Web Application Development

Title:
Working with Java Server Pages and Servlets

Benny Schaich

Copyright

.

Contents

Scenario Implementation 4
Exercise description 4
Building a simple JSP example 5
Setup Eclipse : 5
Create an Eclipse Project 6
Create a new JSP page 8
Deployment 9
Using the common user management 12
Adding security constraints 12
Deploying the Web Application with the Security Constraint 15
JCO connection with User credentials 17
Calling a BAPI on SAP Web AS 22

Scenario Implementation

- Eclipse Project SDK (Download from http://www.eclipse.org/)
- SAP Java EU is installed (Please ensure that you are using the EU with the latest Patch Level (this script is based on Patch Level 2)
- Access to a SAP Web AS 6.20 with a user which has at least one SAP role (which role doesn't matter, we use the "ASAP_AUTORENUMGEBUNG" role for this example)
- The local J2EE engine is configured to connect to the Web AS.
  This requires the same settings as in the J2EE Engine of the central Web AS 6.20. See Integrating the security functions of the SAP Web AS with the J2EE Engine
- The local J2EE Engine is up and running (Start – Programs – SAP J2EE Engine 6.20 – Stand alone server)

Exercise description

This exercise shows how to implement, deploy and run a simple Java Server Page. The Java Server Page will use the Flight Example known from ABAP programming.

Building a simple JSP example

Creating a simple JSP example consists of several steps:

- Configure Eclipse to use the SAP J2EE Engine
- Create an Eclipse project
- Create a JSP page
- Create an Enterprise Archive EAR file that contains the Web Application
- Build the project and deploy the EAR file to the J2EE Engine

Setup Eclipse :

Create a new Project with “File / New/ Project…”.

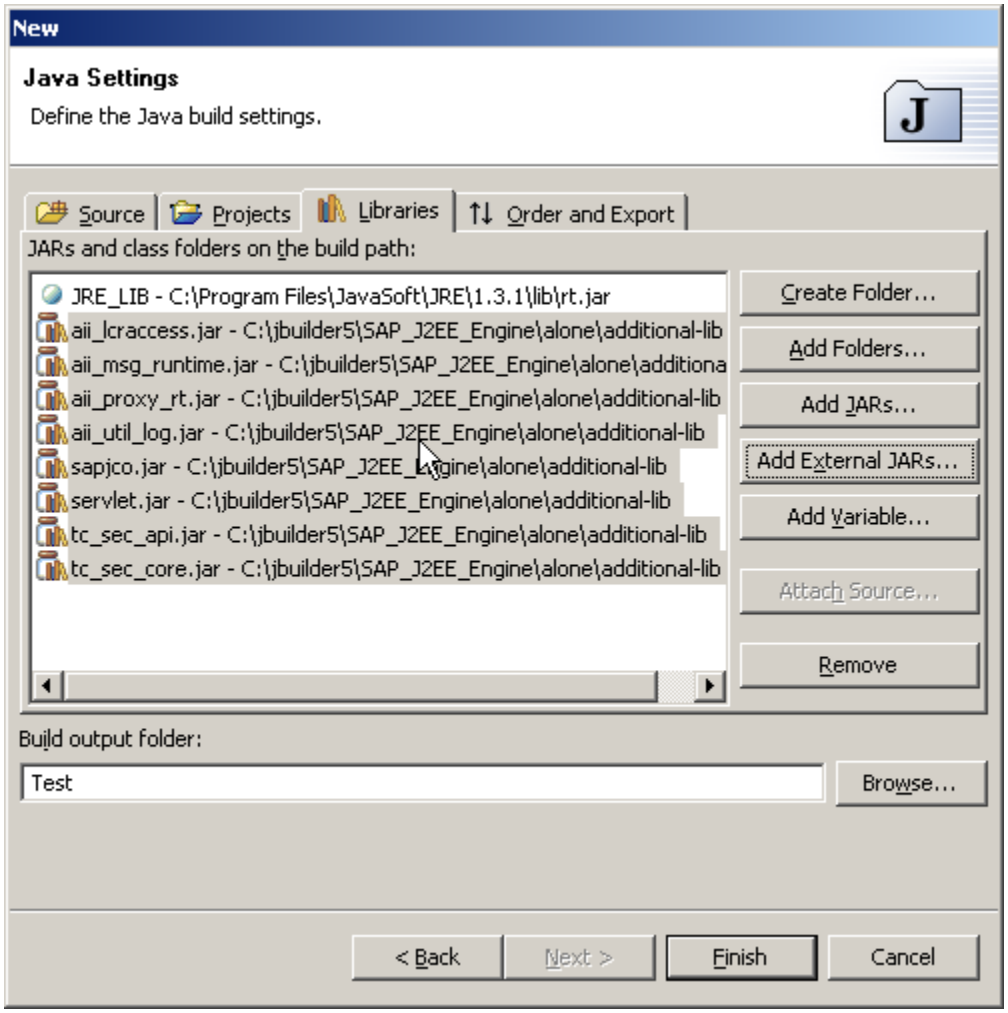
Click on “Next” and enter the Project name “SAPPrerequisites”

Click on “Next” and the “Libraries” tab.

Click “Add External JAR’s…”

Go to Directory <J2EE Engine InstallDirectory>/alone/additional-lib and select the following files:

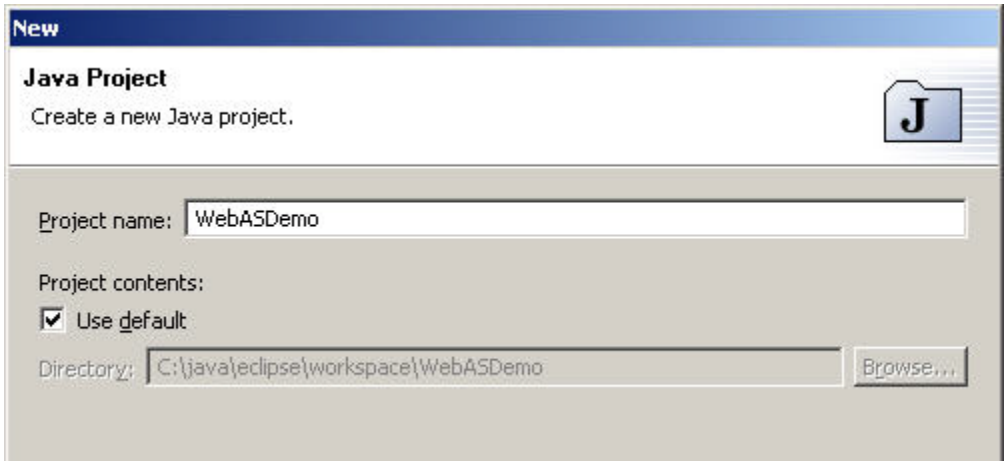
- Tc_sec_api.jar
- Tc_sec_core.jar
- Servlet.jar
- Sapjco.jar
- And all files that start with aii…



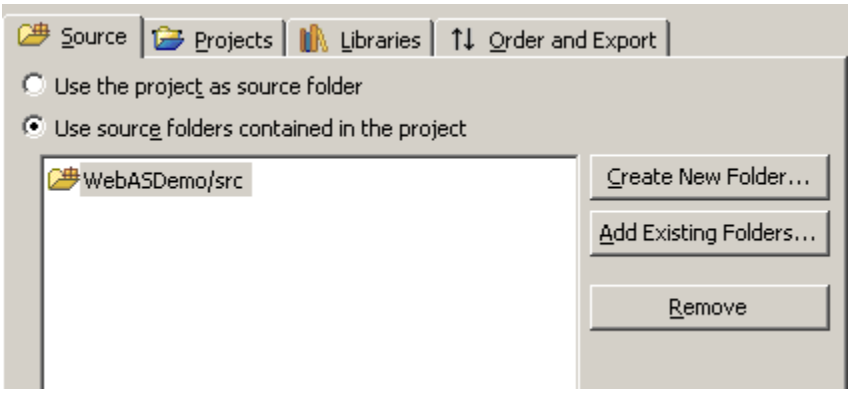
Go to the “Order and Export” tab and click on “Select All” Button

Click “Finish”.

