Enterprise JavaBeans Programming (20 Hours)

Application Server Concepts and Choices

Tomcat

Tomcat is the official reference implementation of the Java Servlet and JavaServer Pages technologies and is part of the Apache Foundation Software system. Essentially, it is a web server with the ability to run both Java Server Pages (JSP) and Java Servlets. Many of the popular application server come bundled with Tomcat, since the ability to deliver web pages that make use of Java code is very important, especially when your developers are already writing code using the Java language.

A Java Servlet is a mini-application that generates HTML pages dynamically. It works very similar to Common Gateway Interface (CGI) applications. A CGI application can be written in just about any language, but instead of outputting text screens, GUI screens or generating data files, the application generates hypertext markup language (HTML), which the web server captures and transmits to the web browser. The full power of the Java environment is available to the servlet.

The JSP technology works a bit differently. Instead of an application that generates HTML for transmission to the user, a JSP document allows developers to mix HTML and Java code within the same file (usually ending with the .JSP suffix). Whenever the user requests that document, the web server scans the file and executes an Java code contained within the HTML. Special tags are used to define which parts of the document are standard HTML and which parts are Java code. Any output generated by the Java code is captured by the web server and is integrated into the HTML prior to transmission to the client.

Both technologies are popular ways to execute Java code on the web server. In both cases, the end result is the ability to generate web pages with dynamic content, perhaps from databases, files or even external hardware devices.

Tomcat is often used in a 3-tier systems, where the client's web browser communicates with the Tomcat web server, which then communicates with database (or other types) servers. In addition,

some application servers include Tomcat to handle the web server side of developing dynamic web sites.

JBoss

JBoss is a very popular open-source Java-based application server. An application server does not deliver web pages, but instead provides a common set of services that can be accessed and used by any type of Java client, including both Java servlets and JSPs.

Application servers are targeted towards large projects with many complex objects and are often used with databases quite heavily. Most application servers can automatically handle storing records in database tables, running multiple threads for performance and assist with complex tasks such as handling database transactions and connection pooling.

Finally, most application server also provide tools used to monitor the status of the J2EE components, along with tools for configuring security settings, clustering and more. JBoss is a full-featured and robust application server capable of handling almost any size of project. It includes Tomcat so you get a complete server system capable of using both Enterprise Java Beans and web pages that use Java servlets and JSPs.

WebSphere

WebSphere is a commercial application server from IBM. It supports both JSP/Servlets and J2EE JavaBeans components and runs on many different hardware/operating system combinations. While not free, a trial version can be downloaded and installed.

In general, WebSphere is considered to be more polished and easier to install than JBoss, as well as providing enhanced security management options.

WebLogic

BEA's WebLogic is another very robust and popular application server with very similar capabilities to both JBoss and WebSphere. Again, while this is a commercial product, a free trial version is available for download.

Other Application Servers

Apple's WebObjects
Borland's Enterprise Server
Macromedia's jRun Server
Novell's exteNd (Novell also recommends JBoss)
ObjectWeb's JOnAS (Free)
Oracle's Application Server
SAP AG Web Application Server
Sun Java System Application Server (Free)
Sybase EAServer (Developer's Version Free)
Tmax Soft's JEUS

Installing and Configuring an Application Server

JBoss Installation

NOTE: These instructions are for JBoss version 4.0.4. Other versions may be different. Consult http://www.jboss.com for more information.

Step 1: Download and install JDK 1.5 from http://java.sun.com

Step 2: Download the JBoss 4.0.4 installer JAR at http://www.jboss.com.

Step 3: Run the following command (Windows users may be able to double-click the file to launch it also):

java -jar jboss-4.0.4.GA-Patch1-installer.jar

Step 4: Install the application using the graphical installer.

> **NOTE:** It is recommended that you install JBoss to a folder without spaces in the name to avoid problems (especially for Unix/Linux systems).

JBossIDE Installation

Since we will be working with JBoss a lot during this course, we want to install the JBossIDE for Eclipse. This plugin automates many complex tasks for building both Enterprise JavaBeans as well as web pages using the JBoss server. I highly recommend downloading and install the JBossIDE bundle, which includes Eclipse and about 10 different plugins in addition to the JBossIDE plugin.

The JBossIDE provides the following features:

- Extensive support for XDoclet.
- Debugging and monitoring of JBoss servers.
- An easy way to configure the packaging layout of archives.
- A simple way to deploy an archive to a JBoss server.
- Several J2EE wizards to simplify J2EE development.
- Source code editors for JSP, HTML and XML.

Step 1: Download the JBossIDE bundle from http://www.jboss.org/products/jbosside.

Step 2: You can use any program with the ability to extract files from the archive such as WinZip, Power Archiver, or others. You may also extract the files using the JAR commands shown below:

cd “\Program Files”
jar xf {download-path}/JBossIDE-1.6.0.GA-Bundle-win32.zip

> **NOTE:** If you use the JAR command under Unix/Linux to extract the files, some files may not get the correct property settings, especially the executable status. Consult the online documentation for more information about this.

Finalizing the Configuration

Now that you have installed both an application server and the development environment, let's make sure everything is working correctly.

You may want to create a shortcut to the Eclipse.exe program first.

Next, run the Eclipse program. I highly recommend checking for updates at this time by visiting the Help -> Software Updates -> Find and Install menu options.

NOTE: You may wish to download and install additional plugins, such as the Visual Editor or C++ Development Plugin depending on which programming languages and/or types of applications you will be developing.

Installing the Visual Editor

While we will not be using the Visual Editor in this course, here is the procedure needed to properly install the plugin:

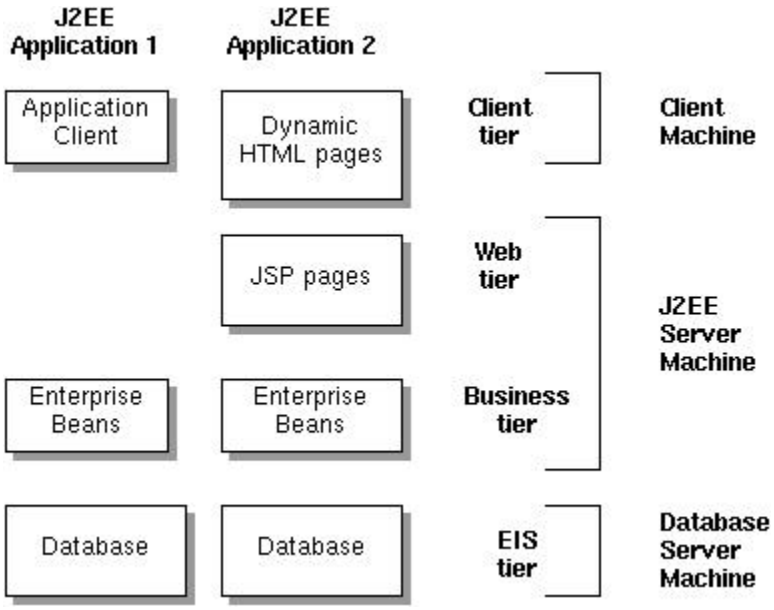
1. Install Eclipse SDK 3.1.2 SDK or the JBossIDE bundle
2. Unzip into a clean directory
3. Run the eclipse.exe out of the eclipse directory and select/create a workspace
4. Do Help->Software Updates->Find and Install...
5. Select the Search for New features to install... and hit the Next button
6. Click on New Remote Site... button, and add this site (name it VE):

http://update.eclipse.org/tools/ve/updates/1.0

7. Click on New Remote Site... button, and add this site (name it EMF): *http://update.eclipse.org/tools/emf/updates*
8. Click on New Remote Site... button, and add this site (name it Old Eclipse): *http://update.eclipse.org/updates/3.0*
9. Select VE, EMF, and Old Eclipse, and Hit Finish
10. Select the mirrors to use as they are asked for
11. Expand the tree VE->VE->Visual Editor SDK 1.1.0.1, and hit the checkbox on it
12. Expand EMF tree, EMF->EMF SDK 2.1.2->EMF SDK 2.1.2 and hit the checkbox on it
13. Expand Old Eclipse->GEF 3.1.1->Graphical Editing Framework 3.1.1 and hit the check box on it
14. Hit Next, accept the licenses, hit Next, hit Finish

J2EE Introduction

J2EE supports a multitiered, distributed application model. Generally this consists of at least four different application layers as shown in the figure below:



The client computer typically uses either a custom Java-base application or a web browser to access pages on a Java-enabled web server.

The web pages can consist of either static HTML pages or dynamically created pages. Dynamically generated pages consist of either JavaServer Pages (JSP) or Java Servlets. In general, JSP is preferred over Java Servlets, since they are easier to create. In many cases, the Java code in the web server will make use of Java objects in the business layer to actually carry out operations.

The business tier is where the logic of the particular business domain resides. Requests for the client and/or web tier are sent to the business tier, which often queries or saves data in a database server. In many cases, the web server software runs on the same computer as the business tier software, but this is not a requirement. This usually provides the best overall performance however.

Finally, the Enterprise Information System tier is where the data

actually resides and generally consist of database servers, enterprise resource planning (ERP), or other legacy systems such as mainframes.

As a general rule of thumb, thin-client multitiered applications are difficult to develop because they involve many lines of intricate code to handle transactions, state management, multithreading, resource pooling and other low-level details. Since J2EE systems are component based, business logic can be organized into reusable components and the J2EE server itself will provide many of the underlying services for the components. This usually means that developers can concentrate on the business problems and rely on the J2EE server to handle the other required services.

Containers and Services

J2EE components are installed into their containers during deployment. Before any web, enterprise bean or application client component can be executed, it must be assembled into a J2EE application and deployed into its container.

