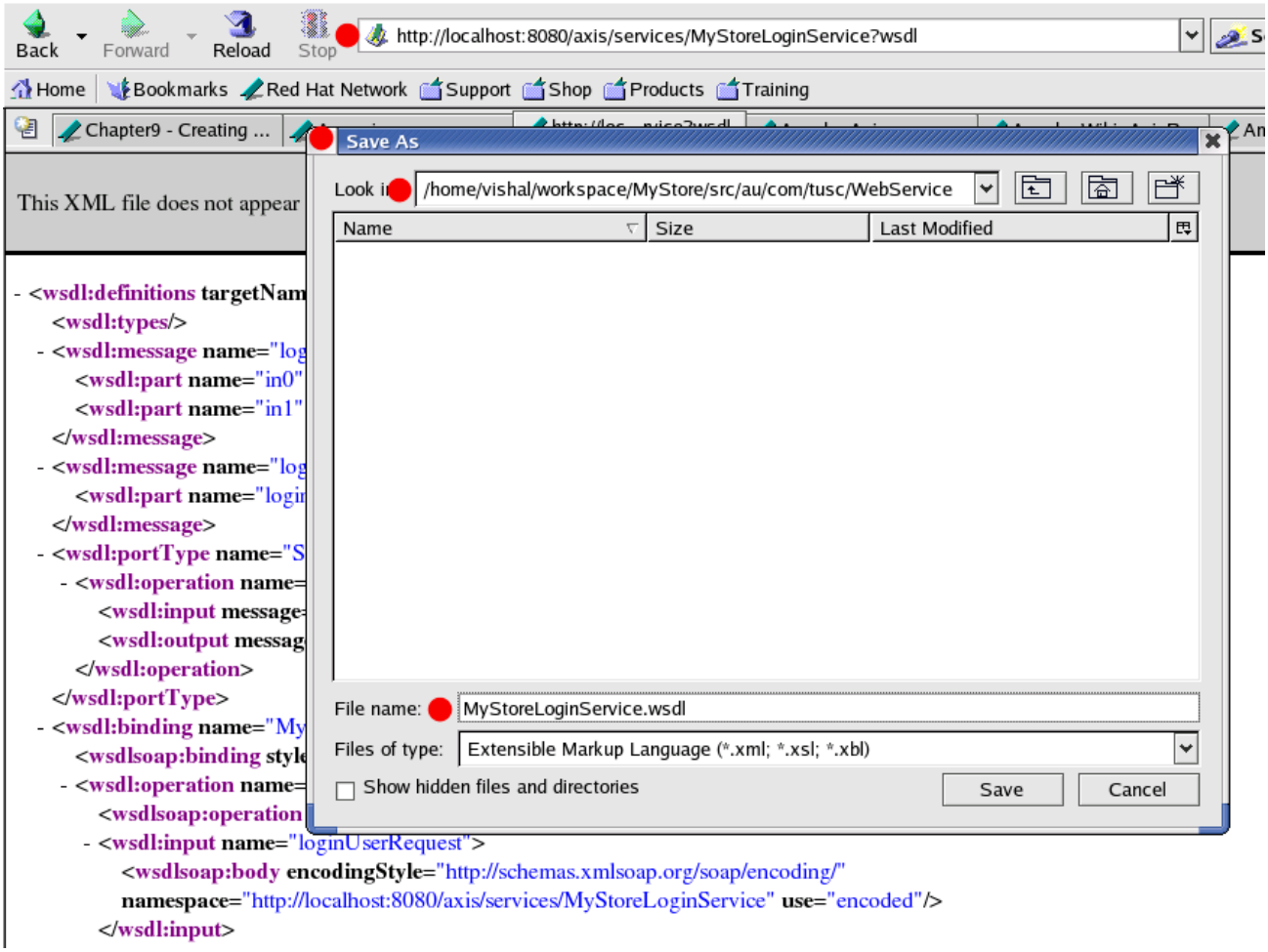
Go to wsdl and it will show the WSDL generated by AXIS for the deployed Web Service as shown below.



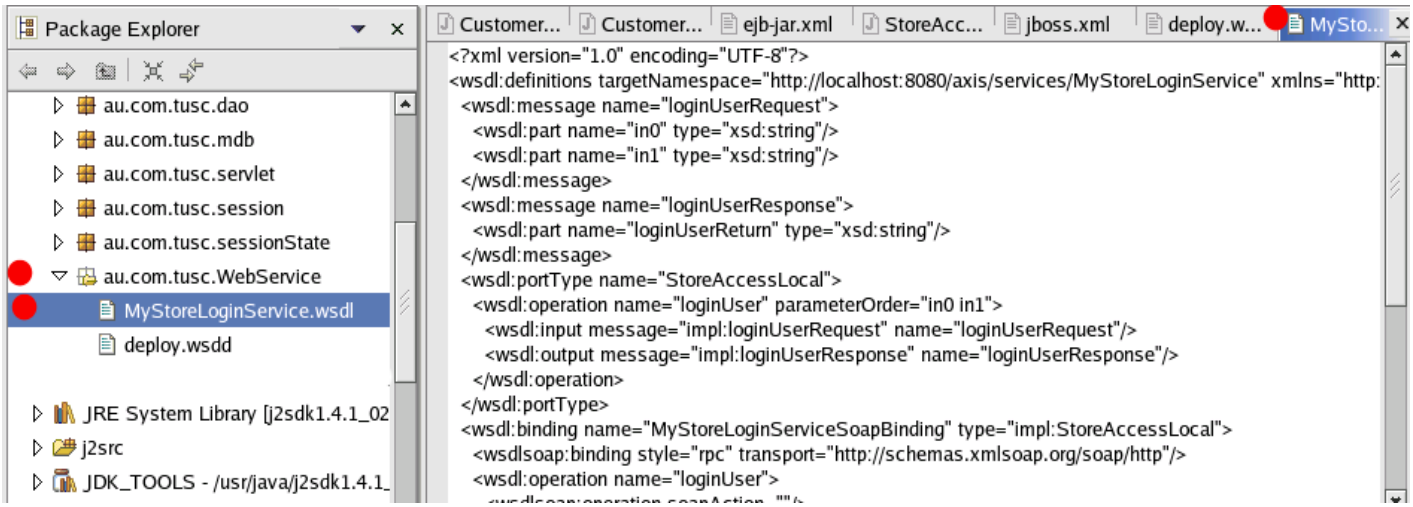

```
- <wsdl:definitions targetNamespace="http://localhost:8080/axis/services/MyStoreLoginService">
    <wsdl:types/>
  - <wsdl:message name="loginUserRequest">
      <wsdl:part name="in0" type="xsd:string"/>
      <wsdl:part name="in1" type="xsd:string"/>
    </wsdl:message>
  - <wsdl:message name="loginUserResponse">
      <wsdl:part name="loginUserReturn" type="xsd:string"/>
    </wsdl:message>
  - <wsdl:portType name="StoreAccessLocal">
    - <wsdl:operation name="loginUser" parameterOrder="in0 in1">
        <wsdl:input message="impl:loginUserRequest" name="loginUserRequest"/>
        <wsdl:output message="impl:loginUserResponse" name="loginUserResponse"/>
      </wsdl:operation>
    </wsdl:portType>
  - <wsdl:binding name="MyStoreLoginServiceSoapBinding" type="impl:StoreAccessLocal">
      <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    - <wsdl:operation name="loginUser">
        <wsdlsoap:operation soapAction=""/>
      - <wsdl:input name="loginUserRequest">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://localhost:8080/axis/services/MyStoreLoginService" use="encoded"/>
        </wsdl:input>
      - <wsdl:output name="loginUserResponse">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://localhost:8080/axis/services/MyStoreLoginService" use="encoded"/>
```