Create an Eclipse Project

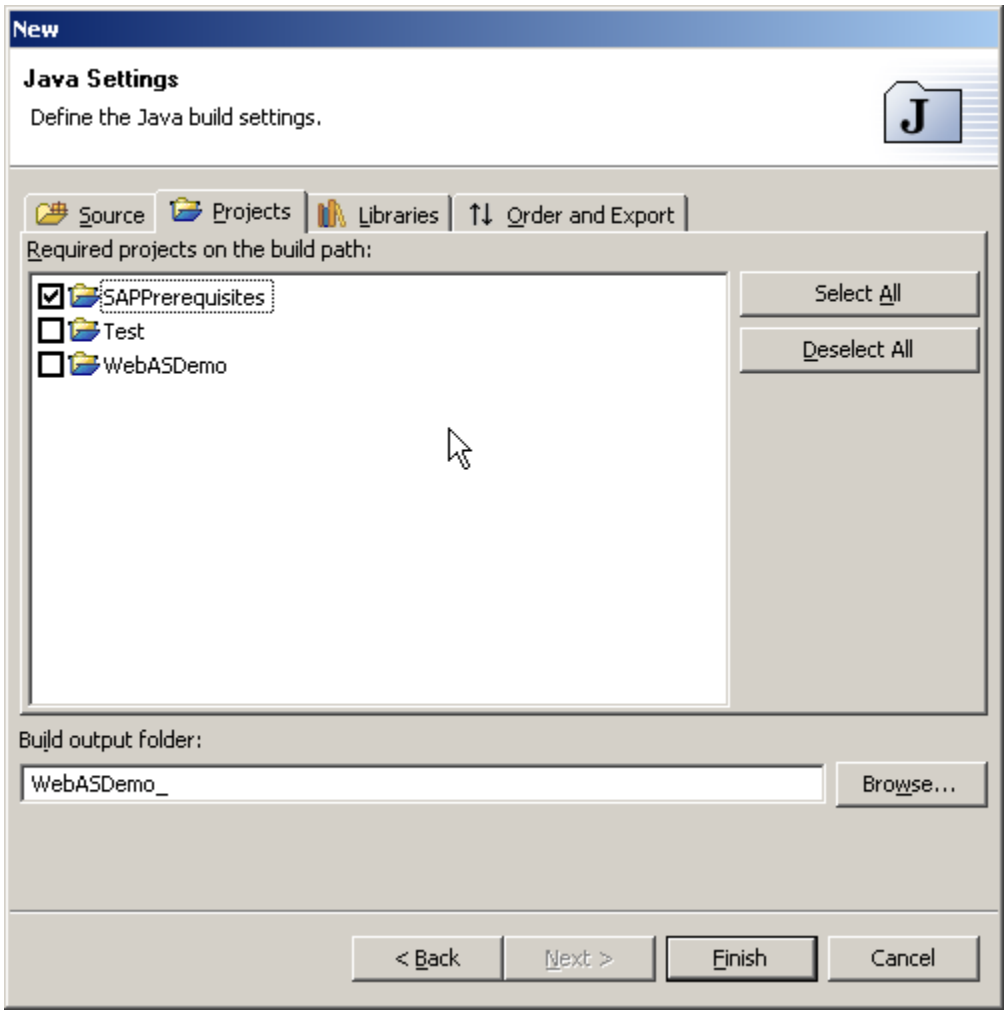


Create a new Project “WebASDemo” in Eclipse. Click “Next>”



Chose Radio Button “Use source folders….”.

Create New Folder named “src”. (Click “yes” if asked to make a /bin directory also)

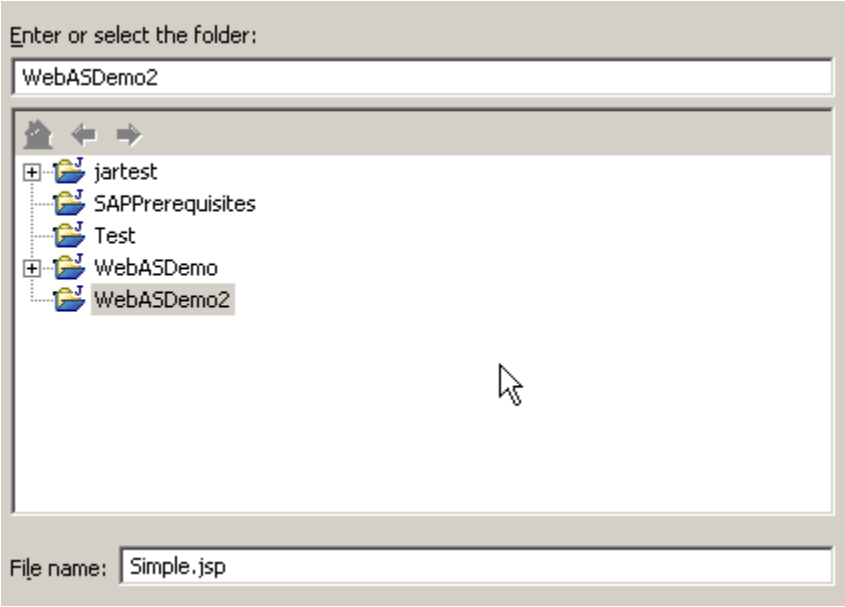


Select the tab Projects.

- Mark Checkbox of the SAPPrerequisites Project and click “finish”.

Create a new JSP page

Create a new Java Server Page, (with “File…” “New” and the “File” Option)



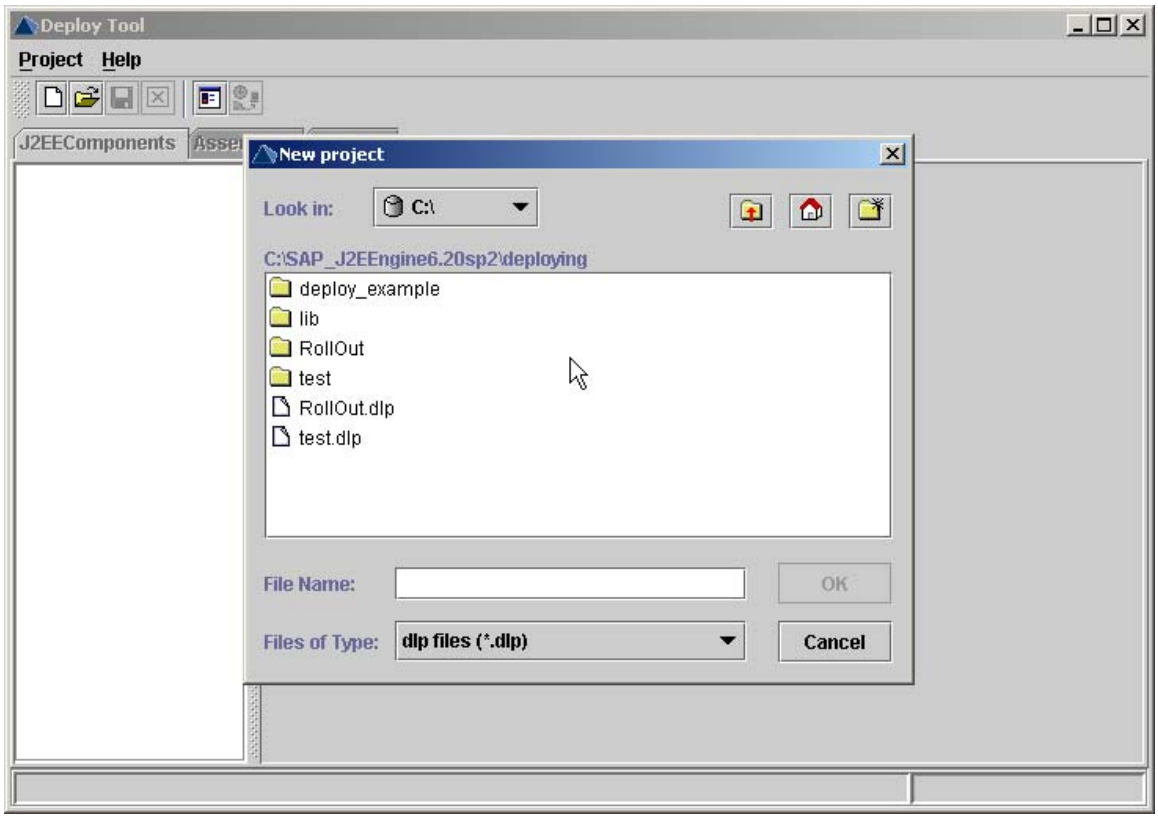
Give it a name for example “Simple.jsp”. Click “Finish”.

Insert the following Text to “Simple.jsp”

```
<html>
<head>
<title>
Simple
</title>
</head>
<body>
<h1>
JavaDemo
</h1>
</body>
</html>
```

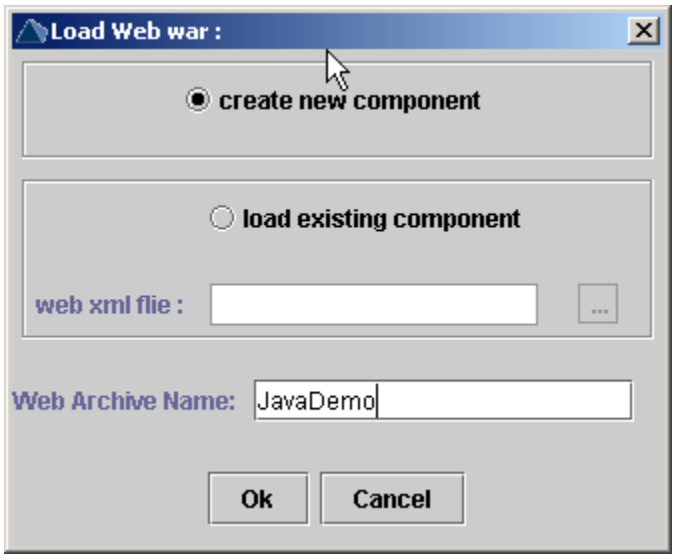
Save your work with the “Save All” Option of the “File” Menu.

