Using the JBoss IDE for Eclipse

Important:

Some combinations of JBoss/JBoss-IDE/Eclipse do not like to work with each other. Be very careful about making sure all the software versions are compatible. This tutorial has been tested with the following combinations of software:

Linux

Configuration #1
JDK 1.4.2
Eclipse 3.0.2 (GTK)
JBoss 3.2.7
JBossIDE 1.4.1.e31-jre14

Configuration #2
JDK 1.5.0
Eclipse 3.1.2
JBoss 4.0.4
JBossIDE 1.6.0

Windows

Configuration #1
JDK 1.5.0_03
JBoss 3.2.7
JBossIDE-1.5M1-jre15 (http://jboss.sourceforge.net)

Configuration #2
JDK 1.5.0_06
JBoss 4.0.4
JBossIDE 1.6.0 GA

Building a Simple EJB

Step 1: Create an EJB project

Select *File -> New Project* from the Eclipse menu

Browse to *JBoss-IDE > J2EE 1.4 Project* and click **Next**

Set the project name to ***SimpleCalc*** and click **Next**

Now, click the **Browse** button (near the bottom) and create a new output folder name ***bin***

Next click the **Add Folder** button (near the upper-right) and creating a folder named ***src***

> NOTE: Creating these folders makes Eclipse store your *.java* source code files in the ***src*** folder while the compiled *.class* files will be stored in the **bin** folder. Separate folders makes packaging and managing your projects easier.

Click on **Finish** to create the project.

Step 2: Create the Bean

Select *File -> New -> Other* from the Eclipse menu

Browse to *JBoss-IDE -> EJB Components,* select *Session Bean* and click **Next**

Enter ***CalculatorBean*** in the *Name* field and ***edu.uah.coned.ejb*** in the *Package* field.

Also enable the *ejbCreate()* check box and click **Finish**.

> NOTE: It is highly recommended that you place all your beans inside a package that ends with **'*.ejb*'** and to name the beans so they end with ***Bean***.

Step 3: Define the Bean

Use the Package Explorer window to expand the *CalculatorBean.java* file found in the *src/edu/uah/coned/ejb* folder. Right-click on the *CalculatorBean* class (the green C icon).

Select *J2EE > Add Business Method* from the menu.

Enter ***add*** for the *Method Name*, ***double*** for the *Return Type* and then use the Add button next to the Parameters window to create 2 parameters named ***a*** and ***b*** both of type *double*. Click the **Finish** button once the method is defined.

Repeat the steps above to 3 more methods named ***subtract***, ***multiply***, and ***divide***. All of them should return a *double* and accept two parameters named **a** and **b** of type *double*.

Step 4: Implement the new methods

Open the *CalculatorBean.java* file (if not already open) and modify the newly created methods as shown below:

```
/**
 * Business method
 * @ejb.interface-method  view-type = "remote"
 */
public double add(double a, double b) {
   return a + b;
}
/**
 * Business method
 * @ejb.interface-method  view-type = "remote"
 */
public double subtract(double a, double b) {
   return a - b;
}
/**
 * Business method
 * @ejb.interface-method  view-type = "remote"
 */
public double multiply(double a, double b) {
   return a * b;
}
/**
 * Business method
 * @ejb.interface-method  view-type = "remote"
 */
public double divide(double a, double b) {
   if (b == 0.0) {
      throw new EJBException("Divide by zero error");
   }
   return a / b;
```

```
    }
```

Step 5: Generate EJB related files

Now that our bean has been created, we need to build interfaces that clients will use to access the code. Enterprise JavaBean components are only used by the application server, never directly by clients. We are going to use the XDoclet tool which is bundled with the JBossIDE to create the various interface files for us.

First, select *Project > Properties* from the menu (or right-click on the ***SimpleCalc*** project and select *Properties)*

Select *XDoclet Configurations* from the list and click the *Enable XDoclet* check box if not already enabled.

Click the **Add Standard** button. Enter **EJB** in the *Name* file, select *Standard EJB* and then click the **OK** button.

Now, click on the new *EJB* configuration. Expand the bottom left tree near and find the *fileset* entry. Double the *includes* property (in the right list) and change it to ****/*Bean.java.** This restricts XDoclet so it only processes files that are named *Bean.java.