Now save this WSDL as it will be required to generate the SOAP Client using Lomboz's SOAP Client creation wizard. Save this file under the package **au.com.tusc.WebService** as shown below.

Note : The Lomboz SOAP client wizard uses the "WSDL2Java" tool, which will build Java proxies and skeletons for services with WSDL descriptions. This tool comes with the Apache-AXIS distribution.



Now go back to Eclipse, and refresh your project. **MyStore** and **MyStoreLoginService.wsdl** will be present now as shown below.



Since the Web Service is deployed successfully it's time to create the test client.

Create Web Service Test Client :

Now, Lomboz (Both versions 2.1_02 & 2.1.1) provides a wizard for Creating SOAP clients for published web services using their corresponding WSDL files.

But before we do that, one necessary step is to make sure we have all the libraries on the classpath neccesary for generateing the stub for the SOAP Client. These are the same libraries used for $AXISCLASSPATH covered above in the chapter.

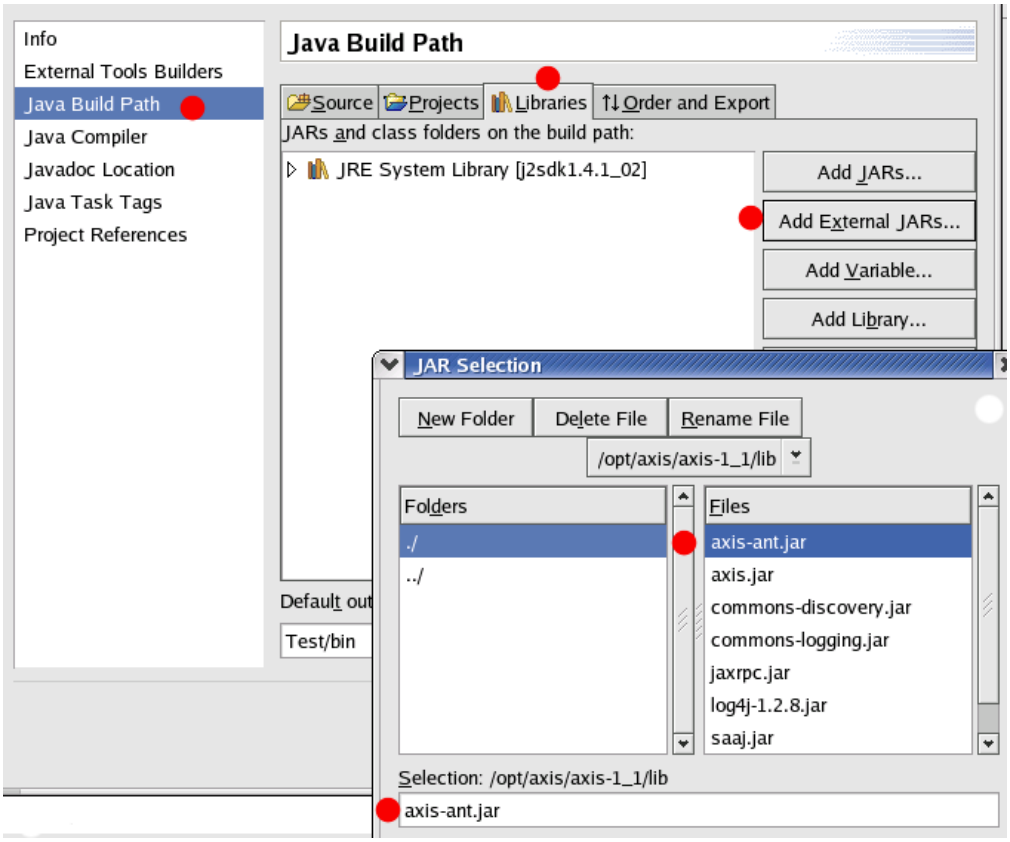
Note : This can be configured by adding the neccessary libraries to the jboss321all.server file used for configuration of Lomboz with Eclipse, which was covered in chapter 1.

To import the required libraries on the projects classpath,

Go to Project Explorer select project MyStore > right click > select properties.

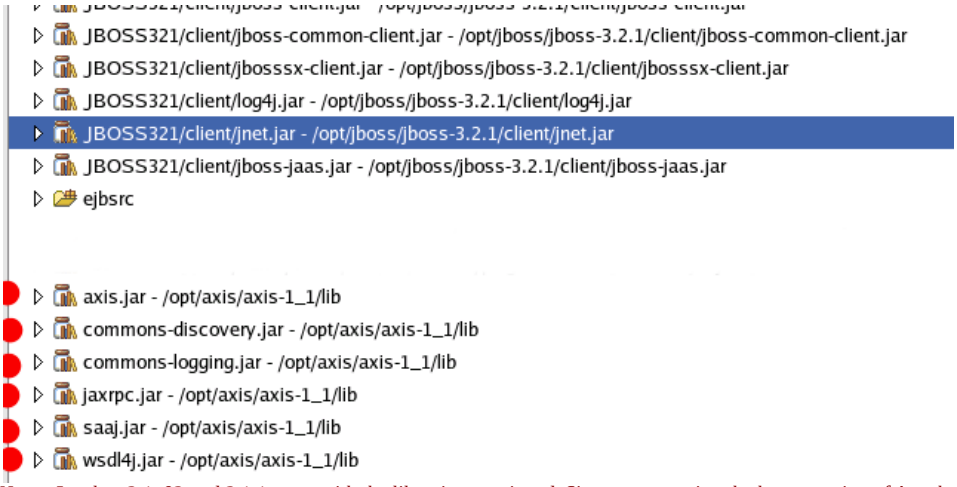
Under Properties > select Java Build Path > select tab Libraries.