Deployment



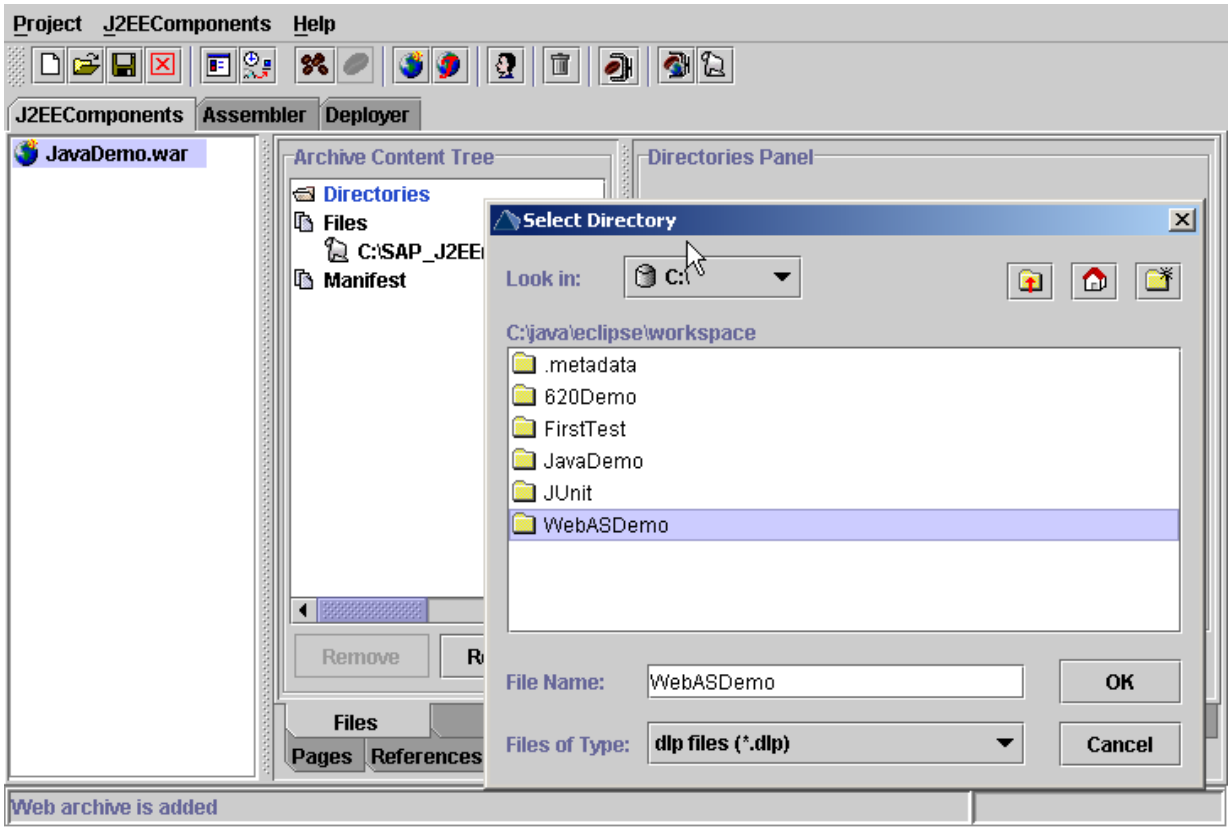
Start the SAP Deployment tool by getting it from the “Start” Menu and Open a new project. Call it WebASDemo.

Use the “Add Web” option from the “J2EEComponents” Menu.



Write in the name of the Web Archive: “JavaDemo” and press OK.

Click on the emerging “JavaDemo.war” file and on the “Directories” list in the Archive Content Tree.



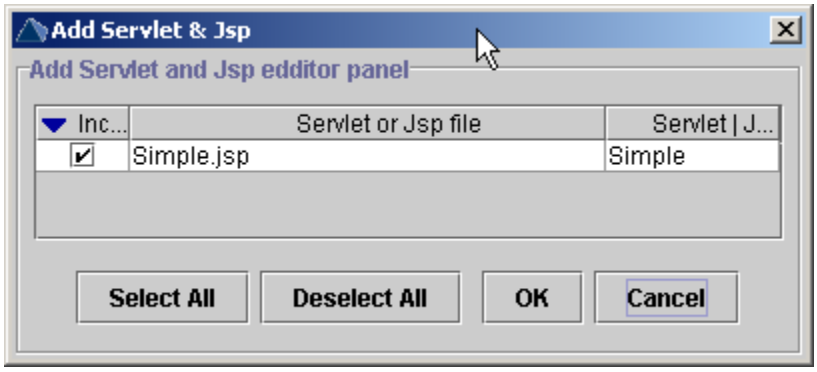
Search for the directory, where your project is located. This usually should be in your Eclipse workspace path "…\eclipse\workspace\WebASDemo".

Chose it and press OK.

Press "Add Directory" in the "Directories Panel".

Select "Add Servlet|JSP" ->"Add from Files…" from the "J2EEComponents" Menu.

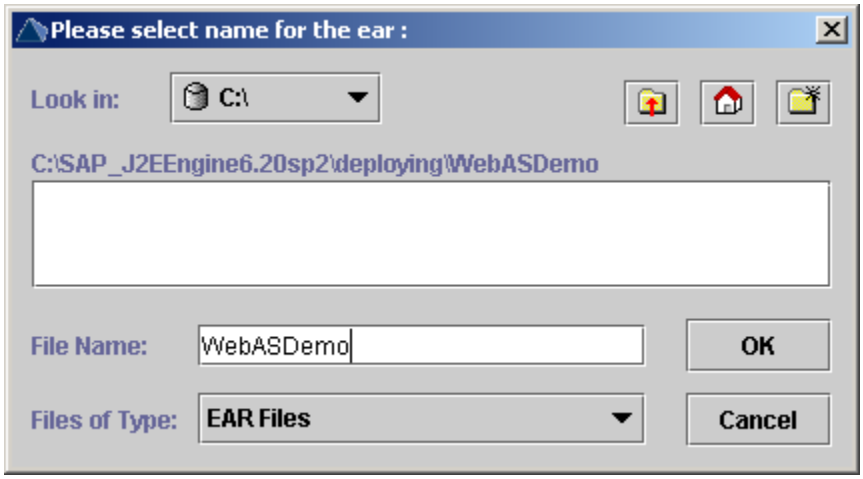
The only existing JSP file is already selected there



Click OK.

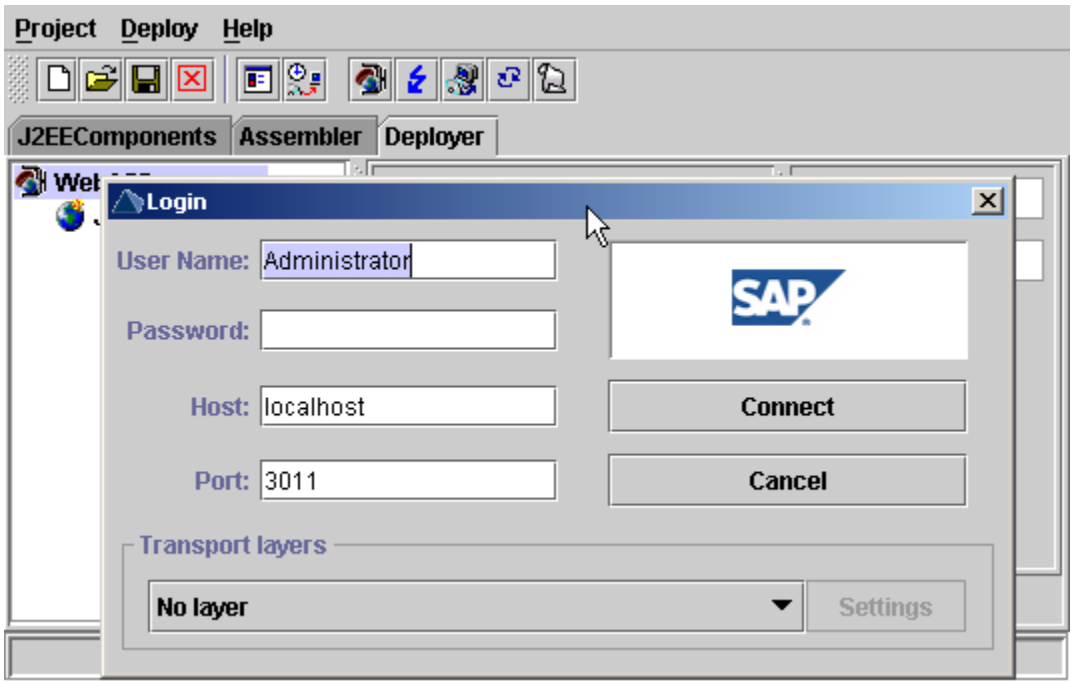
Click "Make Archive" in the "J2EEComponents" Menu.