Assembly is the process of specifying container settings for the components as well as for the J2EE application itself. Container settings involve specifying settings for services including security, transaction management, Java Naming and Directory Interface (JNDI) lookups and remote connectivity.

- The J2EE security model lets you configure a web component or enterprise bean so resources can only be accessed by authorized users.

- The J2EE transaction model lets you specify relationships among methods that make up a single transaction, so all the methods in the transaction are treated as a single unit.

- JNDI lookup services provide a unified interface the multiple naming and directory services in the enterprise.

- The J2EE remote connectivity model manages low-level communications between clients and enterprise beans, normally using RMI. After the enterprise bean is created, clients invoke bean methods as if it were in the same virtual machine, although with a performance loss due to RMI and network overhead.

- J2EE containers also manage non-configurable services including

enterprise bean and servlet life cycles, database connection resource pooling, data persistence, and access to the J2EE platform APIs.

J2EE Platform APIs

Enterprise JavaBeans Technology – An enterprise bean is a body of code with fields and methods that implement business logic. There are 3 types of enterprise beans: session beans, entity beans, and message-driven beans. Entity beans are a heavyweight solution to access database records and often make use of object to relational mapping (ORM) technologies, so developers can avoid writing JDBC and SQL code. Session beans are workers that generally do not save information for long periods of time and message-driven beans react to messages when they are sent.

JDBC – This technology allows you to invoke SQL commands from with Java programs. In the case of container-managed persistence, the database operations are handled by the container automatically and your enterprise beans may not contain any JDBC code at all. Of course, you can still use JDBC within session beans, JSP pages or Java Servlets if needed. However, it is considered bad design to use JDBC inside JSP pages or Java Servlets since this increases coupling between what are supposed to be different layers of J2EE services.

Java Servlet Technology – This allows you to build HTTP-specific classes. They support a request-response programming model and are typically used to extend web servers.

JavaServer Pages (JSP) Technology – JSP pages allow you to mix both Java code and static HTML content in a text-based document.

Java Message Service (JMS) – This is a messaging standard that allows J2EE applications to create, send, receive and read messages. It enables loosely coupled distributed communications that are both reliable and asynchronous. While similar in some ways to e-mail, JMS is primarily targeted for component-to-component communications, not human-to-human communications.

Java Transaction API (JTA) – This standard provides a standard demarcation interface for declaring where a database transaction begins, rolls back and commits.

JavaMail Technology – This standard provides support for sending

and receiving e-mail messages to/from users. It consists of two parts: an application-level interface used by the application to actual send or receive e-mail and a service provider interface for communicating with the mail server.

JavaBeans Activation Framework (JAF) – This is used by the JavaMail technology and provides services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it and create the appropriate JavaBean component to perform those operations. It uses MIME data type specifiers to create the appropriate JavaBean component that can handle the data.

Java API for XML (JAXP) – This API supports creating and consuming XML resources. It is often used to create reports in a platform and programming language neutral way, which can later be translated into other formats such as spreadsheets, HTML pages, or printed documents such as PDF.

J2EE Connector API – This API allows system integrators and tool vendors to create adapters that allow J2EE applications to communicate with external enterprise information systems (EIS) such as databases, employee resource management (ERM) systems or other legacy applications.

Java Authentication and Authorization Service (JAAS) – Provides a way for J2EE applications to authenticate and authorize a specific user or group of users to run it. It is a Java-based version of the standard Pluggable Authentication Module (PAM) framework used on many Unix and/or Linux platforms to provide user-based security.

Container Types

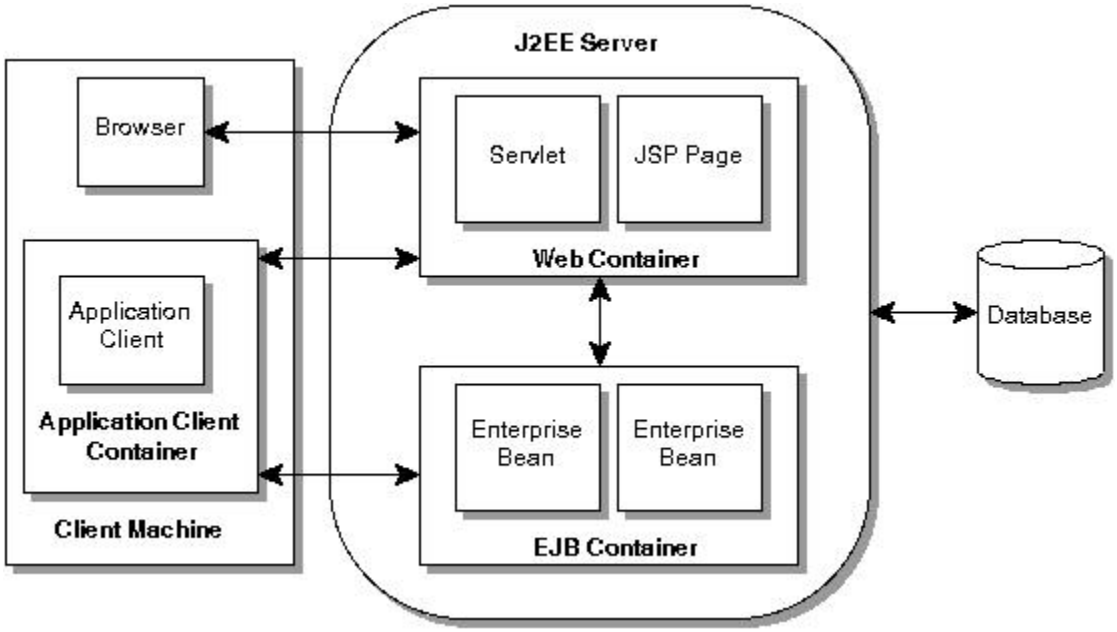
The figure below shows the various containers that are used when creating J2EE applications.

- The Enterprise JavaBeans (EJB) container manages the execution of all enterprise beans for the J2EE application. This runs on the J2EE server.

- A web container manages the execution of all JSP page and servlet components for one J2EE application. This also runs on the J2EE server.

- An application client container manages the execution of client components and runs on the client machine.

- An applet container is the web browser and Java VM plug-in combination running on the client machine.



Packaging

J2EE components must be packaged and bundled into a J2EE application for deployment. Normally each component, along with its related files such as GIF and HMTL files or server-side utility classes, and a deployment descriptor (DD) and are assembled into a module which is then added to the J2EE application.

The J2EE application and each module requires its own deployment descriptor. A deployment descriptor is an XML document that describes the component deployment settings, such as transaction attributes and security requirements. Since the deployment descriptors are declarative, they can be changed without needed to modify the bean source code.

A J2EE application including all of its modules are then delivered in an Enterprise Archive (EAR) file. The individual modules are first packaged into standard Java Archive (JAR) and Web Archive (WAR) files

and then added to the EAR file for final deployment in the J2EE application server.

- Each EJB JAR files contains its deployment descriptor, related files, and the .class files for the enterprise beans themselves.

- Each application client JAR file contains its deployment descriptor, related files, and the .class files for the application client.

- Each WAR file contains its deployment descriptor, related files, and the .class files for the servlets or .jsp files for a JSP page.

In many situations, you can reuse existing components in different J2EE applications by merely assembling the correct modules into the J2EE EAR file.

J2EE Architectures

Goals -

- *Be robust* – Enterprise software is often the lifeblood of the business, therefore the software must be bug-free and reliable.

- *Be performant and scalable* – Enterprise applications must meet the expectations of the users and must be able to handle ever increasing workloads, especially web applications, where the expected number of users is unpredictable. In some cases, scalability can be achieved using clustering technologies, however this is a complex solution that requires sophisticated designs.

- *Take advantage of OO design principles* – Object-oriented design principles have proven themselves valuable for complex systems. One of the most widely used techniques is the use of design patterns, which are recurring solutions to common problems. It is important to implement good OO designs, instead of allowing J2EE to dictate the design.

- *Avoid unnecessary complexity* – The Extreme Programming (XP) model advocates doing "the simplest thing that could possibly work". Often J2EE designs are overly complex, partly because J2EE offers so many different services and features. Developers need to avoid complexity whenever possible to avoid adding costs to the software life cycle.

- *Be maintainable and extensible* – Since maintenance is the most expensive phase of any software project, it is important to have a clean design with loosely coupled components. This means each component in the system should have a clearly defined responsibilities that do not overlap with the responsibilities assigned to other components.

- *Be delivered on time* – Productivity is important, but often neglected in J2EE designs.

- *Be easy to test* – Automated testing procedures are important to any design, especially when changes are made to components. It is vital that the changes can be tested to ensure they do not break other the system.

- *Promote reuse* – Enterprise systems are usually a long-term projects that may be used for many years. Reuse comes in two flavors, object/class reuse and application server infrastructure reuse.

Secondary Goals -

- Support for multiple client types – Often developers assume that J2EE applications allows need to support multiple J2EE-technology clients, such as web applications, stand-alone Java GUI clients using Swing other other windowing systems or Java applets. More and more often however, “thin” web clients are the standard, even for internal applications with an organization (ease of deployment is the primary reason for this).

- Portability – You should always ask yourself how import portability is to the project. While it is possible to write J2EE applications that allow you to change out database engines, or move from one application server to another, often the effort will both add cost and delay the time it takes to develop the application. Portability should only be important if it is a requirement of the business.

Distributed Architectures

Benefits **-**

- The ability to support many clients (possibly of different types) that required a shared “middle tier” of business objects. This does not apply to web applications, because the web container provides a middle tier.

- The ability to deploy any application component on any physical server. This sometimes can be used to split the work between different servers and is most useful when dealing with very CPU intensive calculations.

Disadvantages **-**

- Performance problems – Remote method invocations are many times slower than local methods.