After saving your XDoclet Configuration settings, right-click on the ***SimpleCalc*** project and select the *Run XDoclet* option (you can also press Ctrl+Shift+F1).

Details

Once the XDoclet tool runs, you will find that 2 new folders have been added to your project. They are the *src/edu.uah.coned.interfaces* and *src/META-INF* folders.

The *interfaces* folder should have 3 files that define the interfaces for your Bean class. Clients will use these interfaces to access your Bean. If you add more methods to your Bean class, remember to *Run XDoclet* again to update the interface files.

In the *META-INF* folder you will find 2 files named *ejb-jar.xml* and *jboss.xml*. The first is Sun's standard EJB descriptor file that is required for all Bean classes. The second contains Java Naming and Directory Interface (JNDI) information JBoss needs about your Bean class, so it can be found by clients.

Step 6: Develop a client to use the Bean

At this point we need a Java program that exercises the new Bean we just created. Let's create a Java servlet and HTML page. Alternately, we could also build a Java command-line or GUI application instead.

Select *File -> New -> Other*

Now select *JBoss-IDE -> Web Components -> HTTP Servlet* and click **Next**

Enter ***CalculatorServlet*** for the *Name* and ***edu.uah.coned.web*** for the *Package*.

Also check the **init() method** and **doPost() method** check boxes and click the **Finish** button.

> NOTE: Just as EJB should be named so then end with **Bean**, it is highly recommended that you append **Servlet** to Java servlet classes. It is also a good idea to place servlets in their own package name that ends with **.web**.

Step 7: Implement the Servlet

First add a private variable to the class that will hold a reference to the Bean object.

```
public class CalculatorServlet extends HttpServlet {

    private CalculatorHome home;

    public CalculatorServlet() {
        super();
        // TODO Auto-generated constructor stub
    }
    ...
```

> NOTE: You will need to add an import statement for the **edu.uah.coned.interfaces.CalculatorHome** class to the project. The easist way to do this is by using Eclipse's *Organize Imports* command found under the *Source* menu, or by pressing *Ctrl+Shift+O*.

Next, modify the **init()** method to read as shown below:

```
public void init(ServletConfig config) throws ServletException {
    super.init(config);
    try {
        Context context = new InitialContext();
        Object ref = context.lookup("java:/comp/env/ejb/Calculator");
        home = (CalculatorHome) PortableRemoteObject.narrow(ref,
                                        CalculatorHome.class);
    } catch (Exception e) {
        throw new ServletException("Failed to lookup Calculator in JNDI");
    }
}
```

NOTE: Remember to import the **javax.naming.Context**, **javax.naming.InitialContext** and **javax.rmi.PortableRemoteObject** classes.

Next, implement the **doPost()** method like this:

```
protected void doPost(
   HttpServletRequest request,
   HttpServletResponse response) throws ServletException, IOException {

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   out.println("<html><head><title>");
   out.println("Calculator Results");
   out.println("</title></head>");
   out.println("<body>");

   out.println("<h1>Calculator Results</h1>");

   double a = 0, b = 0, result = 0;
   String operation = "";

   try {
      Calculator bean = home.create();
      String aStr = request.getParameter("a");
      String bStr = request.getParameter("b");

      if (aStr != null && bStr != null) {
         try {
            a = Double.parseDouble(aStr);
            b = Double.parseDouble(bStr);
         } catch (Exception e) {
         }

         if (request.getParameter("Add") != null) {
            operation = " + ";
            result = bean.add(a,b);
         } else if (request.getParameter("Subtract") != null) {
            operation = " - ";
            result = bean.subtract(a,b);
         } else if (request.getParameter("Multiply") != null) {
            operation = " * ";
            result = bean.multiply(a,b);
         } else if (request.getParameter("Divide") != null) {
            operation = " / ";
            result = bean.divide(a,b);
         } else {
            throw new ServletException("Unrecognized operation");
         }

      } else {
         throw new ServletException("Missing one or more input values");
      }

      bean.remove();

      out.println("<p>" + a + operation + b + " = " + result + "</p>");

   } catch (Exception e) {
      out.println(e.getMessage());
      e.printStackTrace(out);
   } finally {
      out.println("</body></html>");
      out.close();
   }
}
```

NOTE: Remember to import the **java.io.PrintWriter** class.