Press Add External jars as shown below.



The following libraries are imported as shown below:

axis.jar, jaxrpc.jar, saaj.jar, wsdl4j.jar, commons-discovery.jar, commons-logging.jar

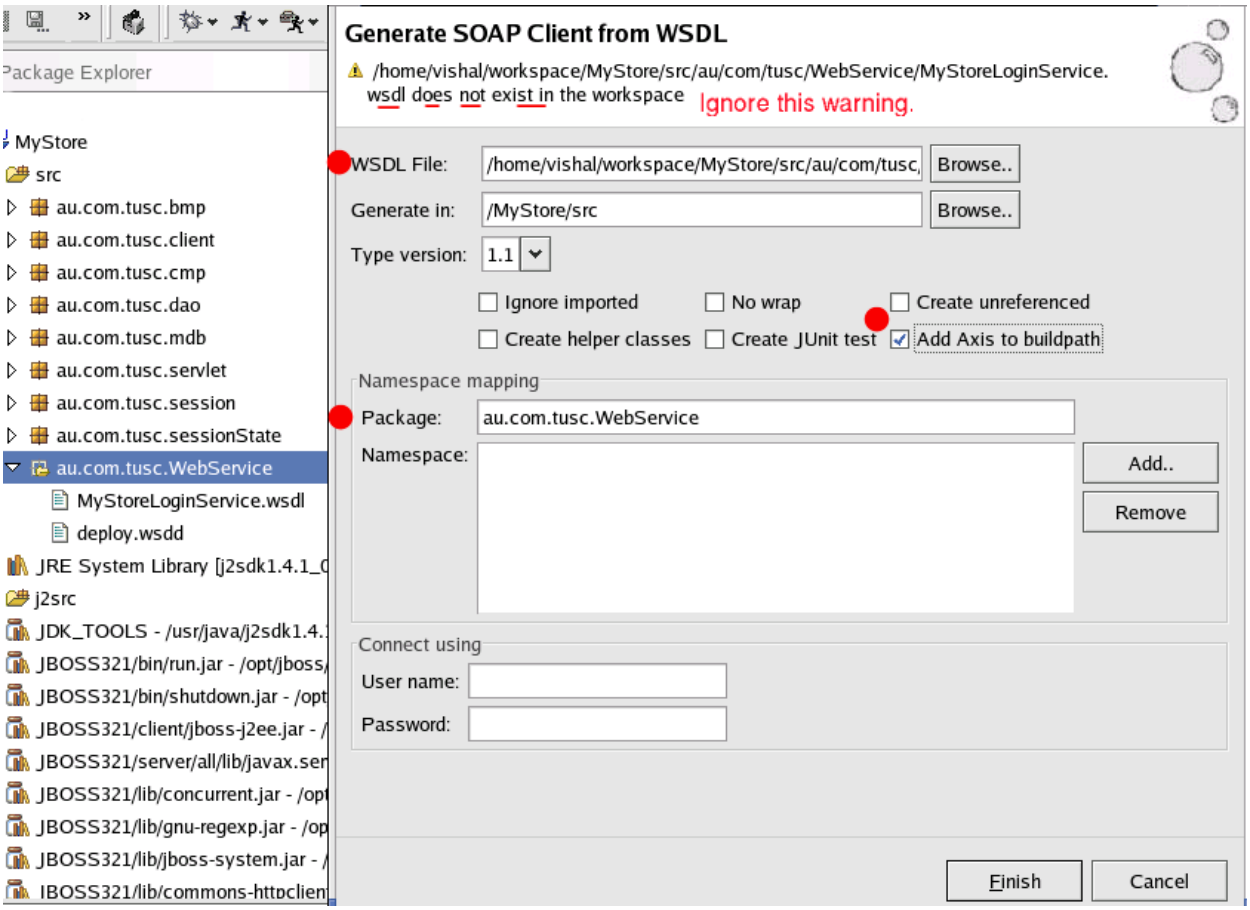


Note : Lomboz 2.1_02 and 2.1.1 come with the libraries mentioned. Since we are using the latest version of Apache-AXIS (V1.1), it is better to import these new libraries instead.

Now under Project Explorer go to src/au.com.tusc.WebService> right click.

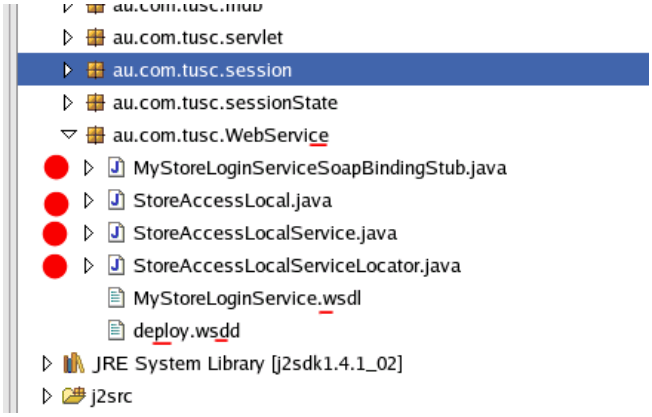
On the pop up menu select New > select Lomboz SOAP client Wizard.

Under the SOAP Client Wizard select 'MyStoreLoginService.wsdl'. Also select the checkbox with Add Axis to buildpath. And under namespace mapping select the package name 'au.com.tusc.WebService' as shown below in figure.



Press Finish.

This will create the stub for the client comprising four files under au.com.tusc.WebService as shown below.



Now let's write the test client.

So, Go to src/au.com.tusc.WebService and add a new file named TestClient.java.