- Complexity – Distributed applications are hard to develop, debug, deploy and maintain.

- Restricts using OO designs – Distributed applications often break

clean OO designs to reduce network activity.

In general, it is best to avoid distributed application designs unless absolutely required.

When to use EJB

Keep in mind that EJB is only one part of the overall J2EE technology, even though many books, web sites and developers think of it as the core of J2EE. When your design requires distributed components that will make use of RMI/IIOP, then EJB are a good solution with many benefits. Primarily the EJB container will manage many complex issues such as multiple threads, database transactions and object life cycle for the developer.

However, all these features do come at a price. Often time EJB is used for the wrong reasons. Use them when needed, but not until there is a clear benefit.

EJB Considerations

The official mantra regarding EJB states “The EJB architecture will make it easy to write applications: Application developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, and other complex low-level APIs.”

While this is at least partly correct, keep in mind that using EJBs often will add as much complexity as they solve as well as providing less performance than carefully designed code that does make use of the low-level APIs. In addition, developers who do not understand how the low-level code works, risk doing either very dangerous things, or writing very ineffective solutions.

EJB Usage Implications

- EJB makes applications harder to test – Distributed applications are always harder to test since they make heavily use of the container services.

- EJB applications are harder to deploy – EJBs have complex classloader issues (making sure JAR files contain the correct Java class files in the correct locations and that the container server knows how to find them); complex deployment descriptors (which

describe components to the container server); and slower development cycles (deploying EJB components is usually slower than deploying web applications.

- EJB with remote interfaces may hinder using good OO design principles – Because remote methods are much slower than local methods, developers will often bundle multiple operations into one method calls to reduce the number of round-trip network packets. This can make working with an EJB-based solutions harder to understand, maintain and develop.

- Using EJB may make simple things harder – Some simple concepts are difficult to achieve with EJB designs (such as the Singleton pattern). Remember EJB is a heavyweight technology, which makes heavy work of some simple problems.

- Reduced choice of application servers – There are more web containers available than EJB containers and web containers are usually easier to use than EJB containers. In addition, EJB containers often require more computer resources (CPU speed, memory, etc.) than the simpler web containers.

Enterprise JavaBeans
Session, Entity and Message-Driven Beans

Introduction

Enterprise JavaBeans are components (implemented much like RMI objects) that run inside of a server (similar to servlets) which is called a container. They provide features that would be impossible, or difficult, to implement on our own, including:

- EJBs have the ability to run on our server, or on another server, seamlessly. This means you can create multi-tiered systems that continue to work, even as EJBs are moved from one server to another.

- The EJB container (server) can manage the mapping of objects to database tables for you. Basically, you define the database tables and connect objects to the tables and let the container handle everything else, including the queries, inserts and updates. You can override this and manage the updates yourself, if needed.

- Just as databases support transactions to allow for safe updates to data, so do EJBs. This makes it possible for an update of several components to be performed in an all-or-nothing fashion.

Just like RMI objects, EJBs consist of a "real" object on the server and a reference on the client system. EJBs have a third role called the *container provider*. The container provider is responsible for providing a number of important services, including transaction processing, security, object persistence, and resource pooling. The container is strictly a server-side entity.

EJBs come in three flavors.

Entity beans are objects that map to a relational database. Each instance of a bean is generally associated with a single row of data in a table. The bean will have a number of variables, one for each column in the row. We need to setup a database table to match our bean, but the container will take of the SQL INSERT, UPDATE, DELETE and SELECT statements for us. Most application servers also have the ability to create the database table from the bean's information. While this is convenient for new projects, in many cases you will have to adapt the bean to an existing database schema.

Session beans are more like normal RMI objects. They perform actions, by themselves or by working with one or more entity beans. Session beans normally have no state of their own, which makes them more efficient than entity beans. However, there are times when you might want to keep some state information in a session bean. For this reason, EJB also offers the stateful session bean, whose state is retained between invocations.

Finally, message-driven beans react to events from other beans. They could be used in many different ways, such as sending an order to a remote shipping department server or triggering an automatic reorder of supplies when the on-hand quantity drops below a specific point or routing web based help requests to the correct technical point of contact.

J2EE Session Beans

An example session bean

We will create a simple CalculatorBean that can add, subtract, multiply and divide two numbers. The actual implementation is quite simple. Since EJB are designed to be located and used from remote clients, in a similar way to working with RMI objects, we must also build some methods to allow clients to find our bean. This involves writing two interfaces and one class.

The class itself will perform the work. It will be named CalculatorBean. One of the interfaces performs the exact same functionality that our interface for RMI objects did. It allows remote clients to connect to our bean and invoke its methods. It will be named Calculator. The other interface is used to find the bean, create new instances of the bean, or destroy copies of the bean. This is called the *home interface*. It will be named CalculatorHome. In addition, we will need a client, which will be the UseCalculator class.

Before we can make the bean available for use under JBoss, we will need two extra XML files that describe the bean to the container. The first is called the "deployment descriptor" and is a standard part of all EJBs. It is in the file named ejb-jar.xml. The second is named jboss.xml and is required so JBoss can make the bean available under its JNDI services.

All of the .class files and .xml files must be placed into a .jar file for use under JBoss. While we could build this by hand, it is probably better to automate these steps using a new "make" tool for Java called Ant. The rules needed to build the jar file are found in the file named build.xml.

File: **build.xml**

```
<?xml version="1.0" encoding="UTF-8" ?>

<project name="Calculator Build Script" default="ejb-jar" basedir=".">

   <!-- Import the environment -->
   <property environment="env" />

   <!-- Inherit the JBoss directory name from the environment -->
   <property name="jboss.dist" value="${env.JBOSS_DIST}"/>

   <!-- In which directory are the .java source files? -->
   <property name="src.dir" value="${basedir}"/>

   <!-- Where should we perform our build? -->
   <property name="build.calculator.dir"
         value="${basedir}/build/calculator"/>

   <!-- Where should javac put compiled Java .class files -->
   <property name="build.classes.dir"
         value="${build.calculator.dir}/classes"/>

   <!-- Location of jndi.properties, describes JNDI to the client -->
   <property name="src.resources" value="${basedir}/resources"/>

   <!-- Add the JBoss jarfiles to our CLASSPATH -->
   <path id="base.classpath">
     <pathelement location="${jboss.dist}/client/jboss-j2ee.jar"/>
     <pathelement location="${jboss.dist}/client/jaas.jar"/>
     <pathelement location="${jboss.dist}/client/jbosssx-client.jar"/>
     <pathelement location="${jboss.dist}/client/jboss-client.jar"/>
     <pathelement location="${jboss.dist}/client/jnp-client.jar"/>
   </path>

   <!--
========================================================
====== -->
   <!-- Verify that JBoss jarfiles are in our CLASSPATH -->
   <!--
========================================================
====== -->
   <target name="validate">
     <echo message="Validating your JBOSS_DIST environment variable"/>
     <available property="classpath_id"
           value="base.classpath"
```

```
            file="${jboss.dist}/client/jboss-j2ee.jar" />
   </target>

   <!--
=====================================================
====== -->
   <!-- Exit with a fatal error if we didn't find the jarfile -->
   <!--
=====================================================
====== -->
   <target name="fail_if_not_valid" unless="classpath_id">
      <fail message="jboss.dist=${jboss.dist} is not a valid JBoss dist directory"/>
   </target>

   <!--
=====================================================
====== -->
   <!-- Print debugging information and set things up -->
   <!--
=====================================================
====== -->
   <target name="init" depends="validate,fail_if_not_valid">

      <echo message="JBoss configuration seems OK!"/>

      <!-- Set the CLASSPATH -->
      <property name="classpath" refid="${classpath_id}" />

      <!-- Print current values for debugging -->
      <echo message="Using JBoss directory=${jboss.dist}" />
      <echo message="Using classpath=${classpath}" />
      <echo message="Using Source directory=${src.dir}" />
      <echo message="Using Build directory=${build.dir}" />
   </target>

   <!--
=====================================================
====== -->
   <!-- Compile all of our classes, clients and EJBs -->
   <!--
=====================================================
====== -->
   <target name="compile" depends="init">
      <echo message="Compiling all of the Java source code"/>

         <mkdir dir="${build.classes.dir}"/>
         <javac srcdir="${src.dir}" destdir="${build.classes.dir}"
            debug="on" deprecation="on" optimize="off">
            <classpath path="${classpath}" />
            <include name="com/bamafolks/rlp/calculator/*.java" />
         </javac>
   </target>
```

```
   <!--
=====================================================
====== -->
   <!-- Compile our classes, and create a jarfile -->
   <!--
=====================================================
====== -->
   <target name="ejb-jar" depends="compile">
      <echo message="Creating the jarfile"/>
        <delete dir="${build.calculator.dir}/META-INF"/>

        <mkdir dir="${build.calculator.dir}/META-INF"/>

        <copy file="${src.dir}/com/bamafolks/rlp/calculator/ejb-jar.xml"
             todir="${build.calculator.dir}/META-INF" />

      <copy file="${src.dir}/com/bamafolks/rlp/calculator/jboss.xml"
           todir="${build.calculator.dir}/META-INF" />

        <jar jarfile="${build.calculator.dir}/calculator.jar">
           <fileset dir="${build.classes.dir}">
                <include name="com/bamafolks/rlp/calculator/Calculator.class" />
                <include name="com/bamafolks/rlp/calculator/CalculatorHome.class" />
                <include name="com/bamafolks/rlp/calculator/CalculatorBean.class" />
           </fileset>

           <fileset dir="${build.calculator.dir}">
                <include name="META-INF/ejb-jar.xml" />
                <include name="META-INF/jboss.xml" />
           </fileset>
        </jar>
   </target>

   <!--
=====================================================
====== -->
   <!-- Deploy the jarfile with JBoss -->
   <!--
=====================================================
====== -->
   <target name="deploy" depends="ejb-jar">
      <copy file="${build.calculator.dir}/calculator.jar"
           todir="${jboss.dist}/deploy" />
   </target>

   <!--
=====================================================
====== -->
   <!-- Run the client from within EJB -->
   <!--
=====================================================
====== -->
   <target name="use-calculator-ejb" depends="deploy">
```

```
      <java classname="com/bamafolks/rlp.calculator.UseCalculator" fork="yes">
        <classpath>
          <pathelement path="${classpath}"/>
          <pathelement location="${build.classes.dir}"/>
          <pathelement location="${src.resources}"/>
        </classpath>
      </java>
   </target>

</project>
```