The final step for the servlet is to add some metadata to the top of the class. XDoclet uses this information to create the required JBoss configuration files for the servlet. This metadata must be created inside the comments at the top of the file. Modify it to read like this:

```
/**
 * Servlet Class
 *
 * @web.servlet          name="Calculator"
 *                       display-name="CalculatorServlet"
 *                       description="Exercises the Calculator EJB"
 *
 * @web.servlet-mapping  url-pattern="/Calculator"
 *
 * @web.ejb-ref          name = "ejb/Calculator"
 *                       type = "Session"
 *                       home = "edu.uah.coned.interfaces.CalculatorHome"
 *                       remote = "edu.uah.coned.interfaces.Calculator"
 *                        description = "CalculatorBean references"
 *
 * @jboss.ejb-ref-jndi   ref-name = "ejb/Calculator"
 *                        jndi-name = "ejb/Calculator"
 */
```

Step 8: Setup and run XDoclet for the Servlet

Previously we setup an XDoclet configuration to automatically generate the *ejb-jar.xml* and *jboss.xml* files for our Bean class. We must do the same thing for the servlet class.

First select *Project > Properties* (or right-click on the ***SimpleCalc*** project and select *Properties* from the menu).

Once again, highlight the *XDoclet Configurations* in the list

Click the **Add Standard** button. Enter **Web** in the *Name* file, select *Standard Web* and then click the **OK** button.

Next, select the *Web* configuration and click on *webdoclet* in the lower-left panel.   Change the *destDir* property property on the right to **src/WEB-INF**. This will force XDoclet to store the servlet related files in that folder within the web project.

Next, select the *fileset* entry on the left list and change the *includes* property to read ****/*Servlet.java**. Again, this makes Xdoclet only process source files that end with that extension.

Finally, clear the check box next to the *jsptaglib* entry on the left list. This is not required for our project.

After saving your changes, right-click on the ***SimpleCalc*** project and select the *Run XDoclet* option.

> NOTE: This should create a new *src/WEB-INF* folder with XML files that describe the servlet to Jboss. XDoclet has extracted the metadata we added to the top of our file and used that to generate a *jboss-web.xml* and *web.xml* file.

Step 9: Create an HTML page

Now we need an HTML page that will allow the user to enter the parameters desired and then invoke the Servlet we just created.

First, add a new folder to the project named **'docroot'**.

Next, create an HTML file named **index.html** and add it to the **docroot** folder. I find it easiest to right-click on the **docroot** folder and then select *New > Other* in the popup menu. Browse until you find the option to create an HTML file.

> NOTE: This option seems to be in different places depending on the version of Eclipse and/or JBossIDE. Try looking in the Web branch, or perhaps the JBoss-IDE branch.

Modify the file like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>Calculator EJB Test Page</title>
</head>
<body bgcolor="#FFFFFF">
 <h1>Calculator Form</h1>
 <form action="Calculator" method="post">
  <table cellpadding="2" cellspacing="2" border="0">
   <tr>
    <td align="right">First Number :</td>
    <td align="left"><input type="text" name="a"></input></td>
   </tr>
   <tr>
    <td align="right">Second Number :</td>
    <td align="left"><input type="text" name="b"></input></td>
   </tr>
   <tr>
    <td colspan="2">
     <input type="submit" name="Add" value="Add"></input>&nbsp;
     <input type="submit" name="Subtract" value="Subtract"></input>&nbsp;
     <input type="submit" name="Multiply" value="Multiply"></input>&nbsp;
     <input type="submit" name="Divide" value="Divide"></input>
    </td>
   </tr>
  </table>
 </form>
</body>
</html>
```

Step 10: Create a J2EE application file

J2EE web applications require a file named application.xml that contains a description and various options used by Java-enabled web server. Create the file by doing this:

First right-click on the ***src/META-INF*** folder and select *New -> Other*

Browse to *JBoss-IDE -> Descriptors > EAR 1.3 Deployment Descriptor* under the branch and click **Next**, then click **Finish**

Double-click the new **application.xml** file and modify it like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
    "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
    "http://java.sun.com/dtd/application_1_3.dtd">
<application>
    <display-name>Calculator Application</display-name>
    <module>
        <ejb>CalculatorEJB.jar</ejb>
    </module>
    <module>
        <web>
            <web-uri>CalculatorWeb.war</web-uri>
            <context-root>/Calculator</context-root>
        </web>
    </module>
</application>
```

Step 11: Packaging and Deploying the Application

Now that our code is completed, we must install the application so we can test it out. This involves creating several different types of JAR files and then deploying them to the JBoss server. Again, the JBoss-IDE for Eclipse has tools to help automate much of this.