Click on "Assembler" Tab, select "WebASDemo" and chose "Make EAR" this time from the "Assembler" Menu that appears instead of the "J2EEComponents" Menu.



Insert the name for the EAR file and press “OK” button.

Select the “Deployer” Tab.



Connect to your local J2EE Server ( button) and deploy the Application with “Deploy” – “Deployment” – “Deploy EAR” or the deployment button. 

Try to access the JSP in the Browser with the link http://localhost/JavaDemo/Simple.jsp

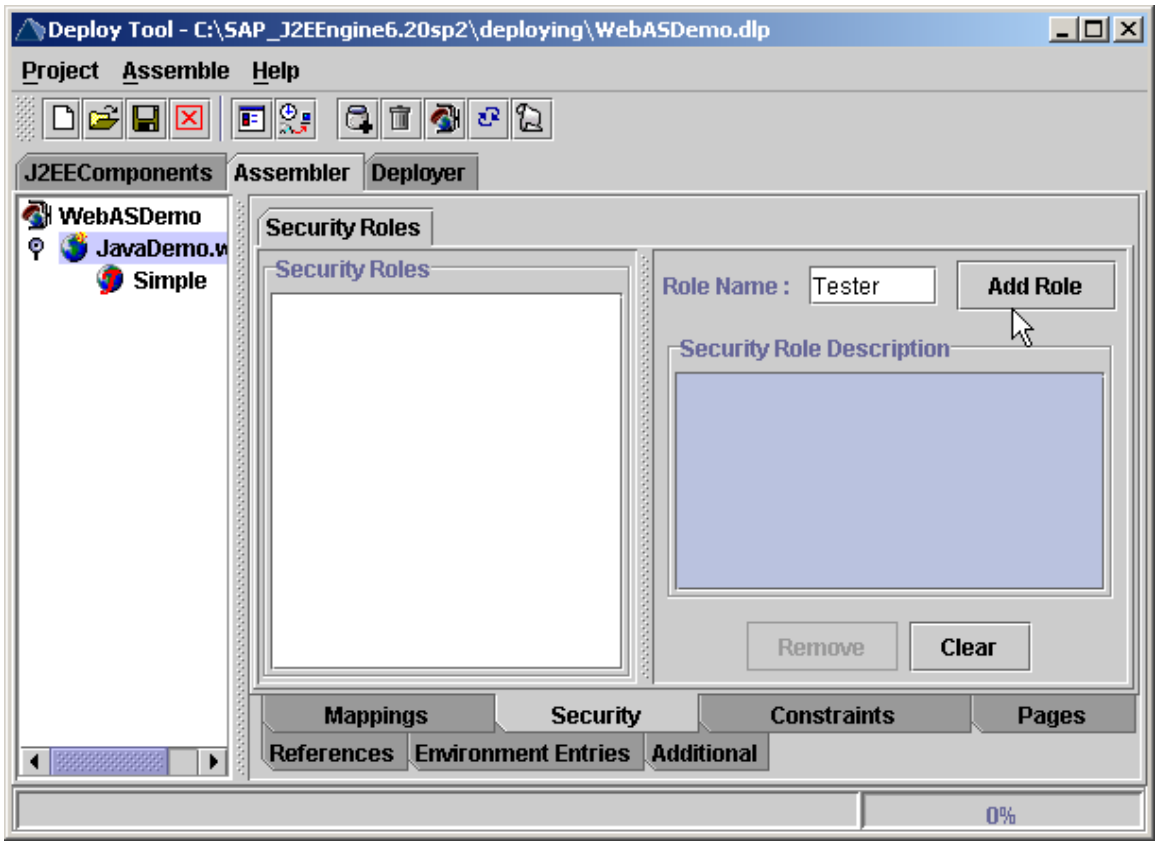
Please ensure the link is case sensitive!

Using the common user management

Using the central Web AS User management for accessing the JSP example needs several additional steps:

- Adding a security constraint and a role to the Web Application
- Mapping this role to a Web AS role of the central system at deployment

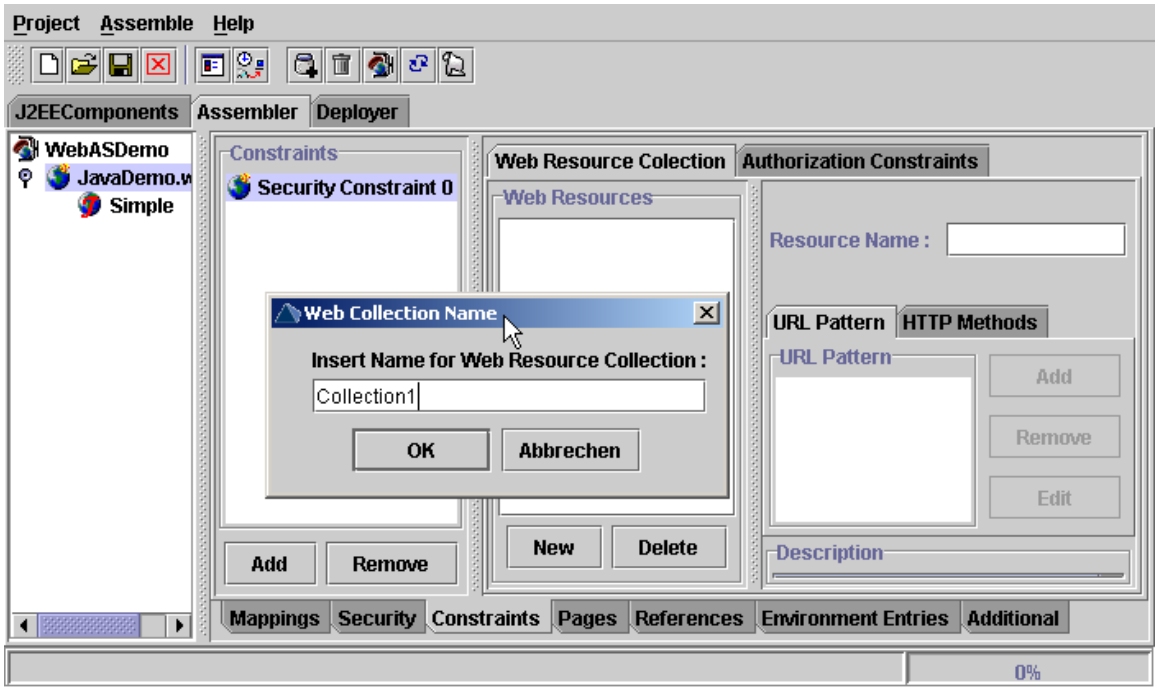
Adding security constraints



Switch to the “Assembler” Tag in the Deployment Tool and select the “JavaDemo.war” file.

Go to the “Security” Tab and enter the Role Name “Tester”.

Click the “Add Role” button.



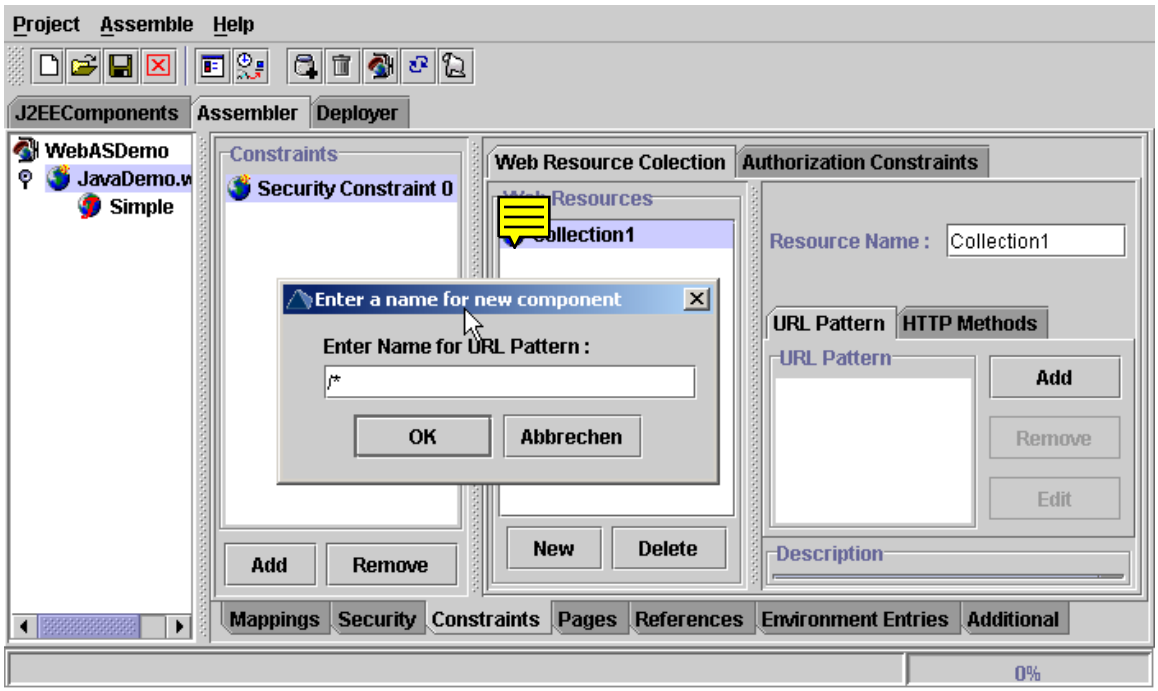
Click on “Constraints” Tab.