Add the following code to this (TestClient.java) file as shown below. We will analyze this code later.

```
/*
 * Created on Sep 8, 2003
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
package au.com.tusc.WebService;
import org.apache.axis.AxisFault;

public class TestClient {

    public static void main(String[] args) {

        try {

          // Make a service
        StoreAccessLocalServiceLocator service = new StoreAccessLocalServiceLocator();
        StoreAccessLocal  port =  service.getMyStoreLoginService();

          // Make the actual calls to the method
        String userID = port.loginUser("ANDY","PASSWD");

          System.out.println("USERID  recvd is : " + userID);
        }
        catch (AxisFault af) {
           System.err.println("An Axis Fault occurred: " + af);
           System.out.println(" AXIS exception is" + af.dumpToString());
        }
        catch (Exception e)  {
           System.err.println("Exception caught: " + e);
        }
    }
}
```

Now let's examine the 2 main statements.

StoreAccessLocalServiceLocator service = **new** StoreAccessLocalServiceLocator();

This is necessary as the ServiceLocator object returns the endpoint for the web service which is used for making the call. In this example the end point is "http://localhost:8080/axis/services/MyStoreLoginService";

StoreAccessLocal port = service.getMyStoreLoginService();

This returns the reference of the local interface for the bean so that the necessary methods can be invoked.

Note : There are four different styles of service in Axis 1.1 as specified in the user guide that comes with the Apache-AXIS distribution.

1. *RPC services use the SOAP RPC conventions, and also the SOAP "section 5" encoding. The test client which we have created above is using SOAP RPC style.*

2. *Document services do not use any encoding (so in particular, you won't see multiref object serialization or SOAP-style arrays on the wire) but DO still do XML<->Java databinding.*

3. *Wrapped services are just like document services, except that rather than binding the entire SOAP body into one big structure, they "unwrap" it into individual parameters. The web clients covered later on are using this style of service.*

4. *Message services receive and return arbitrary XML in the SOAP Envelope without any type mapping / data binding. If you want to work with the raw XML of the incoming and outgoing SOAP Envelopes, write a message service.*

   For more information please refer to the use guide.

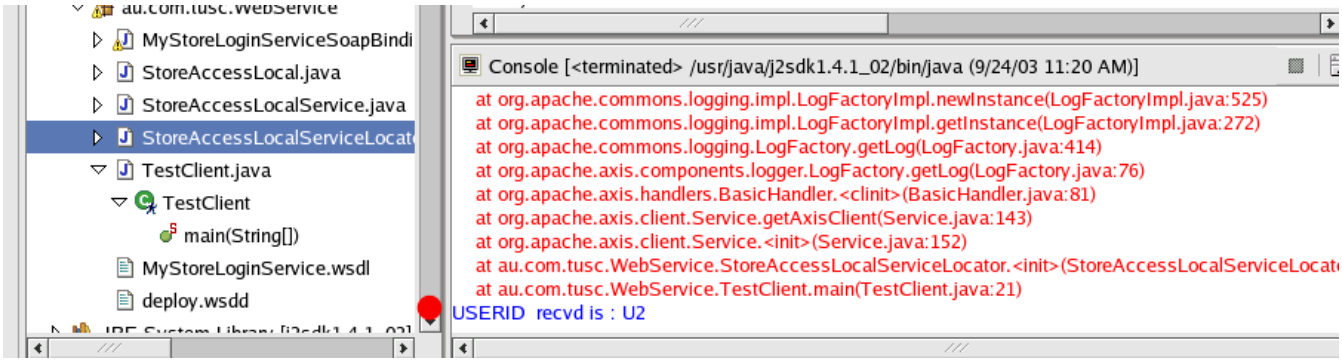
Test your Client :

To test your client, select au.com.tusc.WebService.TestClient node > Go to the top level menu and select the 'Running Man' icon.

On that select 'Run as' > select Java Application.

Note : The steps to run test clients have been covered in previous chapters.

Now, under your console, If you get UserID as as 'U2' then your call was successful, as shown below.



We have learned how to create, deploy and access web services using a Java client. Now that same web service can be accessed using web components like servlets (JSP/ Servlets).

There are three different ways of creating web service clients which are

1. *Using the Static Stub Client implemented in the client example discussed above.*
2. *Dynamic Proxy Client.*
3. *Dynamic Invocation Interface (DII) Client used in the Web client implemented in next section.*

Create Web Client :

To create a web client all the necessary steps have been covered in chapter 8, so leaving those details, we can straight away add the following lines to our servlet named '**Login**' under **au.com.tusc.WebService** as shown below in this code snippet from Login.java.

```
package au.com.tusc.WebService;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import javax.xml.namespace.QName;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;

/**
  * @author vishal
  *
  * To change the template for this generated type comment go to
  * Window>Preferences>Java>Code Generation>Code and Comments
  */
public class Login extends HttpServlet {

        public void init(ServletConfig config) throws ServletException {
                super.init(config);
                System.out.println("Entering Login.init()") ;
                System.out.println("Leaving Login.init()") ;
        }

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

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

       protected void processRequest (HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {
                System.out.println("Entering Login.processRequest()");
                response.setContentType("text/html");
                PrintWriter out = response.getWriter ();
                out.println("<html><title>MyStore Login</title>");
                out.println("<body><b><h2>Welcome to MyStore </h2></b></body>");
                out.print("<body><h2>Login details : UserID is :   ");
                try {
                        String username =    callWebService();
                        out.print(username);
                } catch (Exception e) {
                        System.out.println("Exception in Login.processRequest ");
                }
```

```
            out.print("</h2></body></html>");
            if ( out != null ) out.close();
            System.out.println("Leaving Login.processRequest()");
        }

        private String callWebService() throws Exception {

            // URL is used for creating the call, which uses the WSDL.
            Call call = new Call( new URL("http://localhost:8080/axis/services/MyStoreLoginService?wsdl") );

            // Calls the object, passing in the username and passwd. The return value is stored as an object.
            String username = (String) call.invoke ("loginUser", new Object[] { new String("ANDY") , new String("PASSWD") } );
            return username;
        }
}
```