File: **ejb-jar.xml**

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar>
   <description>ATF Calculator Session Bean</description>
   <display-name>Calculator Session Bean</display-name>
   <enterprise-beans>
      <session>
         <ejb-name>Calculator</ejb-name>
         <home>com.bamafolks.rlp.calculator.CalculatorHome</home>
         <remote>com.bamafolks.rlp.calculator.Calculator</remote>
         <ejb-class>com.bamafolks.rlp.calculator.CalculatorBean</ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Bean</transaction-type>
      </session>
   </enterprise-beans>
</ejb-jar>
```

File: **jboss.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
  <enterprise-beans>
    <session>
          <ejb-name>Calculator</ejb-name>
          <jndi-name>calculator/Calculator</jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

File: **Calculator.java**

```
package com.bamafolks.rlp.calculator;

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

public interface Calculator extends EJBObject {
```

```
        public int multiply( int num1, int num2 ) throws RemoteException;
        public int divide( int num1, int num2 ) throws RemoteException;
        public int add( int num1, int num2 ) throws RemoteException;
        public int subtract( int num1, int num2 ) throws RemoteException;

}
```

File: **CalculatorHome.java**

```
package com.bamafolks.rlp.calculator;

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

public interface CalculatorHome extends EJBHome {

        Calculator create() throws RemoteException, CreateException;
}
```

File: **CalculatorBean.java**

```
package com.bamafolks.rlp.calculator;

import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class CalculatorBean implements SessionBean
{
        // This version of multiply() handles integers
        public int multiply(int num1, int num2)
        {
                System.out.println("Multiply invoked with ints '" +
                    num1 + "' and '" + num2 + "'.");
      return num1 * num2;
        }

        // This version of divide() handles integers
        public int divide(int num1, int num2)
        {
                System.out.println("Divide invoked with ints '" +
                                        num1 + "' and '" + num2 + "'.");
                return num1 / num2;
        }

        // This version of add() handles integers
        public int add(int num1, int num2)
        {
                System.out.println("Add invoked with ints '" +
```

```
                                        num1 + "' and '" + num2 + "'.");
                return num1 + num2;
        }

        // This version of subtract() handles integers
        public int subtract(int num1, int num2)
        {
                System.out.println("Subtract invoked with ints '" +
                                        num1 + "' and '" + num2 + "'.");
                return num1 - num2;
        }

        // ejbCreate -- we don't need this for our session bean
        public void ejbCreate() {}

        // ejbRemote -- we don't need this for our session bean
        public void ejbRemove() {}

        // ejbActivate -- we don't need this for our session bean
        public void ejbActivate() {}

        // ejbActivate -- we don't need this for our session bean
        public void ejbPassivate() {}

        // setSessionContext -- we don't need this for our session bean
        public void setSessionContext(SessionContext sc) {}
}
```

File: **jndi.properties**

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

The EJB Client

Before a client can start working with EJB, it must get a reference to an EJB factory, called a *home interface*. Once the client has the home interface, then it can be used to create new EJB instances, lookup existing EJB objects, or delete EJB objects. The EJB objects can be used to access data, perform tasks, and generally get things done.

In order to locate an EJB interface, you must perform a lookup using the JNDI interface. An EJB server publishes itself under a specific name under a JNDI namespace.

Example Client:

```
package com.bamafolks.rlp.calculator;
```

```
import java.util.Properties;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

import com.bamafolks.rlp.calculator.Calculator;
import com.bamafolks.rlp.calculator.CalculatorHome;

class UseCalculator
{
   public static void main(String[] args)
   {
         Properties prop = new Properties();
         prop.put
(Context.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces.NamingContextFactory");
         prop.put(Context.PROVIDER_URL,"localhost:1099");
         prop.put
(InitialContext.URL_PKG_PREFIXES,"=org.jboss.naming:org.jnp.interfaces");


      try
         {
            InitialContext jndiContext = new InitialContext(prop);
            System.out.println("Got context");

                  try {
                  // Get and display all children
                  NamingEnumeration ne = jndiContext.list("");
                  while( ne.hasMore() ) {
                    NameClassPair ncPair = (NameClassPair) ne.next();
                    System.out.print( "JNDI: " );
                    System.out.print( ncPair.getName() + " (type " );
                    System.out.println( ncPair.getClassName() + ")" );
                  }
                  } catch( Exception ex ) {
                           System.err.println("Error processing JNDI list: " + ex.getMessage()
);
                           ex.printStackTrace(System.err);
                  }

                  System.out.println("Attempting to lookup calculator bean in JNDI...");

            // Get a reference to the Calculator Bean
            Object ref = jndiContext.lookup("calculator/Calculator");
            System.out.println("Got reference");

            // Get a reference from this to the Bean's Home interface
            CalculatorHome home = (CalculatorHome)
               PortableRemoteObject.narrow(ref, CalculatorHome.class);

            // Create a Calculator object from the Home interface
            Calculator calculator = home.create();
```

```
            // call multiply()
            System.out.println("Multiplying 2 x 3:");
            System.out.println("Answer: " + calculator.multiply(2, 3));

            // call divide()
            System.out.println("Dividing 2 x 3:");
            System.out.println("Answer: " + calculator.divide(2, 3));

                  // call add()
            System.out.println("Adding 2 + 3:");
            System.out.println("Answer: " + calculator.add(2, 3));

            // call subtract()
            System.out.println("Subtracing 2 - 3:");
            System.out.println("Answer: " + calculator.subtract(2, 3));
         }
      catch(Exception e)
         {
            System.out.println( "Exception in UseCalculator" );
            System.out.println(e.toString());
            e.printStackTrace();
         }
   }
}
```

Sample Entity Bean Project

See the jTunes project.

Java Authentication and Authorization Services

The JAAS system is designed so developers can build their own authentication and authorization modules that can be plugged into Java's existing security system and activated using a couple of simple property files.

Using JAAS from client code involves working with 4 different types of objects. There are a LoginHandler (which gathers the user's login and password information), a LoginContext (which holds the user's information and is used to request login/logout), one or more Principal objects (which represents the user's account) and one or more Credential objects (which can be used to grant permissions).

Here are the general steps programs need to perform when using JAAS-based user authentication:

1 – Create a **LoginHandler** object.

2 – Create a new **LoginContext** object. You must pass the name of the desired JAAS configuration to use as well as a reference to the handler object into the constructor. The configuration details exactly which **LoginModule** will be used to validate the user's name and password.

3 – Invoke the *login()* method of the **LoginContext** object. It will use the **LoginHandler** to request the user's name and password. It will also load the specified JAAS configuration and attempt to validate the user's information. If the login fails, a **LoginException** will be thrown.

4 – Call the *getSubject()* method of the **LoginContext** object to retrieve the user's authentication information. This will return a reference to a **Subject** object that holds additional information about the user. Primarily you will be interested in the list of principals and credentials which are used to test for authentication (permissions).

5 – Use the *logout()* method of the **LoginContext** object to log the user out of the system.

Let's examine a very simple JAAS module that reads a list of user names and passwords from a Java properties file. This module does

not attempt to encrypt passwords, so it should not be used in a production environment.

Step 1 – JAAS Configuration File

First, you must create a JAAS configuration file that will activate our custom JAAS login module. This file can be named just about anything, but I recommend saving it as jaas.config.

```
Example {
        com.bamafolks.jaas.SimpleLoginModule
        required
        debug=”false”
        usernames=”joe:bill:sue”
        passwords=”password1:password2:password3”;
};
```

This associates the name “Example” with the SimpleLoginModule class from the com.bamafolks.jaas package. The “required” word specifies that in order for the login to be successful, the user is required to successfully authenticate against this module. The “debug”, “usernames” and “passwords” lines should be self-explanatory.

Different login modules will require options unique to that module. For example, a login module that uses a database will probably require at least a JDBC URL while a module that authenticates against a Windows domain will probably need the address of the domain controller and the name of the domain itself.

JAAS is flexible enough to support several different login modules within the same configuration. Each module can be configured as “optional”, “requisite”, “sufficient” or “required”. The meaning of each is shown below:

See also:

http://java.sun.com/j2se/1.4.2/docs/api/javax/security/auth/login/Configuration.html

Required

The LoginModule is required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list.

Requisite

The LoginModule is required to succeed. If it succeeds, authentication continues down the LoginModule list. If it fails, control immediately returns to the application (authentication does not proceed down the LoginModule list).

Sufficient

The LoginModule is not required to succeed. If it does succeed, control immediately returns to the application (authentication does not proceed down the LoginModule list). If it fails, authentication continues down the LoginModule list.

Optional

The LoginModule is not required to succeed. If it succeeds or fails, authentication still continues to proceed down the LoginModule list.

The overall authentication succeeds only if all Required and Requisite LoginModules succeed. If a Sufficient LoginModule is configured and succeeds, then only the Required and Requisite LoginModules prior to that Sufficient LoginModule need to have succeeded for the overall authentication to succeed. If no Required or Requisite LoginModules are configured for an application, then at least one Sufficient or Optional LoginModule must succeed.

This means you can combine different login modules in many ways to provide additional credentials or to validate user names and passwords against different security modules (Active Domain, LDAP, Unix, or perhaps databases).

> NOTE: You can also configure system-wide JAAS settings in the *java.security* policy file (located in the *{JRE}/lib/security* directory).

Step 2 – Create the SimpleLoginModule class

Since we are building our our login module, we now need to create a class that implements the LoginModule interface.

SimpleLoginModule.java