Click the “Add” Button for Constraints.

Click the “New” Button in the Web Resource Collection.

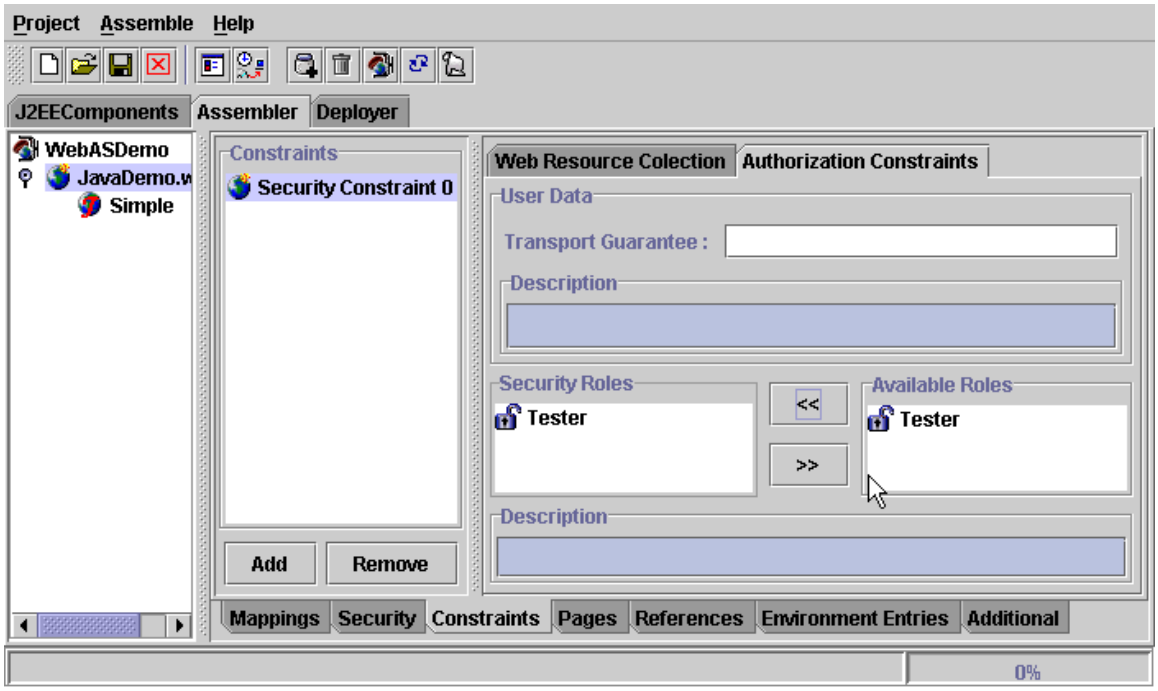
Insert the name for the Resource Collection: “Collection1”.

Press “OK” Button and select the newly created Collection.

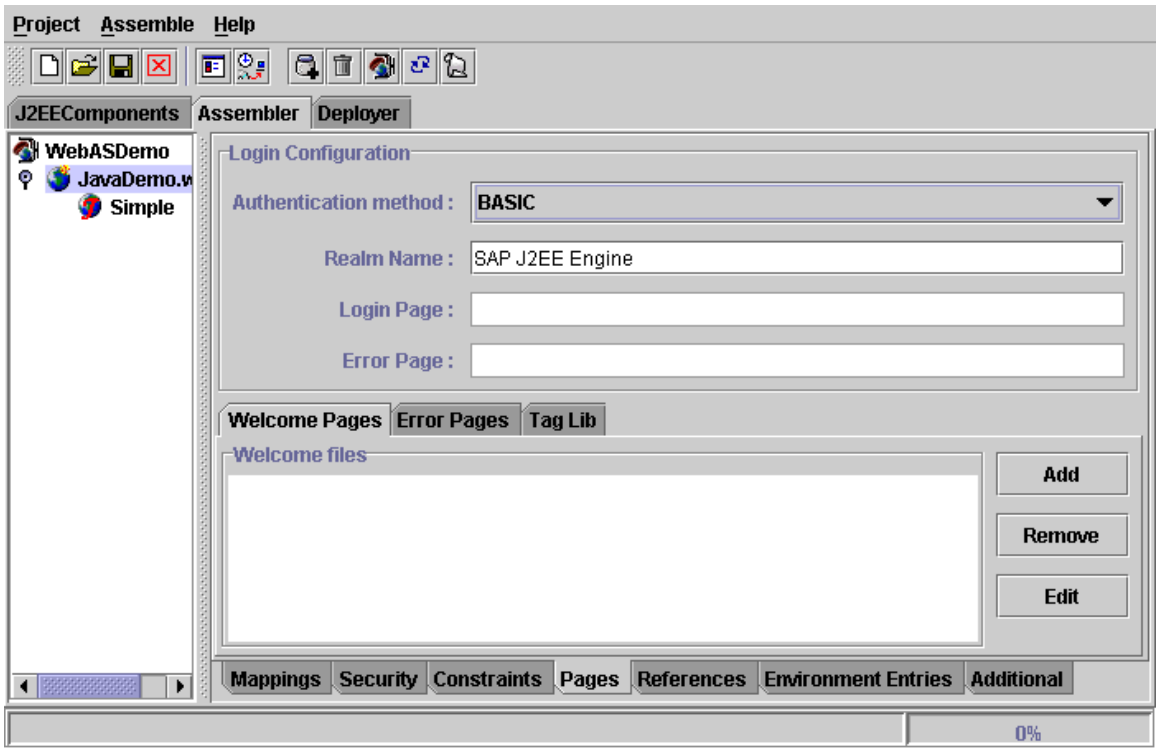


Select the “URL Pattern” Pad and press the “Add” Button.

Insert Pattern "/*" (a slash followed by a star) and press OK.



Select "Authorization Constraints" Tab and select "Tester" from the "Available Roles" list. Then press the arrow Button (<<) to transport the role to the "Security Roles" list.



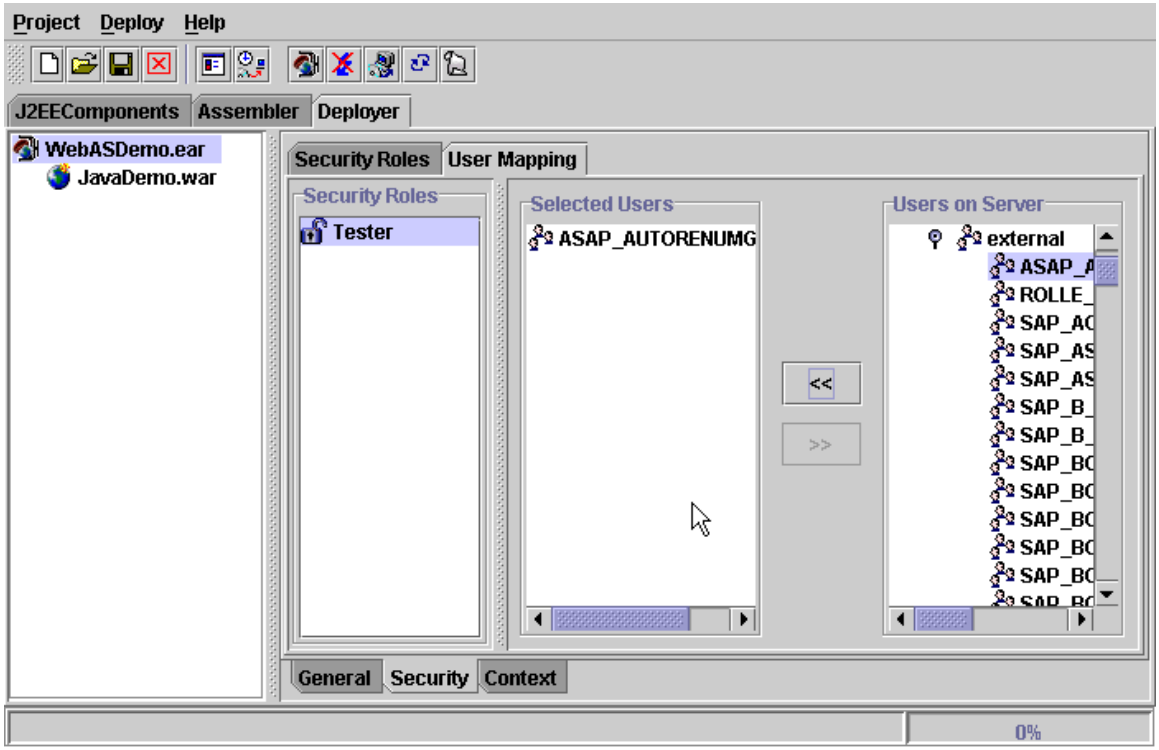
Select the "Pages" Tab and enter the Realm Name into the appropriate Field.

Select "Authentication Method" BASIC.