Here are the files we need to create:

EJB Jar – This jar file will contain our Bean interfaces, classes and the EJB deployment descriptor files (under *src/META-INF*) we created earlier using the XDoclet utility.

EJB Client Jar – This jar file will contain only the EJB interfaces (withou the Bean) and is needed by the servlet class.

Web Application War – This file (with a .WAR extension) will contain the web site related files including the HTML documents, servlet class, the EJB Client and various deployment descriptors.

J2EE Application EAR – The Enterprise Application Archive (with a .EAR extension) will contain the EJB Jar, WAR and the web deployment descriptors (under *src/WEB-INF*). This has everything JBoss needs to run our application.

Start by bringing up the *Properties* window for the project again.

Highlight the '*Packing Configurations*' option on the list and click the check box near the top named *Enable Packaging*.

Now click the **Add Standard** button and enter ***CalculatorEJB.jar*** in the Name field and select **Standard-EJB.jar** from the list. Click the **OK** button to create a new configuration with default options.

We need to customize a couple of settings, so must remove a couple of options from the defaults, so expand the CalculatorEJB.jar tree so you can see what files are currently scheduled to be added to the JAR. Remove the **'/SimpleCalc/src/META-INF/MANIFEST.MF'** and **'/SimpleCalc/src/META-INF/jbosscmp-jdbc.xml'** entries.

> NOTE: We are not using JDBC for this example, so the j*bosscmp-jdbc.xml* file is not needed. The *MANIFEST.MF* file is normally used to set the correct CLASSPATH entries and other related properties needed by the application, but since the *jboss.xml* file already contains that information, we can safely remove it from this configuration. Application servers other than JBoss may have slightly different requirements.

Once again click on the **Add Standard** button. Enter ***CalculatorEJB-client.jar*** in the *Name* field and select *Standard-JAR.jar* from the list. Press **OK** to create a default configuration.

Again, the default settings need to be adjusted, so expand the tree and remove the MANIFEST.MF entry. Next, highlight the **'*/SimpleCalc/bin*'** entry and click on **Edit**.

Change the *Includes* to read ***edu/coned/uah/interfaces/*.class.*** This settings means the JAR file will only have the interface classes that are required by client programs.

Next, click on **Add Standard** again. Enter ***CalculatorWeb.war*** in the *Name* field and and select *Standard-WAR.war* from the list. Press **OK**.

Expand the tree and again remove the MANIFEST.MF file from the list.

Also edit the ***'/SimpleCalc/bin'*** entry and change the *Includes* value to ***edu/coned/uah/web/*.class.*** That way the WAR file will only contain classes related to our servlet.

Since the servlet uses our bean class, we must add the bean interface classes to the WAR package. If you remember, we earlier setup our packages so the interface classes will be packaged into the **CalculatorEJB-client.jar** file. So that means we need to add that file to the WAR file also.

To do this, right-click on the ***'CalculatorWeb.war'*** package and select **Add File** from the menu. Now, click the **Project File...** button to bring up a file browser window. Unfortunately, the file we need has not yet been created, so instead of browsing for it, just enter the name ***/SimpleCalc/CalculatorEJB-client.jar*** and click **OK**. That file must be stored in the WEB-INF/lib folder within the web application archive, so enter ***WEB-INF/lib*** in the *Prefix* field and click **OK**.

The WAR file must also have the HTML page we created earlier. This will be the default page users see when they visit our application. Right-click on the *CalculatorWeb.war* file and select the **Add Folder** option. Browse to find the **docroot** folder and add it. That should complete our WAR package.

Finally we need to create one more package. The last package must be an Enterprise Application Archive that contains all of the other JAR and WAR files we just defined. This will be the complete application and all support files needed by JBoss.

Click on **Add Standard** once last time. Enter ***CalculatorApp.ear*** in the *Name* field and and select *Standard-EAR.ear* from the list. Click **OK** to create the new package definition.