```
package com.bamafolks.jaas;
```

```
import java.util.Iterator;
import java.util.Map;
import java.util.Vector;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;

public class SimpleLoginModule implements LoginModule {

        // Initial state variables
        CallbackHandler callbackHandler;
        Subject subject;
        Map     sharedState;
        Map     options;

        // List of users and passwords
        String[] usernames = null;
        String[] passwords = null;

        // Temporary state variables
        Vector  tempCredentials;
        Vector  tempPrincipals;

        // Authentication status
        boolean success;

        // Configurable options
        boolean debug;

        /**
         * Creates a simple login module that authenticates using a
         * list of usernames and passwords from the JAAS configuration.
         *
         */
        public SimpleLoginModule() {
                tempCredentials = new Vector();
                tempPrincipals = new Vector();
                success = false;
                debug = false;
        }

        public void initialize(Subject subject,
                        CallbackHandler callbackHandler,
                        Map sharedState, Map options) {

                // Store the initial state
                this.callbackHandler = callbackHandler;
```

```
		this.subject        = subject;
		this.sharedState    = sharedState;
		this.options        = options;

		// Scan options
		if (options.containsKey("debug"))
			debug = "true".equalsIgnoreCase((String) options.get("debug"));

		if (options.containsKey("usernames"))
			usernames = ((String) options.get("usernames")).split(":");

		if (options.containsKey("passwords"))
			passwords = ((String) options.get("passwords")).split(":");

		if (debug) {
			println("\tSimpleLoginModule: intialize");
			println("\tSimpleLoginModule: usernames -");
			showList(usernames);
			println("\tSimpleLoginModule: passwords -");
			showList(passwords);
		}
	}

	/**
	 * Verify the password against the list
	 * of usernames and passwords.
	 */
	public boolean login() throws LoginException {

		if (debug)
			println("\tSimpleLoginModule: login");

		if (callbackHandler == null)
			throw new LoginException("Error: no CallbackHandler available "
+
					"to handle getting user login information.");

		if (usernames.length == 0)
			throw new LoginException("Error: no usernames are registered!");

		if (usernames.length != passwords.length)
			throw new LoginException("Error: mismatch between number of
usernames and passwords!");

		try {
			NameCallback nameCB = new NameCallback("Username: ");
			PasswordCallback passCB = new PasswordCallback("Password: ",
false);
			Callback[] handlers = new Callback[] { nameCB, passCB };

			callbackHandler.handle(handlers);

			String username = nameCB.getName();
```

```
                        String password = new String(passCB.getPassword());
                        passCB.clearPassword();

                        success = validate(username, password);

                        nameCB = null;
                        passCB = null;

                        if (!success)
                                throw new LoginException("Authentication failed.  Bad
username or password.");

                        return true;

                } catch (LoginException e) {
                        throw e;
                } catch (Exception e) {
                        success = false;
                        throw new LoginException(e.getLocalizedMessage());
                }
        }

        @SuppressWarnings("unchecked")
        public boolean commit() throws LoginException {
      if (debug)
         println("\tSimpleLoginModule: commit");

      if (success) {

         if (subject.isReadOnly()) {
            throw new LoginException ("Subject is read-only");
         }

         try {
            Iterator it = tempPrincipals.iterator();

            if (debug) {
               while (it.hasNext())
                  System.out.println("\t\t[SimpleLoginModule] Principal: " + it.next().
toString());
            }

            subject.getPrincipals().addAll(tempPrincipals);
            subject.getPublicCredentials().addAll(tempCredentials);

            tempPrincipals.clear();
            tempCredentials.clear();

            return true;

         } catch (Exception e) {
            e.printStackTrace(System.out);
            throw new LoginException(e.getMessage());
```

```
            }
        } else {
            tempPrincipals.clear();
            tempCredentials.clear();
            return(true);
        }
            }

            public boolean abort() throws LoginException {

                    if (debug)
                            println("\tSimpleLoginModule: abort");

                    // Clear the login state
                    success = false;

                    logout();

                    return true;
            }

            public boolean logout() throws LoginException {

                    if (debug)
                            println("\tSimpleLoginModule: logout");

                    tempPrincipals.clear();
                    tempCredentials.clear();

                    // Remove the principals w added to the subject
                    Iterator it = subject.getPrincipals(SimplePrincipal.class).iterator();
                    while (it.hasNext()) {
                            SimplePrincipal p = (SimplePrincipal) it.next();
                            if (debug)
                                    println("\tSimpleLoginModule: removing principle " +
p.toString());
                            subject.getPrincipals().remove(p);
                    }

                    // Also remove the credentials we added to the subject
                    it = subject.getPublicCredentials(SimpleCredential.class).iterator();
                    while (it.hasNext()) {
                            SimpleCredential c = (SimpleCredential) it.next();
                            if (debug)
                                    println("\tSimpleLoginModule: removing credential " +
c.toString());
                            subject.getPublicCredentials().remove(c);
                    }

                    return true;
            }

            private boolean validate(String username, String password) {
```

```
                SimplePrincipal p = null;
                SimpleCredential c = null;

                for (int i = 0; i < usernames.length; i++) {
                        if (usernames[i].equalsIgnoreCase(username) && passwords[i].
equals(password)) {

                                // Create some principals and credentials
                                p = new SimplePrincipal(username);
                                tempPrincipals.add(p);

                                c = new SimpleCredential();
                                tempCredentials.add(c);

                                return true;
                        }
                }
                return false;
        }
        private static void println(String text) {
                System.out.println(text);
        }

        private static void showList(String[] array) {
                for (String s : array) {
                        println("\t" + s);
                }
        }
}
```

Step 3 – The SimplePrincipal and SimpleCredential classes

The SimpleLoginModule above requires 2 additional classes named SimplePrincipal and SimpleCredential.

SimplePrincipal.java

```
package com.bamafolks.jaas;

import java.io.Serializable;
import java.security.Principal;

public class SimplePrincipal implements Principal, Serializable {

        private String name;

        public SimplePrincipal() {
                name = "";
        }

        public SimplePrincipal(String name) {
```

```
		this.name = name;
	}

	public boolean equals(Object o) {
		if (o == null)
			return false;

		if (this == o)
			return true;

		if (o instanceof SimplePrincipal) {
			return ((SimplePrincipal) o).getName().equals(name);
		}

		return false;
	}

	public int hashCode() {
		return name.hashCode();
	}

	public String toString() {
		return name;
	}

	public String getName() {
		return name;
	}
}
```

SimpleCredential.java

```
package com.bamafolks.jaas;

import java.util.Properties;

public class SimpleCredential extends Properties {

	public SimpleCredential() {
	}

}
```

Step 4 – The ConsoleLoginHandler class

Next, we need another helper class that somehow gets the user's login name and password. For this simple project, let's define one that uses the keyboard to gather this information.

ConsoleLoginHandler.java

```
package com.bamafolks.jaas;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;

public class ConsoleLoginHandler implements CallbackHandler {

	public ConsoleLoginHandler() {
	}

	public void handle(Callback[] callbacks) throws IOException,
			UnsupportedCallbackException {

		for (int i = 0; i < callbacks.length; i++) {

			if (callbacks[i] instanceof NameCallback)
				handle((NameCallback) callbacks[i]);
			else if (callbacks[i] instanceof PasswordCallback)
				handle((PasswordCallback) callbacks[i]);
			else
				throw new UnsupportedCallbackException(
						callbacks[i], "Callback class not supported");
		}
	}

	private void handle(NameCallback nameCB) throws IOException {
		print(nameCB.getPrompt());
		nameCB.setName(readLine());
	}

	private void handle(PasswordCallback passCB) throws IOException {
		print(passCB.getPrompt());
		passCB.setPassword(readLine().toCharArray());
	}

	private void print(String text) {
		System.out.print(text);
	}

	private static String readLine() throws IOException {
		return (new BufferedReader(new InputStreamReader(System.in))).
readLine();
	}
}
```

Step 5 – Client Program

Finally, we need a test program to try out our new JAAS login module.

LoginTest.java

```
package com.bamafolks.jaas;

import java.security.Principal;
import java.util.Properties;
import java.util.Set;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

//
// In order to test this code, you must create a file similar to this:
//
// Example {
//   SimpleLoginModule required debug="true" usernames="joe:bill:sue"
passwords="password1:password2:password3";
// }
//
// When you run the program, specify the file above like this:
//
// java -Djava.security.auth.login.config=jaas.config com.bamafolks.jaas.LoginTest

public class LoginTest {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		boolean loginSuccess = false;
		Subject subject = null;

		try {
			ConsoleLoginHandler cbh = new ConsoleLoginHandler();

			LoginContext lc = new LoginContext("Example", cbh);

			try {
				lc.login();
				loginSuccess = true;

				subject = lc.getSubject();

				System.out.println("Principal Names");
				System.out.println("---------------");
```

```
                                Set<Principal> principals = subject.getPrincipals();
                                if (principals.isEmpty()) {
                                        System.out.println("No principals defined!");
                                } else {
                                        for (Principal principal : principals)
                                                System.out.println(principal.getName());
                                }

//                              // NOTE: This also works...
//                              Iterator it = subject.getPrincipals().iterator();
//                              while (it.hasNext())
//                                      System.out.println(it.next().toString());