Now select Option “Make EAR” from the “Assemble” Menu again.

Deploying the Web Application with Security Constraint

If not already, connect to your J2EE Engine.

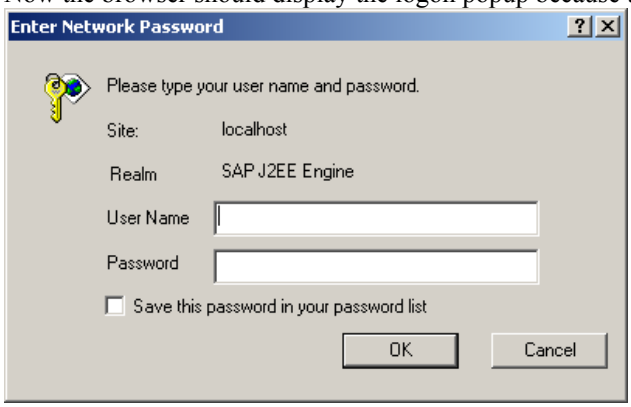


Select the WebASDemo.ear, switch to the Security tab below and then to the User Mapping tab. Expand the Users on server tree. You should now see under the guests – external node the SAP roles of the central Web AS. (If not your local J2EE Engine is not configured properly, see Prerequisites).

Select a role (for example the first “ASAP_AUTORENUMGEBUNG” role) and map it to your “Tester” role. The SAP role appears in the Selected Users window.

Now deploy the Application again and access it in the browser as before with the link http://localhost/JavaDemo/Simple.jsp

Now the browser should display the logon popup because the page requires a user logon.



Use a SAP user who has the specified SAP role (“ASAP_AUTORENUMGEBUNG” in our example) of the central SAP Web AS to logon.

You can also display the current user name in the “Simple.jsp” JSP directly by using the UserPrincipal object of the HTTP request:

Enter the yellow code:

```
…
<body>
<h1>
JavaDemo
<p>
Hello Mr. <%= request.getUserPrincipal().getName().toString()%>
</h1>
</body>
```

Save Editor contents.

Go to Deployment Tool and select option “Make All” in Menu “Project”. This deploys the project and you can again go to your browser and see the changes.

JCO connection with User credentials

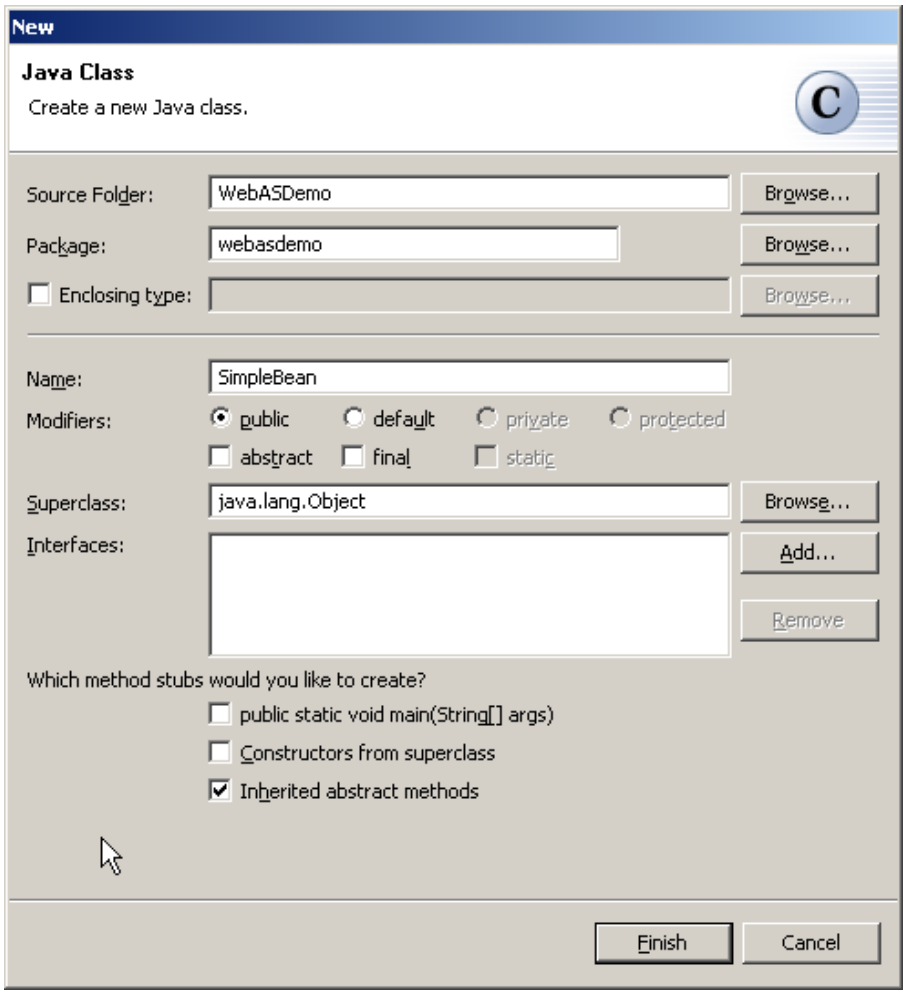
The J2EE engine runs now the web application with the user credentials of an SAP user. It is now possible to use this user credentials to get a valid JCO connection to the ABAP part of the Web AS to call a RFC or BAPI. This requires the following steps

- Including the necessary libraries into the project
- Enhancing the SimpleBean class with JCO method calls
- Calling this methods in the Simple.jsp Java Server Page

Accessing the Web AS via JCO will be done by the SimpleBean.java class.

Adding a JavaBean

Open a new Java class with File ->New -> Class.



Enter the package name “webasdemo” and the name of the class “SimpleBean”.

Add the code for

- Importing the necessary classes for JCO, SSO security and User Principal classes and the HTTPRequest
- A JCOClient object as private class attribute
- A method for opening the JCO connection
- A method for closing the connection

```
<html>
<head>
<title>
Simple
</title>
</head>
<jsp:useBean id="SimpleBeanId" scope="session"
class="webasdemo.SimpleBean" />
<jsp:setProperty name="SimpleBeanId" property="*" />
<body>
<h1>
SAPTechEd Java Demo
</h1>
<p>
Hello <%= request.getUserPrincipal().getName().toString()%>
<p>
<%= SimpleBeanId.openConnection( request )%> <br>
<%= SimpleBeanId.closeConnection()%>
</body>
</html>
```

The open and close calls of the JCO are encapsulated in try-catch statements and return exceptions as string in case of runtime errors.

The JCO object is created by the ConnectionFactory with the help of the UserPrincipal object. To get the Principal it is necessary to have access to the HTTP Request object of the JSP (or servlet), so a reference to the HTTP request has to be given to the openConnection method.

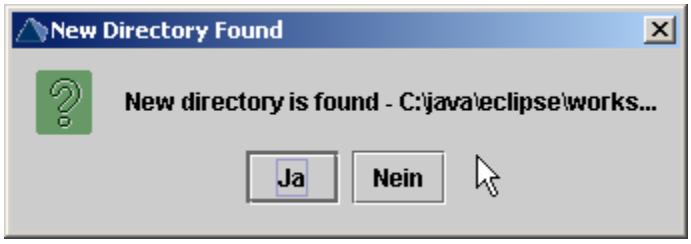
To try the JCO connection the methods are called in the “Simple.jsp” Java Server Page:

```
package webasdemo;

import com.sap.mw.jco.*;
import com.sap.security.sso.*;
import java.security.Principal;
import javax.servlet.http.HttpServletRequest;

public class SimpleBean {

  private JCO.Client jcoclient = null;

  public String openConnection( HttpServletRequest request ) {

     try{
       Principal principal = request.getUserPrincipal();
       ConnectionFactory cf = new ConnectionFactory();
       jcoclient = cf.getConnection(principal);
       jcoclient.connect();
       return "Connect OK!";
     }
     catch(Exception ex){
       return ex.toString();
     }
  }

  public String closeConnection(){
     try{
       jcoclient.disconnect();
       return "Close OK!";
     }
     catch(Exception ex){
       return ex.toString();
     }
  }

}
```

Chose Option “Save All” in the “File” Menu of Eclipse and switch to the Deployment Tool.