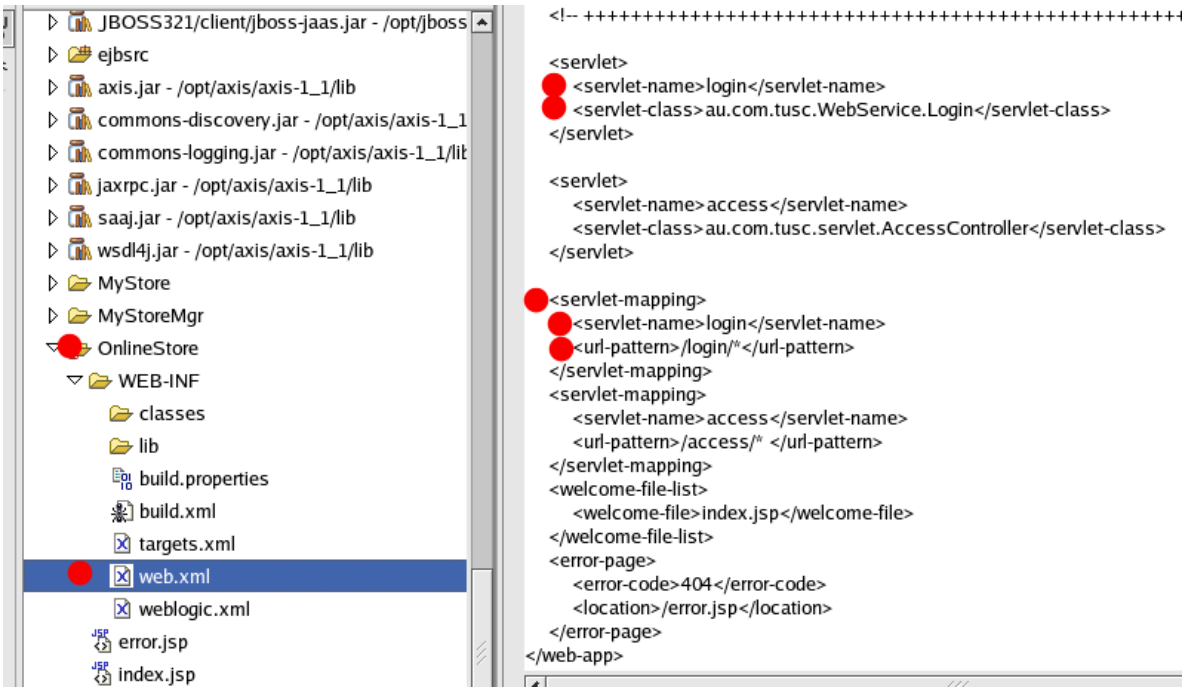
Note : Another way of invoking the loginUser method is to use Axis' JAXRPC Dynamic Invocation Interface implementation of the Service interface. The Service class should be used as the starting point for accessing SOAP Web Services. Typically, a Service will be created with a WSDL document, and along with a serviceName you can then ask for a Call object that will allow you to invoke the Web Service. Shown below is the other way of implementing the callWebService() method. Both versions of callWebService() method use the DII approach.

```
        private String callWebService() throws Exception {

            // Endpoint is used for making the call
            String endpoint = "http://localhost:8080/axis/services/MyStoreLoginService";

            // The Service object is the starting point for accessing the web service.
            Service service = new Service();

            // The call object is used to actually invoke the web service.
            Call call = (Call)service.createCall();

            // Sets the call objects endpoint address
            call.setTargetEndpointAddress(endpoint);

            // Sets the operation name associated with this Call object.
            call.setOperationName(new QName("loginUser"));

            // Calls the object, passing in the username and passwd. The return value is stored as an object.
            Object returnValue = call.invoke(new Object[] { new String("ANDY") , new String("PASSWD") });
            return (String) returnValue;
        }
```

Note: The difference between the two approaches is that the Call interface provides support for the dynamic invocation of a service endpoint whereas the Service class acts as a factory for the Dynamic proxy for the target service endpoint.

Also shown below is **web.xml** where the servlet's access name and its mapping are shown.

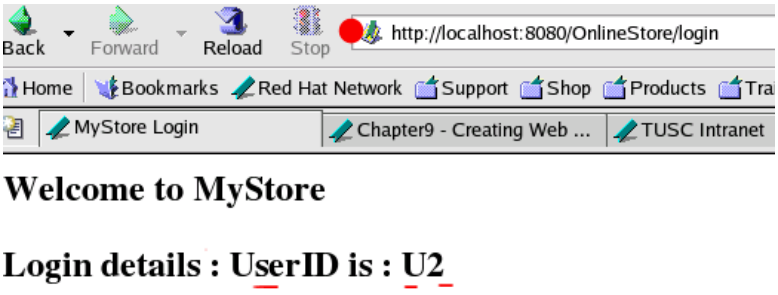
Note : The steps to create a servlet are covered in chapter 8. Servlet Login is included in the OnlineStore web module.


JBOSS321/client/jboss-jaas.jar - /opt/jboss
ejbsrc
axis.jar - /opt/axis/axis-1_1/lib
commons-discovery.jar - /opt/axis/axis-1_1
commons-logging.jar - /opt/axis/axis-1_1/lib
jaxrpc.jar - /opt/axis/axis-1_1/lib
saaj.jar - /opt/axis/axis-1_1/lib
wsdl4j.jar - /opt/axis/axis-1_1/lib
MyStore
MyStoreMgr
OnlineStore
WEB-INF
classes
lib
build.properties
build.xml
targets.xml
web.xml
weblogic.xml
error.jsp
index.jsp
<!-- ++
<servlet>
<servlet-name>login</servlet-name>
<servlet-class>au.com.tusc.WebService.Login</servlet-class>
</servlet>
<servlet>
<servlet-name>access</servlet-name>
<servlet-class>au.com.tusc.servlet.AccessController</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>login</servlet-name>
<url-pattern>/login/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>access</servlet-name>
<url-pattern>/access/* </url-pattern>
</servlet-mapping>
<welcome-file-list>
<welcome-file>index.jsp</welcome-file>
</welcome-file-list>
<error-page>
<error-code>404</error-code>
<location>/error.jsp</location>
</error-page>
</web-app>
```

Now once you have completed the servlet, deploy the web module OnlineStore. If there are no error messages in the console then the web module has been deployed succesfully.
```

Note : The steps are covered in chapter 8 on how to deploy web modules.

To invoke the login method on the published web service go to this url http://localhost:8080/OnlineStore/login.