                                System.out.println("Credentials (Groups/Permissions)");
                                System.out.println("-------------------------------");

                                Set<Object> subjectCredentials =
                                         subject.getPublicCredentials();
                                for (Object object : subjectCredentials)
                                        System.out.println(object);

//                              // NOTE: Our SimpleLoginModule uses Properties object to
//                              //       store the list of credentials, so this works also.
//                              Set<Properties> credentials =
/                                       subject.getPublicCredentials(Properties.class);
//                              for (Properties properties : credentials)
//                                      properties.list(System.out);

                                System.out.println("Login succeeded.  Logging out now.");
                                lc.logout();
                                System.out.println("You are now logged out.");
                        } catch (LoginException lex) {
                                System.out.println("Login failed!");
                                System.out.println(lex.getClass().getName() + ": " +
                                        lex.getMessage());
                        }

                } catch (Exception ex) {
                        System.out.println(ex.getClass().getName() + ": " +
                                        ex.getMessage());
                        ex.printStackTrace();
                }
                System.exit(0);
        }
}
```

After compiling all the source code, you can run the program using a command line like this:

```
java -Djava.security.auth.login.config=jaas.config com.bamafolks.jaas.LoginTest
```

Default Login Modules

Sun provides several login modules that you can use out of the box for many authentication and authorization tasks. Consult the JDK documentation regarding the following classes (in the com.sun.security.auth packages):

Login Modules (*com.sun.security.auth.module* package):

JndiLoginModule
KeyStoreLoginModule
Krb5LoginModule
NTLoginModule
UnixLoginModule

Callback Handlers (*com.sun.security.auth.callback* package):

DialogCallbackHandler
TextCallbackHandler

Principal Classes (*com.sun.security.auth* package):

NTDomainPrincipal
NTSid
NTSidDomainPrincipal
NTSidGroupPrincipal
NTSidPrimaryGroupPrincipal
NTSidUserPrincipal
NTUserPrincipal
UnixNumericGroupPrincipal
UnixNumericUserPrincipal
UnixPrincipal

See also: *jaasExample2* (Unix Login) and *jaasExample3* (Windows Login)

Securing your J2EE Applications

You have several options when it comes time to secure your J2EE applications. Some of the options are required to be implemented by all application servers, however each server will probably provide its own security mechanisms also.

Standard Security Settings

You can apply security settings to the methods of an EJB (within the ejb-jar.xml), or to web pages (within the web.xml) for your application. Generally, most developers will apply security settings to the web pages, unless you are supporting remote interfaces for your beans.

First, you must define one or more security roles, which are similar to using groups in user management situations.

Here is an example of an ejb-jar.xml file with security roles:

```
<ejb-jar>
  <!-- Entity and session beans go here --!>
  <assembly-descriptor>
    <security-role>
      <description>
        A role that represents everyone.
      </description>
      <role-name>everyone</role-name>
    </security-role>
    <security-role>
      <description>
        A role that represents an administrator or manager.
      </description>
      <role-name>administrator</role-name>
    </security-role>
  </assembly-descriptor>
</ejb-jar>
```

Next, you must assign the roles to bean methods (also in ejb-jar.xml), like this:

```
  <method-permission>
    <role-name>administrator</role-name>
    <method>
      <ejb-name>ArtistBean</ejb-name>
      <method-name>create</method-name>
    </method>
  </method-permission>
  <method-permission>
```

```
    <role-name>everyone</role-name>
    <method>
      <ejb-name>ArtistBean</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>
```

Other options you can use are:

```
<unchecked/>
```

This grants access to call the method, not matter what role the user has.

```
<run-as>
  <role-name>role</role-name>
</run-as>
```

Use this with a bean to force the bean to impersonate the named *role* when calling other bean methods. Normally the originally caller's security role is propagated throughout all bean methods, but this can be used to make sure a bean can access other beans even if the original caller does not have sufficient permission.

```
<exclude-list>
  <method>
    <!-- bean and method names -->
  </method
</exclude-list>
```

This optional attribute makes the listed method cannot be called. You should make those method throw either a java.rmi.remoteException (remote interface) or javax.ejb.AccessLocalException (local interfaces) if the method is every called.

Since EJBs are often designed with only local interfaces and no remote interfaces, the only components that can call the bean methods are those running in the same container (JSP and servlets). In that case, it is more efficient to assign security settings to the various web pages instead of the beans themselves.

You can use a very similar set of settings as just described in the web.xml file to protect web pages. Here is an example:

```
<web-app>
 <security-role>
  <description>
```

```
    Web users that can login and access the site.
  </description>
  <role-name>web-user</role-name>
 </security-role>
<security-role>
  <description>
    Administrators can access all pages on the site.
  </description>
  <role-name>administrator</role-name>
 </security-role>
 <security-constraint>
  <display-name>User Security</display-name>
  <web-resource-collection>
    <web-resource-name>jTunes Application</web-resource-name>
    <description>Security settings for user pages</description>
    <url-pattern>/app/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
   <role-name>web-user</role-name>
  </auth-constraint>
 </security-constraint>
 <security-constraint>
  <display-name>Admin Security</display-name>
  <web-resource-collection>
    <web-resource-name>jTunes Admin Pages</web-resource-name>
    <description>Security settings for admin pages</description>
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>administrator</role-name>
  </auth-constraint>
 </security-constraint>
 <login-config>
  <auth-method>FORM</auth-method>
  <realm-name>jTunes Login</realm-name>
  <form-login-config>
   <form-login-page>/Login.jsp</form-login-page>
   <form-error-page>/LoginError.jsp</form-error-page>
  </form-login-config>
 </login-config>
</web-app>
```

Here I have defined 2 groups of users (or roles), the *web-user* role and the *administrator* role. Next, the *security-constraint* options along with the nested *web-resource-collection* and *auth-constraint* sections are used to define what role a user requires in order to access the URLs.

I have also used the *login-config* section to define which page is displayed when users need to login as well as the page to show if the login fails. The login form page should look something like this:

```
<html>
<head>
<title>Login Page</title>
</head>
<body>
<form method="POST" action="j_security_check">
  <div align="center">
   <table border="0">
    <tr><td align="right">Username:</td><td align="left"><input type="text"
name="j_username" /></td></tr>
    <tr><td align="right">Password:</td><td align="left"><input type="password"
name="j_password" /></td></tr>
    <tr><td colspan="2" align="center"><input type="submit" name="Login"
value="Login" /></td></tr>
   </table>
  </div>
</form>
</body>
</html>
```

Notice that the form will post itself to the *j_security_check* URL and contains two fields named *j_username* and *j_password*. The *j_security_check* URL is built into the application server and will attempt to validate the user's name and password combination.

Exactly how that happens varies, but most application servers use some form of JAAS login module to check for valid logins.

By default the JBoss server uses a very simple JAAS login module (called the **UsersRolesLoginModule**) that requires you to add two files to your .WAR file named *user.properties* and *roles.properties*. Here are examples of what those files look like:

users.properties

```
joe=top-secret
admin=pa55w0rd
```

roles.properties

```
joe=web-user
admin=web-user,administrator
```

As you can see, the user named *joe* has a password of *top-secret* and has the *web-user* role while the user named *admin* has a password of *pa55w0rd* and has both the *web-user* and *administrator* roles.

This is merely the default system used by JBoss. While it is convenient for initial testing, it is awkward to use in real projects, especially since you have to repackage and redeploy your application again to add or remove users.

The jboss-web.xml file can be changed to select different login modules, such as the much more flexible DatabaseServerLoginModule.

The DatabaseServerLoginModule

This is a JAAS login module that supports authentication and role mapping using a JDBC connection.

First you will need two tables in your database similar to this:

```
CREATE TABLE Principals(
        PrincipalID VARCHAR(64) PRIMARY KEY,
        Password VARCHAR(64)
)

INSERT INTO Principals VALUES ('test','top-secret')
INSERT INTO Principals VALUES ('randy','top-secret')

CREATE TABLE Roles (
        PrincipalID VARCHAR(64),
        Role VARCHAR(64),
        RoleGroup VARCHAR(64)
)

INSERT INTO Roles VALUES ('test','Echo','Roles')
INSERT INTO Roles VALUES ('test','caller_test','CallerPrincipal')
INSERT INTO Roles VALUES ('randy','Java','Roles')
INSERT INTO Roles VALUES ('randy','Coder','Roles')
INSERT INTO Roles VALUES ('randy','caller_randy','CallerPrincipal')
INSERT INTO Roles VALUES ('randy','Echo','Roles')
```

Next, you must customize the *login-config.xml* file (found under *{jboss}/server/default/conf*) and define a new section like this:

```
<application-policy name = “DatabaseUsers”>
 <authentication>
  <login-module code = “org.jboss.security.auth.spi.DatabaseServerLoginModule”
flag=”required”>
   <module-option name = “unauthenticatedIdentity”>
    guest
```

```
    </module-option>
    <module-option name=”dsJndiName”>
      java:/usersDS
    </module-option>
    <module-option name=”principalsQuery”>
      SELECT PASSWD FROM Principals WHERE PrincipalID = ?
    </module-option>
    <module-option name=”rolesQuery”>
      SELECT ROLEID, 'Roles' FROM  Roles WHERE  PrincipalID = ?
    </module-option>
   </login-module>
 </authentication>
</application-policy>
```

A sample Sun legacy format corresponding DatabaseServerLoginModule configuration would be:

```
testDB {
        org.jboss.security.auth.spi.DatabaseServerLoginModule required
        dsJndiName="java:/MyDatabaseDS"
        principalsQuery="SELECT PASSWD FROM JMS_USERS WHERE USERID=?"
        rolesQuery="SELECT ROLEID, 'Roles' FROM JMS_ROLES WHERE USERID=?"
};
```

The corresponding login-config.xml format entry is:

```
<application-policy name = "jbossmq">
  <authentication>
    <login-module code = "org.jboss.security.auth.spi.DatabaseServerLoginModule"
      flag = "required">
      <module-option name = "unauthenticatedIdentity">guest</module-option>
      <module-option name = "dsJndiName">java:/MyDatabaseDS</module-option>
      <module-option name = "principalsQuery">SELECT PASSWD FROM JMS_USERS
WHERE USERID=?</module-option>
      <module-option name = "rolesQuery">SELECT ROLEID, 'Roles' FROM JMS_ROLES
WHERE USERID=?</module-option>
    </login-module>
  </authentication>
</application-policy>
```

Finally, in your jboss-web.xml file, place this:

```
<jboss-web>
  <security-domain>java:/jaas/DatabaseUsers</security-domain>
</jboss-web>
```

jTunes Custom Users Table

As you have seen, the UsersRolesLoginModule, which JBoss uses by default, is not very flexible. There is no way for users to sign up for an account dynamically using this module. Let's see if we can change our jTunes project to use a database table of user accounts instead.

Step 1 – Create the tables

```
CREATE TABLE customer
(
  id INTEGER AUTO_INCREMENT PRIMARY KEY NOT NULL,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  email VARCHAR(100) NOT NULL UNIQUE,
  password VARCHAR(64) NOT NULL,
  expires DATETIME,
  secret VARCHAR(64),
  secret_expires DATETIME
);

CREATE TABLE roles
(
  id INTEGER AUTO_INCREMENT PRIMARY KEY NOT NULL,
  email VARACHAR(100) NOT NULL,
  roleid VARACHAR(64) NOT NULL
);
```

The first table will store the user's account information, while the second will be used to associate an email address with one or more security roles.

Step 2 – Configure JBoss

Next, edit the file named *login-config.xml* (in the *{jboss}/server/default/conf* directory). Add the following section to the file:

```
<policy>
  <!-- Leave the existing sections alone -->

  <application-policy name=”MusicUsers”>
    <authentication>
        <login-module code="org.jboss.security.ClientLoginModule" flag="required">
        </login-module>
        <login-module code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
          flag = "required">
          <module-option name="unauthenticatedIdentity">guest</module-option>
          <module-option name="dsJndiName">java:/musicDS</module-option>
          <module-option name="principalsQuery">
```

```
            SELECT password FROM customer WHERE email=?
          </module-option>
          <module-option name="rolesQuery">
            SELECT ROLEID, 'Roles' FROM roles WHERE email=?
          </module-option>
        <module-option name="hashCharset">UTF-8</module-option>
        <module-option name="hashEncoding">base64</module-option>
        <module-option name="hashAlgorithm">MD5</module-option>
        </login-module>
      </authentication>
    </application-policy>

</policy>
```

Here we have defined a new policy named *MusicUsers* that will connect to the database via the *java:/musicDS* datasource. Whenever JBoss needs to verify a password, it will run the following query:

```
SELECT password FROM customer WHERE email=?
```

JBoss will also run the following query (after the user's password is verified) to get a list of roles (permissions) the user has:

```
SELECT ROLEID, 'Roles' FROM roles WHERE email=?
```

The hard-coded 'Roles' in the query is required by JBoss.

In addition, we have stated that passwords are not stored in the database, but instead the database holds the results of running the UTF-8 password string through the MD5 encryption algorithm and then converting the result to a string using the Base64 encoder. This means that even if someone does gain direct access to the database, passwords will not be compromised.

List of options:

dsJndiName: The name of the DataSource of the database containing the Principals and Roles tables.

principalsQuery: The prepared statement query, equivalent to:

"SELECT Password FROM Principals WHERE PrincipalID=?"

rolesQuery: The prepared statement query, equivalent to:

"SELECT Role, RoleGroup FROM Roles WHERE PrincipalID=?"

NOTE: Value of RoleGroup column always has to be 'Roles' (with capital 'R'). This is specific to JBoss.

unauthenticatedIdentity=name, Defines the principal name that should be assigned to requests that contain no authentication information. This can be used to allow unprotected servlets to invoke methods on EJBs that do not require a specific role. Such a principal has no associated roles and so can only access either unsecured EJBs or EJB methods that are associated with the unchecked permission constraint.

password-stacking=useFirstPass, When password-stacking option is set, this module first looks for a shared username and password under the property names "javax.security.auth.login.name" and "javax.security.auth.login.password" respectively in the login module shared state Map. If found these are used as the principal name and password. If not found the principal name and password are set by this login module and stored under the property names "javax.security.auth.login.name" and "javax.security.auth.login.password" respectively.

hashAlgorithm=string: The name of the java.security.MessageDigest algorithm to use to hash the password. There is no default so this option must be specified to enable hashing. When hashAlgorithm is specified, the clear text password obtained from the CallbackHandler is hashed before it is passed to UsernamePasswordLoginModule?.validatePassword as the inputPassword argument. The expectedPassword as stored in the users.properties file must be comparably hashed.

hashEncoding=base64|hex: The string format for the hashed pass and must be either "base64" or "hex". Base64 is the default.

hashCharset=string: The encoding used to convert the clear text password to a byte array. The platform default encoding is the default.

ignorePasswordCase=true|false: (3.2.3+) A boolean flag indicating if the password comparison should ignore case. This can be useful for hashed password encoding where the case of the hashed password is not significant.

principalClass: (3.2.4+) An option that specifies a Principal implementation class. This must support a ctor taking a String

argument for the principal name.

suspendResume: (4.0.3+) A boolean flag that specifies that any existing JTA transaction be suspended during DB operations. The default is "true", i.e. query the database outside the thread's current transaction.

Next, we have to also modify the file named *auth.conf* (in the *{jboss}/client* directory). Add the following sections to that file:

```
MusicUsers {
  org.jboss.security.ClientLoginModule required;
  org.jboss.security.auth.spi.DatabaseServerLoginModule required;
};
```

The ClientLoginModule does not actually validate users. Instead it copies the user's information automatically whenever a bean method is called. This is called security propagation.

Step 3 – Add security settings to the XML deployment descriptors

Now we must inform JBoss about the new security policy and configure which roles are required by users to access our beans and web pages.

XDoclet does not directly support many of the security features used by JBoss, but it does allow you to create XML files that can be merged into the deployment descriptors when you run Xdoclet. If you examine the *merge* directory in the jTunes5 project, you will see I have created 3 merge files.

ejb-method-permissions.ent file:

```
<method-permission>
  <role-name>web-user</role-name>
  <method>
    <ejb-name>Artist</ejb-name>
    <method-name>*</method-name>
  </method>
  <method>
    <ejb-name>Album</ejb-name>
    <method-name>*</method-name>
  </method>
  <method>
    <ejb-name>Song</ejb-name>
    <method-name>*</method-name>
  </method>
```

```
  <method>
    <ejb-name>Customer</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
<method-permission>
  <role-name>administrator</role-name>
  <method>
    <ejb-name>Roles</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

That file holds the security settings related to our beans and will be merged into the *ejb-jar.xml* file by Xdoclet.

ejb-security-role.xml file:

```
<security-role>
  <description>Web users that can login and access the
site.</description>
  <role-name>web-user</role-name>
</security-role>
<security-role>
  <description>Administrators can access all pages on the
site.</description>
  <role-name>administrator</role-name>
</security-role>
```

This file merely declares the available security role names and is merged into the *ejb-jar.xml* file by XDoclet.

web-security.xml file:

```
<security-role>
  <description>Web users that can login and access the
site.</description>
  <role-name>web-user</role-name>
</security-role>
<security-role>
  <description>Administrators can access all pages on the
site.</description>
  <role-name>administrator</role-name>
</security-role>
<security-constraint>
  <display-name>User Security</display-name>
  <web-resource-collection>
    <web-resource-name>jTunes Application</web-resource-name>
    <description>Security settings for jTunes user pages</description>
    <url-pattern>/app/*</url-pattern>
  </web-resource-collection>
```

```
  <auth-constraint>
    <role-name>web-user</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <display-name>Admin Security</display-name>
  <web-resource-collection>
    <web-resource-name>jTunes Admin Pages</web-resource-name>
    <description>Security settings for jTunes admin pages</description>
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>administrator</role-name>
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>jTunes Login</realm-name>
  <form-login-config>
    <form-login-page>/Login.jsp</form-login-page>
    <form-error-page>/LoginError.jsp</form-error-page>
  </form-login-config>
</login-config>
```

This file controls the security settings for web pages. I have restricted all pages in the */app/** directory as requiring users to have the *web-user* role in order to access the page, while pages under the */admin/** area require the *administrator* role.

The *login-config* section tells JBoss to display the /Login.jsp file whenever it needs to authenticate a user. That files will request the user login with an e-mail address and password. JBoss then can load the JAAS configuration called *MusicUsers* to validate the login request.

Step 4 – Telling JBoss to use the *MusicUsers* domain

The final step is to let JBoss know it should use our new *MusicUsers* security domain instead of the default. This is easily accomplished by bringing up the Xdoclet properties and finding the *securityDomain* setting under the *jboss* section for the EJB. Also remember to do this under the *jbosswebxml* section for the Web.

Step 5 – Build a signup page

Now that all of the security settings are in place, it is time to create a new web page that users can use to create their own accounts. I have created two JSP files that allow this.