Deploy and Test

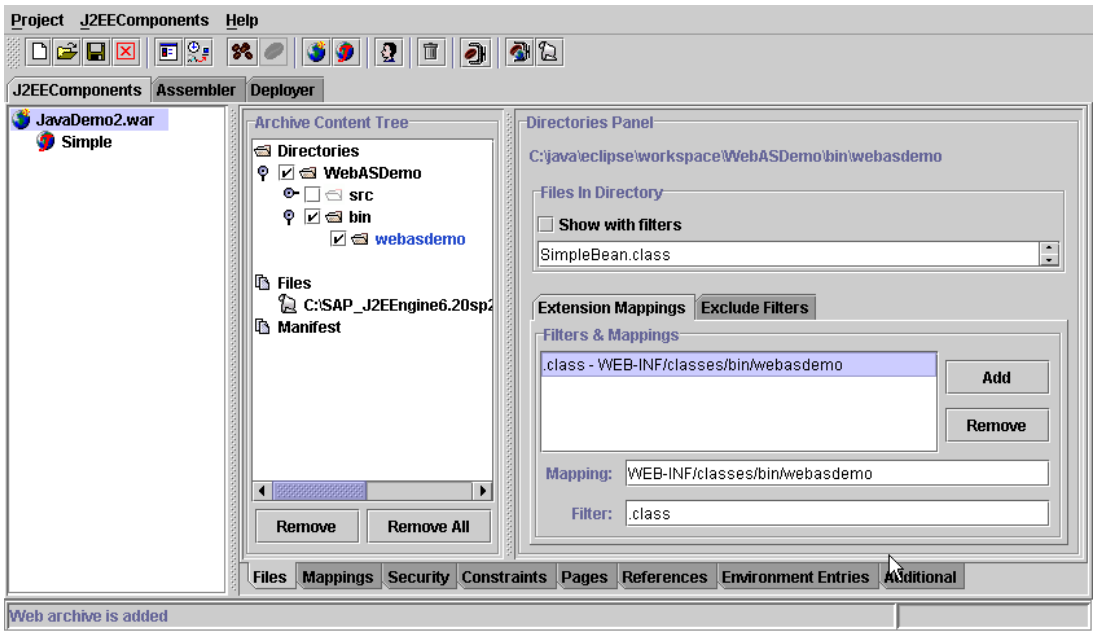
When going back to the Deploy tool, and clicking on the .WAR file, this will bring up a new window, because the directory for the package is found by the Deploy tool.



Click on Yes.

Now we need to tell the Deployment tool to include the just created class. Got to the J2EE Components Tab and select the directory in the “Archive Content Tree” that you entered. (WebASDemo in the example). Open the tree. You can deselect the \src directory to prevent it to be part of the ear. You need to keep the \bin directory, because the necessary file is in there.

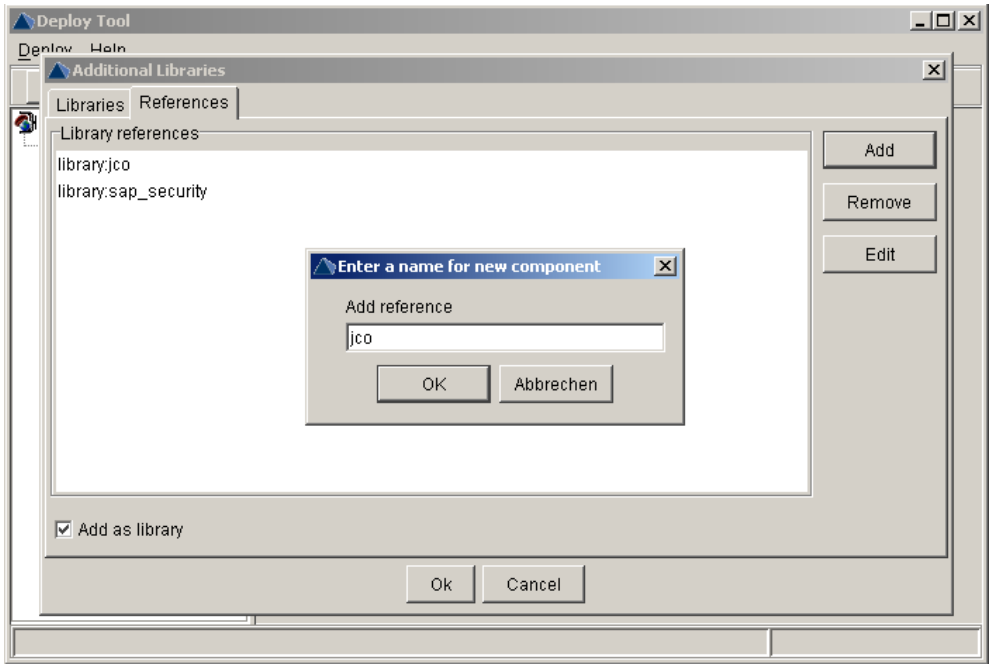
Now, go to the “directories Panel”, chose “Extension Mappings” and click on the automatically generated Mapping.



The mapping is the relative directory in the root of your application on the server, where your files will be located. As you can see, the mapping is “WEB-INF/classes/bin/webasdemo”.

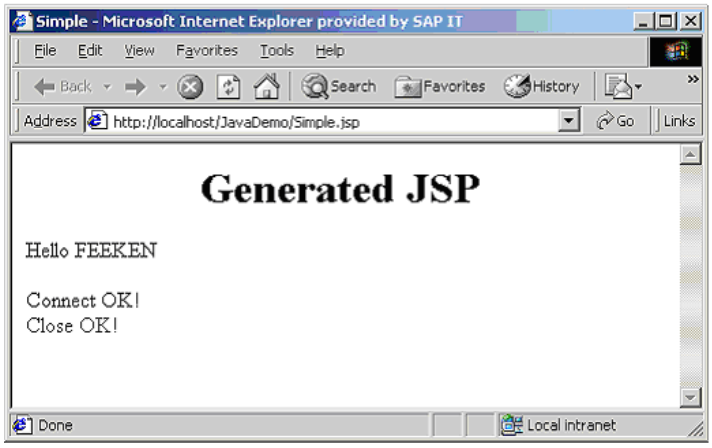
What does this mean? The standard defines that all classes have to be in a directory “WEB-INF/classes”. This will be the root from where the java classes are expected. Knowing that directory structures are mapping to java package names, you easily find out that something here would go wrong: Your package is named “webasdemo” and not “bin.webasdemo”.

Because of that you have to change the mapping to be “WEB_INF/classes/webasdemo”.



The used libraries for the JCO and the SSO Security classes must be declared at the deployment. Add the library names “jco” and “sap_security” in the Add reference pop up as library names. (These names are defined in the libraries.txt file of the J2EE Engine).

If you’re not connected anymore reconnect to the server and chose Option “Make All” Check it in the browser.



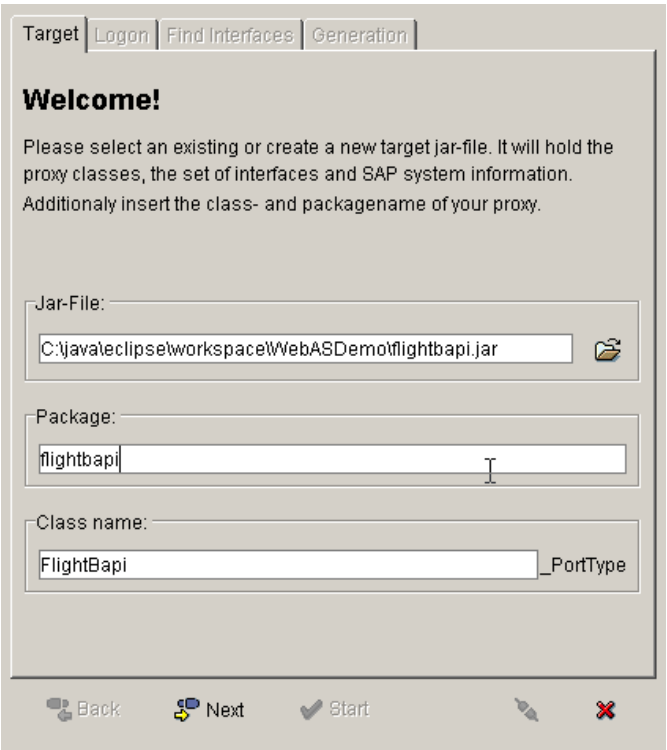
The page should show the successful return Strings of the open and close methods.

Call a BAPI on SAP Web AS

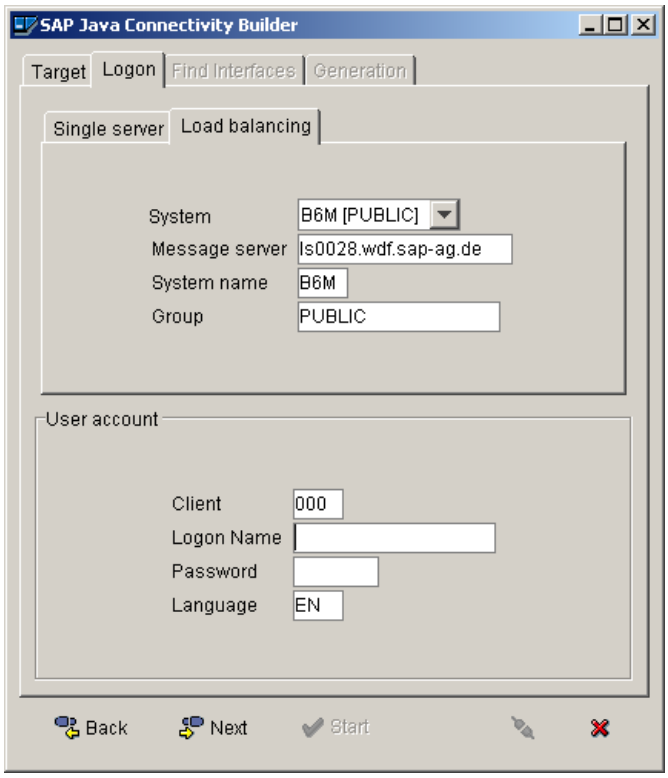
As last step of the Demo a BAPI in the Web AS should be called. We chose the BAPI_FLIGHT_GETLIST from the well-known training flight data model.