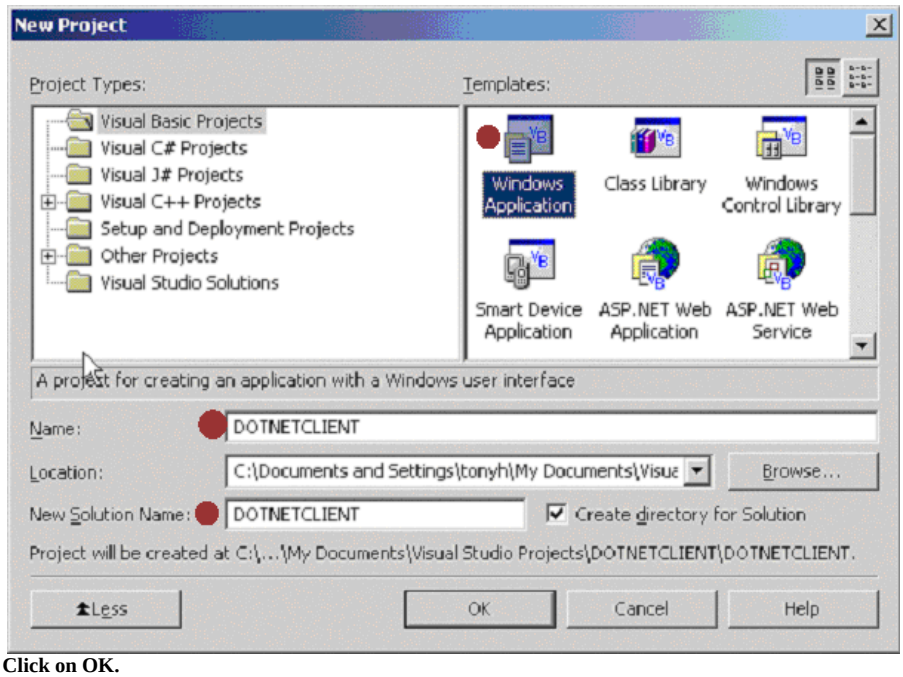
If you get the following screen shown below then your call is succesfull.



Until now we have seen how to access web services using web clients and Java clients from the same environment. Now to really experience the amazing power of Web Services we will invoke this web service using a .NET (Microsoft Client).

Create VB.Net Client :

Now, to create a VB.Net client create a new Windows Application Project for Visual Basic.NET named DOTNETCLIENT as shown below in the figure.



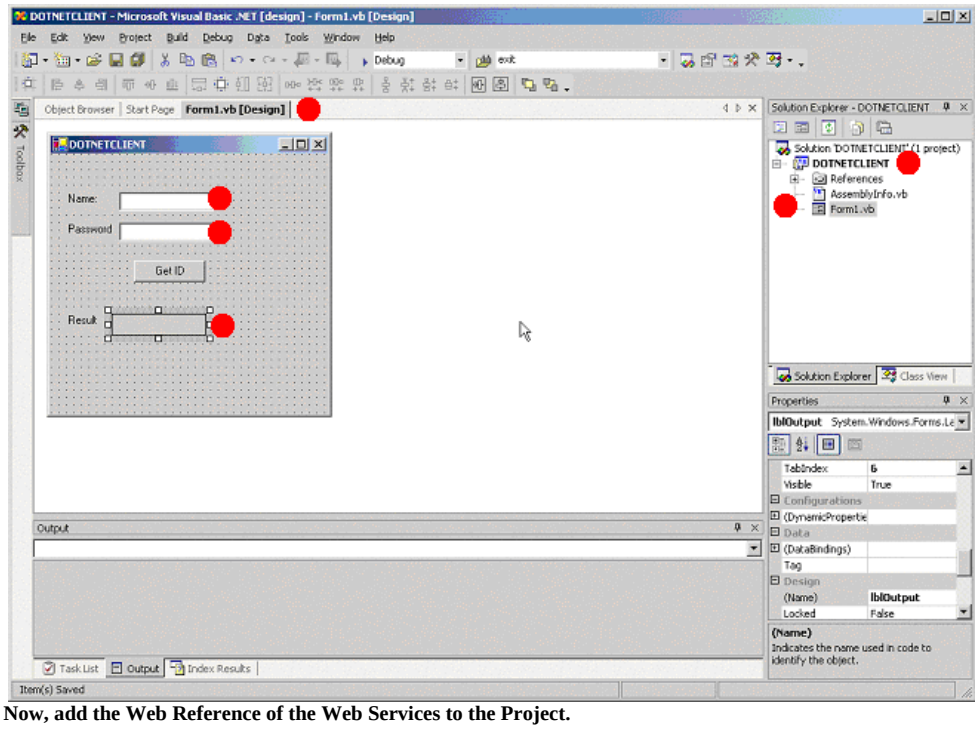
Click on OK.

Change the form name property to DOTNETCLIENT.

Add the following controls to the form with the following properties as shown below in the figure.

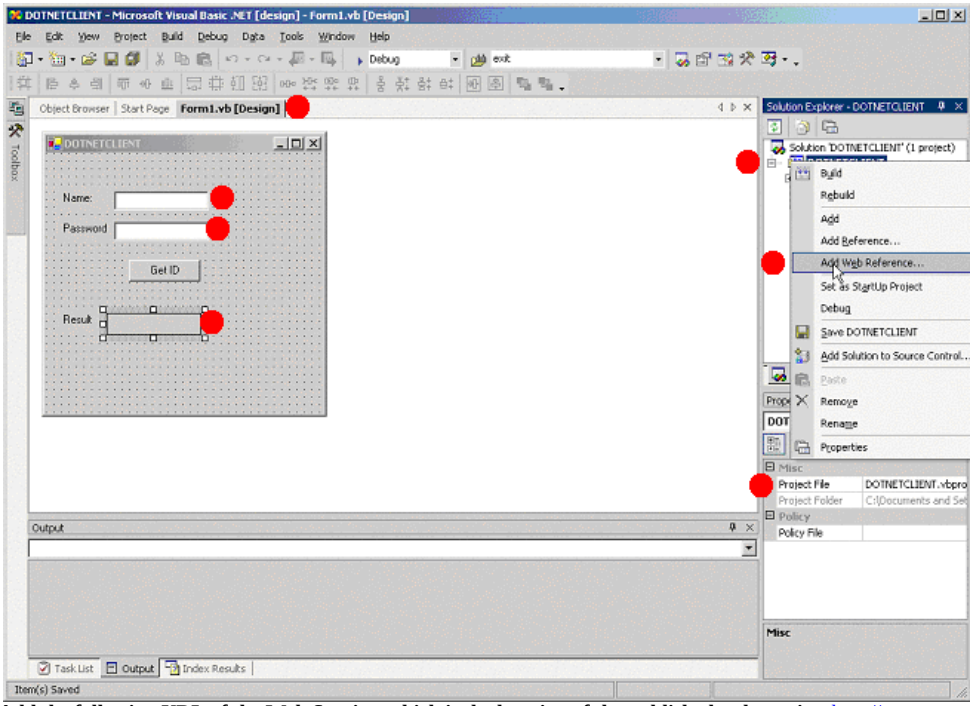
```
Label Control
Name: lblName
Text: Name

Label Control
Name: lblPassword
Text: Password
Label Control
Name: lblResult
Text: Result

Label Control
Name: lblOutput
Text:

TextBox Control
Name: TextBoxName
Text:

TextBox Control
Name: TextBoxPassword
Text:
```

```
PasswordChar: *

Button Control
Name: btnGetID
Text: Get ID
```