/docroot/Signup.jsp:

```
<%@page import="com.bamafolks.web.*"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>jTunes Signup</title>
</head>
<body>
<%= HtmlAssistant.getLogo() %>
<div align="center"><h2>Signup for jTunes</h2></div>
<div align="center"><h4>Complete the form to signup for a jTunes user
account</h4></div>
<div align="center">
 <form method="POST" action="/jtunes/CreateAccount.jsp">
  <table border="0">
  <tr><td align="right">*First Name:</td><td align="left"><input
type="text" name="first_name" /></td></tr>
  <tr><td align="right">*Last Name:</td><td align="left"><input
type="text" name="last_name" /></td></tr>
  <tr><td align="right">*E-mail Address:</td><td align="left"><input
type="text" name="email" /></td></tr>
  <tr><td align="right">*Password:</td><td align="left"><input
type="password" name="password1" /></td></tr>
  <tr><td align="right">*Confirm Password:</td><td align="left"><input
type="password" name="password2" /></td></tr>
  <tr><td colspan="2"> </td></tr>
  <tr><td align="center" colspan="2"><input type="submit" value="Signup"
></td></tr>
  </table>
 </form>
</div>
</body>
</html>
```

CreateAccount.jsp:

```
<%@page import="com.bamafolks.web.*"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>jTunes User Signup</title>
</head>
<body>
<%
      String firstName = request.getParameter("first_name");
      String lastName = request.getParameter("last_name");
```

```
      String email = request.getParameter("email");
      String password1 = request.getParameter("password1");
      String password2 = request.getParameter("password2");

      if (firstName == null ||
                  lastName == null ||
                  email == null ||
                  password1 == null ||
                  password2 == null ) {
%>
<h2>Error: Cannot register new account</h2>
<p><font color="red">You failed to provide one or more required
values.</font></p>
<p>Please go back and try again.</p>
<%
      } else {
            if (!password1.equals(password2)) {
%>
<h2>Error: Passwords do not match</h2>
<p><font color="red">You did not enter the same password twice.</font></
p>
<p>Please go back and try again.</p>
<%
            } else {
                  String[] tokens = email.split("@");
                  if (tokens.length != 2 || tokens[0].length() == 0 ||
tokens[1].length() == 0 ) {
%>
<h2>Error: Bad e-mail address</h2>
<p><font color="red">The e-mail address you entered does not appear to
be valid.</font></p>
<p>Please go back and try again.</p>
<%
                  } else {
                        if( HtmlAssistant.createNewUser(firstName,
lastName, email, password1) ) {
%>
<h2>Congratulations!</h2>
<p>Your new user account is now ready to use.  Please use your e-mail
address and password to login.</p>
<p><a href="/jtunes/app/index.jsp">Visit the Main Page</a></p>
<%
                        } else {
%>
<h2>Error: Failed to create account</h2>
<p>The system was unable to sign you up at this time.  Please try again
later.</p>
<%
                        }
                  }
            }
      }
%>
```

```
</body>
</html>
```

HtmlAssistant.createNewUser() method:

```
public static boolean createNewUser(
        String firstName,
        String lastName,
        String email,
        String password) {

        boolean success = false;
        Context context = null;
        Connection conn = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;

        try {
                context = new InitialContext();
                Object o = context.lookup("java:/musicDS");
                DataSource ds = (DataSource) o;

                conn = ds.getConnection();
                stmt = conn.prepareStatement("INSERT INTO customer (first_name,
last_name, email, password, expires) VALUES (?,?,?,?,?)");

                // Encrypt password into MD5 hash using Base64

                // NOTE: This required adding the commons-codec
                //       library from the Apache Jakarata project

                MessageDigest md5 = MessageDigest.getInstance("MD5");
                byte[] digest = md5.digest(password.getBytes());
                String encoded = Base64.encode(digest).trim();

                // Calculate an expiration of 1 year from now
                java.util.Date now = new java.util.Date();
                now.setYear(now.getYear() + 1);
                Date expires = new Date(now.getTime());

                stmt.setString(1,firstName);
                stmt.setString(2,lastName);
                stmt.setString(3,email);
                stmt.setString(4,encoded);
                stmt.setDate(5,expires);

                if( stmt.executeUpdate() == 1 ) {
                        stmt.close();

                        // Add new user to 'web-user' role also
                        stmt = conn.prepareStatement("INSERT INTO roles (email,roleid)
VALUES (?,?)");
```

```
                        stmt.setString(1,email);
                        stmt.setString(2,"web-user");
                        if (stmt.executeUpdate() == 1)
                                success = true;
                }

        } catch (NamingException e) {
                e.printStackTrace();
        } catch (SQLException e) {
                e.printStackTrace();
        } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
        } finally {
                if (rs != null)
                        try { rs.close(); } catch (SQLException e) { e.printStackTrace(); }
                if (stmt != null)
                        try { stmt.close(); } catch (SQLException e) { e.printStackTrace();
}
                if (conn != null)
                        try { conn.close(); } catch (SQLException e) { e.printStackTrace();
}
        }

        return success;
}
```

Finally, I modified the /docroot/Login.jsp as shown below:

```
<%@page import="com.bamafolks.web.*"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>jTunes Login Page</title>
</head>
<body>
<%= HtmlAssistant.getLogo()%>
<form method="POST" action="j_security_check">
  <div align="center">
   <table border="0">
    <tr><th align="center" colspan="2">You must login to continue</th></
tr>
    <tr><td colspan="2"><hr></td></tr>
    <tr><td align="right">E-mail Address:</td><td align="left"><input
type="text" name="j_username" /></td></tr>
    <tr><td align="right">Password:</td><td align="left"><input
type="password" name="j_password" /></td></tr>
    <tr><td colspan="2"> </td></tr>
    <tr><td colspan="2" align="center"><input type="submit"
name="Login" value="Login" /></td></tr>
```

```
   </table>
  </div>
</form>
<div align="center"><a href="/jtunes/Signup.jsp">Create New Account</a>
| <a href="/jtunes/LostPassword.jsp">Forgot Your Password</a></div>
</body>
</html>
```

There are still some things left to do, including adding a LostPassword.jsp page and an Administrator page.

EJB Version 3.0

As you can see, EJB is not the easist environment in the world to program. There are many places where mistakes can be made and a lot of XML that needs to be written to describe the beans, web pages and security settings. Many developers get frustrated when they first learn about EJB, after the initial excitement wears off. Even with the help of automated tools like XDoclet the job is tough.

Luckily, Sun is aware of the complexities and has decided to try and improve the situation. The latest EJB specifications (version 3.0) are primarily designed to make your job as a developer much easier. It does require an EJB3 enabled server and also Java 5.

Let's build a simple EJB3 bean.

Step 1 – Install JBoss in EJB3 mode.

The default installation of JBoss does not support EJB3. Although EJB3 support can be added on, I highly recommend just installing a second copy of JBoss so you have 2 different servers.

> **NOTE:** You must use the JBoss installer JAR file in order to activate an EJB3 server. During the installation process, be sure you ask for an EJB3 container.

Step 2 – Start a new EJB project.

Using Eclipse, create a new EJB3 Project named EasyCalcEJB3. It is useful if you already have your EJB3 version of JBoss configured and running before you create the project.

Step 3 – Define your interface

Next, create a new interface named EasyCalc in the com.bamafolks.ejb3 package name. Edit the file to look like this:

```
package com.bamafolks.ejb3;

public interface EasyCalc {

        double add(double a, double b);
        double subtract(double a, double b);
        double multiply(double a, double b);
        double divide(double a, double b);
```

```
}
```

Step 4 – Create the remote and local interfaces like this:

File: *EasyCalcRemote.java*

```
package com.bamafolks.ejb3;

import javax.ejb.Remote;

@Remote
public interface EasyCalcRemote {

}
```

File: *EasyCalcLocal.java*

```
package com.bamafolks.ejb3;

import javax.ejb.Local;

@Local
public interface EasyCalcLocal extends EasyCalc {

}
```

Step 5 – Create the bean class.

File: EasyCalcBean.java

```
package com.bamafolks.ejb3;

import javax.ejb.EJBException;
import javax.ejb.Stateless;

@Stateless
public class EasyCalcBean implements EasyCalcLocal, EasyCalcRemote {

	public double add(double a, double b) {
		return a + b;
	}

	public double subtract(double a, double b) {
		return a - b;
	}

	public double multiply(double a, double b) {
		return a * b;
	}
```

```
	public double divide(double a, double b) {
		if (b == 0.0) {
			throw new EJBException("divide by zero");
		}
		return a / b;
	}

}
```

Step 6 – Package the EJB.

Use the Packaging Configurations wizard to create a standard EJB JAR file named EasyCalcEJB.jar. Once the configuration is created, you may remove everything except the */EasyCalcEJB3/bin/* folder. Once you have done this, the packaging to create your JAR file.

Step 7 – Deploy the EJB.

You can right-click on the newly generated EasyCalcEJB.jar file and deploy it to the EJB3 enabled version of JBoss. JBoss will use reflection and some built in default to register your bean, without all the need for the complex deployment descriptors.

Step 8 – Create a client app

Add a new class called *EasyCalcEJB3Client* to the *com.bamafolks.ejb3.client* package as shown below:

```
package com.bamafolks.ejb3.client;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import com.bamafolks.ejb3.EasyCalc;

public class EasyCalcEJB3Client {

	/**
	 * @param args
	 * @throws NamingException
	 */
	public static void main(String[] args) throws NamingException {
		InitialContext ctx = new InitialContext();
		EasyCalc calculator = (EasyCalc) ctx.lookup("EasyCalcBean/remote");

		System.out.println("1 + 1 = " + calculator.add(1,1));
		System.out.println("2 - 1 = " + calculator.subtract(2,1));
```

```
        }

}
```

The bean is automatically registered under the names '*EasyCalcBean/remote*' and '*EasyCalcBean/local*' within JNDI by the EJB3 deployment package.

That was pretty easy wasn't it?