Once again, expand the tree and remove the MANIFEST.MF file.

Next, right-click on the **CalculatorApp.ear** file and select **Add File** from the menu. Add the ***'CalculatorWeb.war'*** file (again you must enter the name manually since it does not yet exist). Repeat and add the ***'CalculatorEJB.jar'*** file also.

The packaging configuration is now complete, so click **OK** to save the packaging settings.

Now you can generate the archive files by right-clicking on the project and selecting *Run Packaging* from the menu. If the packaging is successful, you should now see the new files in the Project Explorer window.

Step 12: Deploy the Application

You can now install the application by right-clicking the ***CalculatorApp.ear*** file and selecting *Deployment* -> *Deploy To* from the popup menu.

Select the desired JBoss installation where you want to deploy the application and click OK.

> NOTE: You can add additional JBoss targets by creating a new entries int the Debug setup window. If you did not see any JBoss options in the last step, use Run > Debug to define a new Jboss configuration.

Step 13: Test the Application

Use your web brower to visit http://localhost:8080/Calculator/. You should see a form where you can enter 2 numbers and request a calculation.

Accessing the Bean from a normal Java Applications

Once your Bean is deployed under a JBoss server, any Java applications can access it if needed. Let's create a simple console application that uses our shiny new bean.

Step 1: Create a new Java Project

Select *File -> New -> Project*

Highlight the *Java Project* wizard and click **Next**

Enter ***CalcClientApp*** for the *Project name* and click **Next**

> NOTE: You may wish to create separate source and bin folders for the code and class files like we did for the bean project. You can also make this the default behavior by setting the appropriate options under *Window -> Preferences -> Build Path.*

Visit the *Libraries* tab where we must add 2 JARs to our project.

Click on **Add JARs** and browse down into the *SimpleCalc* project and select the ***CalculatorEJB-client.jar*** file.

Next, click on **Add External JARs** and browse to find the ***client*** folder under your JBoss installation. Select the ***jbossall-client.jar*** file.

Step 2: Create the source code

Use *File -> New Class* to create a new class named ***CalcClient***. Make sure the option to create a main method is enabled.

Modify the source code as shown below:

```
import java.util.Hashtable;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

import edu.uah.coned.interfaces.Calculator;
import edu.uah.coned.interfaces.CalculatorHome;

/*
 * Created on May 23, 2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

/**
 * @author randy
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class CalcClient {

   private static Calculator bean = null;

   public static void main(String[] args) {
      try {

         ////////////////////////////////////////////////////////////
         // NOTE: You can put these values in jndi.properties instead
         //       of hard-coding them inside the application.

         Hashtable ht = new Hashtable();

         ht.put
(InitialContext.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces.NamingContext
Factory");
         ht.put(InitialContext.PROVIDER_URL,"jnp://localhost:1099");
         ht.put
(InitialContext.URL_PKG_PREFIXES,"org.jboss.naming:org.jnp.interfaces");

         // Find and create a reference to the bean using JNDI
         Context context = new InitialContext(ht);
         Object ref = context.lookup("ejb/Calculator");
         CalculatorHome home = (CalculatorHome)
PortableRemoteObject.narrow(ref, CalculatorHome.class);
         bean = home.create();

         System.out.println("4 + 3 = " + bean.add(4,2));
         System.out.println("4 - 3 = " + bean.subtract(4,3));
         System.out.println("4 * 3 = " + bean.multiply(4,3));
         System.out.println("4 / 3 = " + bean.divide(4,3));

         // System.out.println("Expecting an error.");
         // System.out.println("4 / 0 = " + bean.divide(4,0));

      } catch (Exception e) {
         System.out.println("Error: " + e.getMessage());
         e.printStackTrace();
      } finally {
         if (bean != null) {
            try {
               bean.remove();
            } catch (Exception e) {
               System.out.println("Error removing bean:" + e.getMessage
());
               e.printStackTrace();
            }
         }
      }
   }
}
```

Step 3: Test the Client

At this point you should be able to run and/or debug the code, after making sure JBoss is running and that the CalculatorBean is deployed.

> NOTE: If you forget to add the jbossall-client.jar or CalculatorEJB-client.jar when creating the project, you may do so in the Libraries tab of the appropriate Debug configuration.