Now, add the Web Reference of the Web Services to the Project.

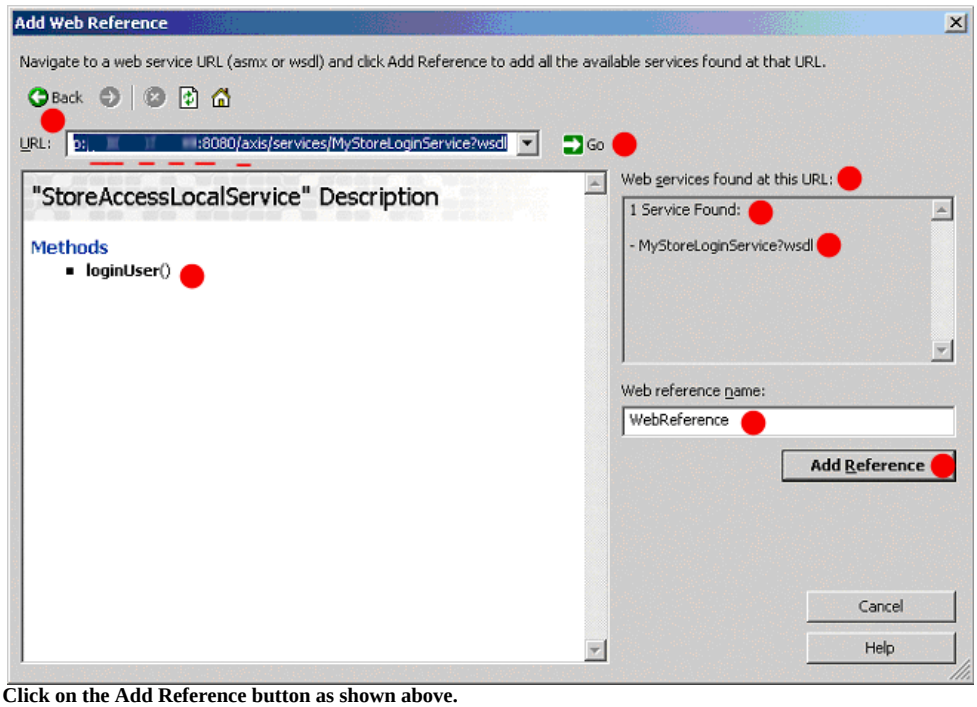
Right-click on the DOTNETCLIENT from the Solution Explorer and click on Add Web Reference as shown below.



Add the following URL of the Web Service, which is the location of the published web service. http://xxx.yyy.zzz.aaa:8080/axis/services/MyStoreLoginService?wsdl.

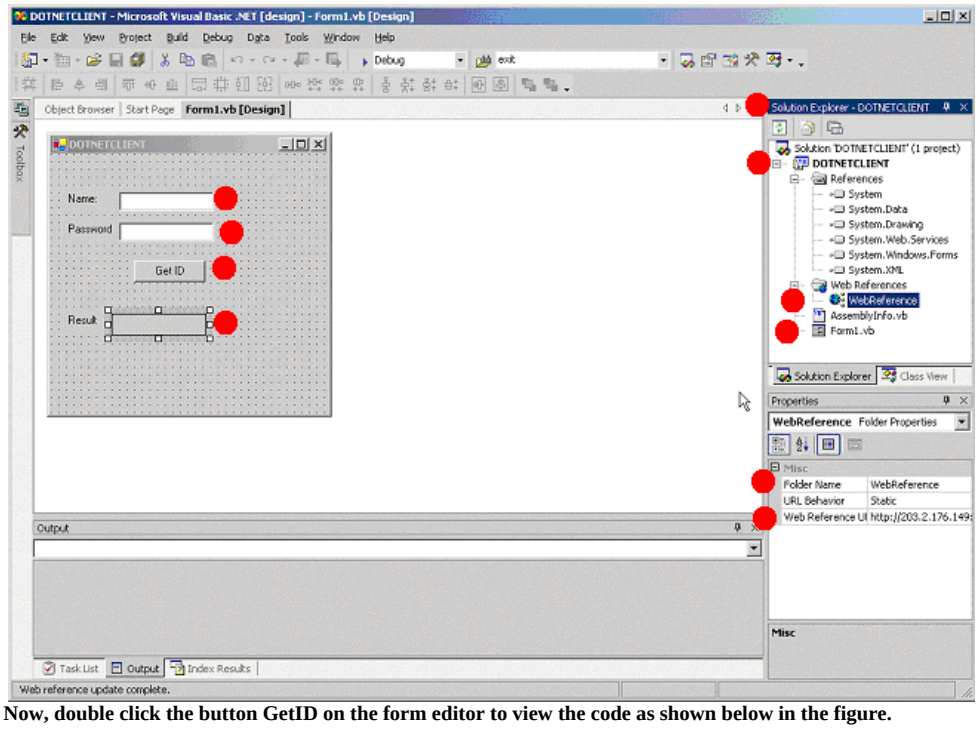
Note : This URL is pointing to the machine where the web service is published.

Click on GO - as shown below in the figure it will show the service with its method(s) available under it.



Click on the Add Reference button as shown above.

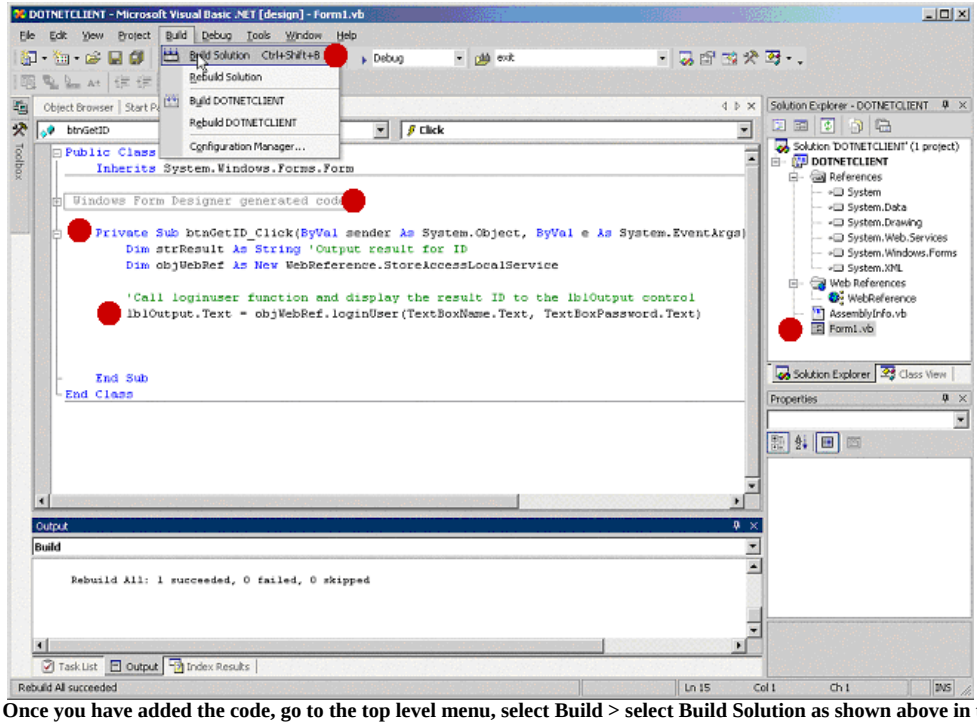
This will bring a new screen with Web Reference added to the project via Solution Explorer as shown below.



Now, double click the button GetID on the form editor to view the code as shown below in the figure.

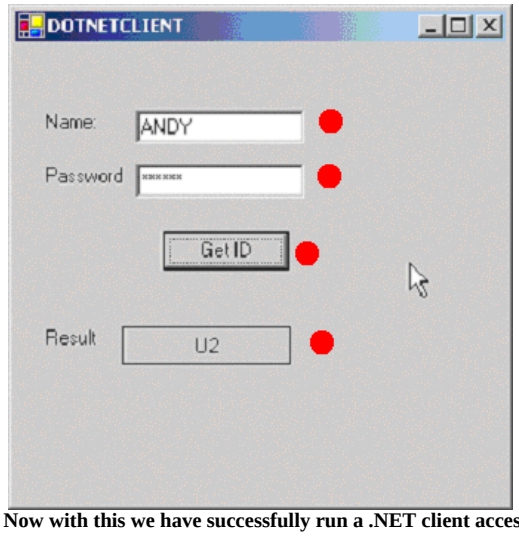
Add the following code for the Sub btnGetID_Click().

```
Dim strResult As String 'Output result for ID

Dim objWebRef As New WebReference.StoreAccessLocalService

'Call loginuser function and display the result ID to the lblOutput control
lblOutput.Text = objWebRef.loginUser(TextBoxName.Text, TextBoxPassword.Text)
```



Once you have added the code, go to the top level menu, select Build > select Build Solution as shown above in the figure.

Now, start the application and you will get this form shown below. Enter Name as 'ANDY' and Password as 'PASSWD' . Press the GetID button and the Result should be 'U2'.



Now with this we have successfully run a .NET client accessing our web services

Create Perl Client :

To create a Perl client we need the SOAP::Lite module to invoke methods on deployed web services.

First of all download the latest version of SOAP::Lite and follow the standard Perl module installation procedures by entering the very familiar command sequence:

```
perl Makefile.PL
make
make test
make install
```

If you have the CPAN.pm module installed on your machine and you are connected to the Internet, then run the following sequence of commands as shown below:

```
[vishal@vishal vishal]$ perl -MCPAN -e shell
cpan shell -- CPAN exploration and modules installation (v1.61)
ReadLine support available (try 'install Bundle::CPAN')
cpan> install SOAP::Lite
CPAN: Storable loaded ok
CPAN: LWP::UserAgent loaded ok
Fetching with LWP:
ftp://mirror.aarnet.edu.au/pub/perl/CPAN/authors/01mailrc.txt.gz
Going to read y/sources/authors/01mailrc.txt.gz
CPAN: Compress::Zlib loaded ok
Fetching with LWP:
----------------------------------------------
----------------------------------------------
```

```
-----------------------------------------------
CPAN: Digest::MD5 loaded ok
Fetching with LWP:
ftp://mirror.aarnet.edu.au/pub/perl/CPAN/authors/id/K/KU/KULCHENKO/CHECKSUMS
Checksum for y/sources/authors/id/K/KU/KULCHENKO/SOAP-Lite-0.55.tar.gz ok

cpan> bye
Lockfile removed.
[vishal@vishal vishal]$
```

If you are using ActiveState Perl on Windows the equivalent would be the 'ppm' command.

Note : For more information on the SOAP::Lite module please refer to this site http://www.soaplite.com/beta/install.html.

Now let's create a Perl client to invoke the logInUser method on the deployed web service named MyStoreLoginService.

Create a file named webServiceClient.pl and add the following code shown below.

```
#
# WebServiceClient.pl
#
# Test client for MyStoreLoginService Web service
# Apache Axis - Chapter 9 'Creating Web Services'
#
use SOAP::Lite ;

my $myStoreLoginService = SOAP::Lite->service('http://localhost:8080/axis/services/MyStoreLoginService?wsdl');
my $username = $myStoreLoginService->loginUser("ANDY","PASSWD");
print "MyStore login details : Andy's username is : $username \n";
```

Save this file, and execute it to see the result shown below.

```
[vishal@vishal vishal]$ perl WebServiceClient.pl
MyStore login details : Andy's username is : U2
[vishal@vishal vishal]$
```

Now with this we have successfully run a Perl client accessing the web service.

Prev TOC Next