Generate Proxy Objects

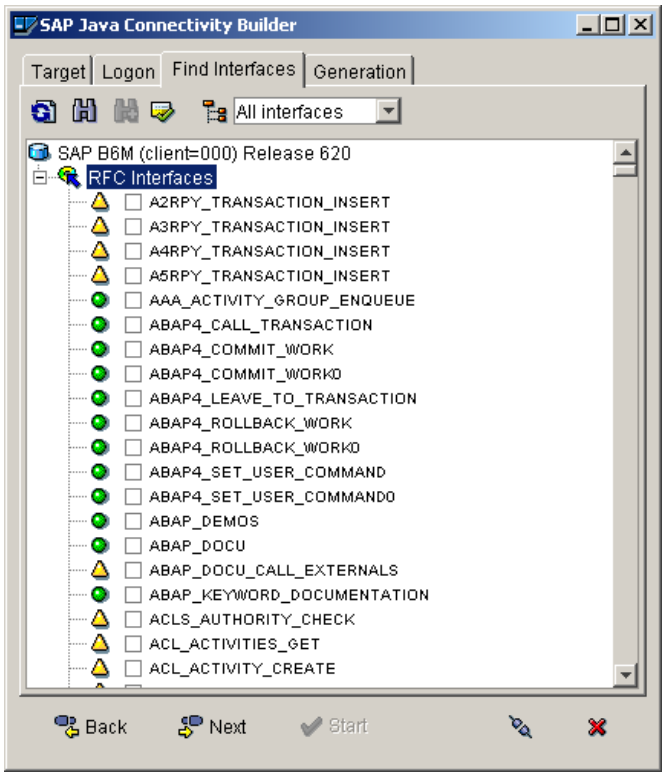
Open the SAP Connectivity Builder via “Start” “Programs”-“SAP Java Development Tools”-“Java Connectivity Builder Start Tool”.



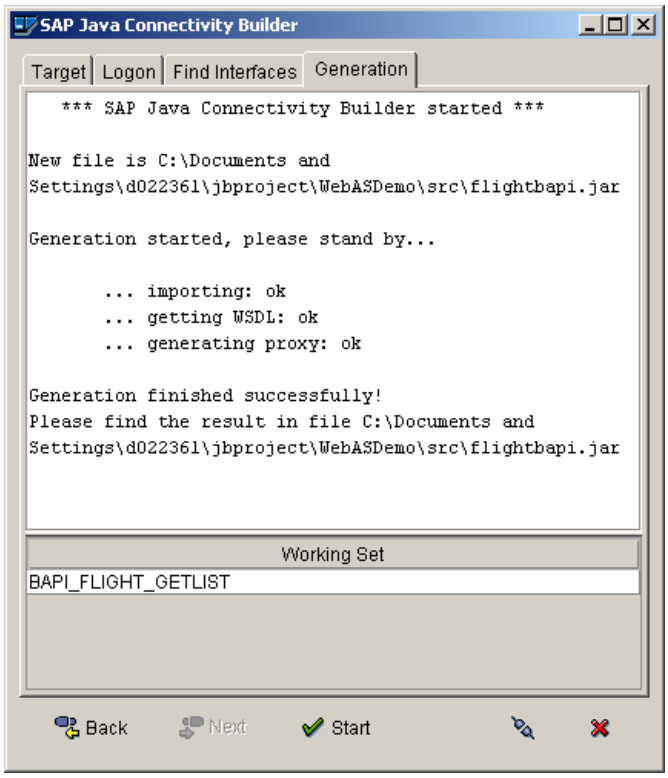
Chose the Path for the JAR file (preferably a path that lies in your project tree) with the filename for example “/flightbapi.jar”. Name the package that should contain the BAPI proxy classes (for example “flightbapi”) and name the main class that contains the main method call that calls the BAPI (for example “FlightBapi”). Click “Next” and confirm the creation of the new .jar file.



Connect to the central SAP Web AS 6.20 System and click “Next”



Expand the “RFC Interfaces Tree” (this may take some time), scroll down and select the “BAPI_FLIGHT_GETLIST” function module.

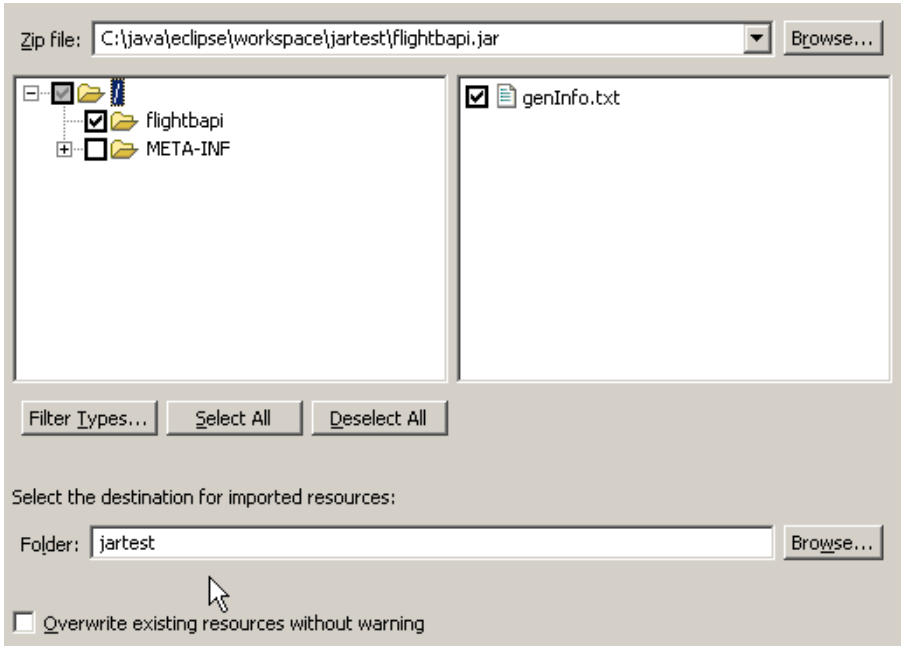


Click “Start”, the Java files for the BAPI proxy classes will be generated. Close the Connectivity Builder after the generation.

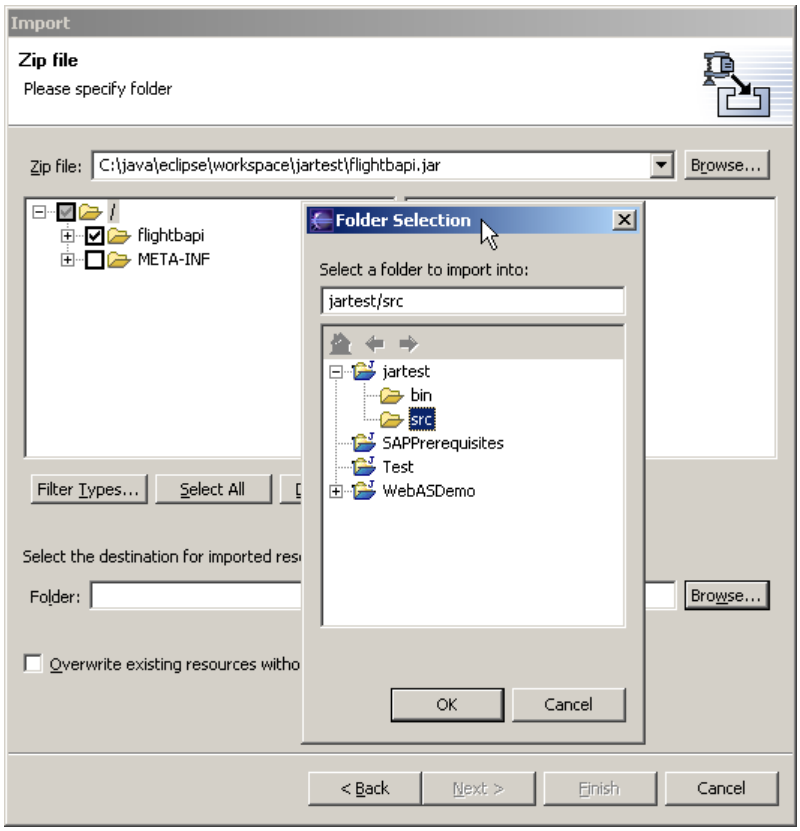
Importing the generated sources

Go to “File” – “Import” and select ZIP Files to be imported.

Click on “Next”.



Chose the flightbapi.jar file from the directory, where you defined it to be generated. You can deselect the Meta-Inf content and genInfo.txt, because we do not need such files.



Select the folder, where the contents shall be written by clicking the “Browse…” Button and chose the directory “src”.

Click OK

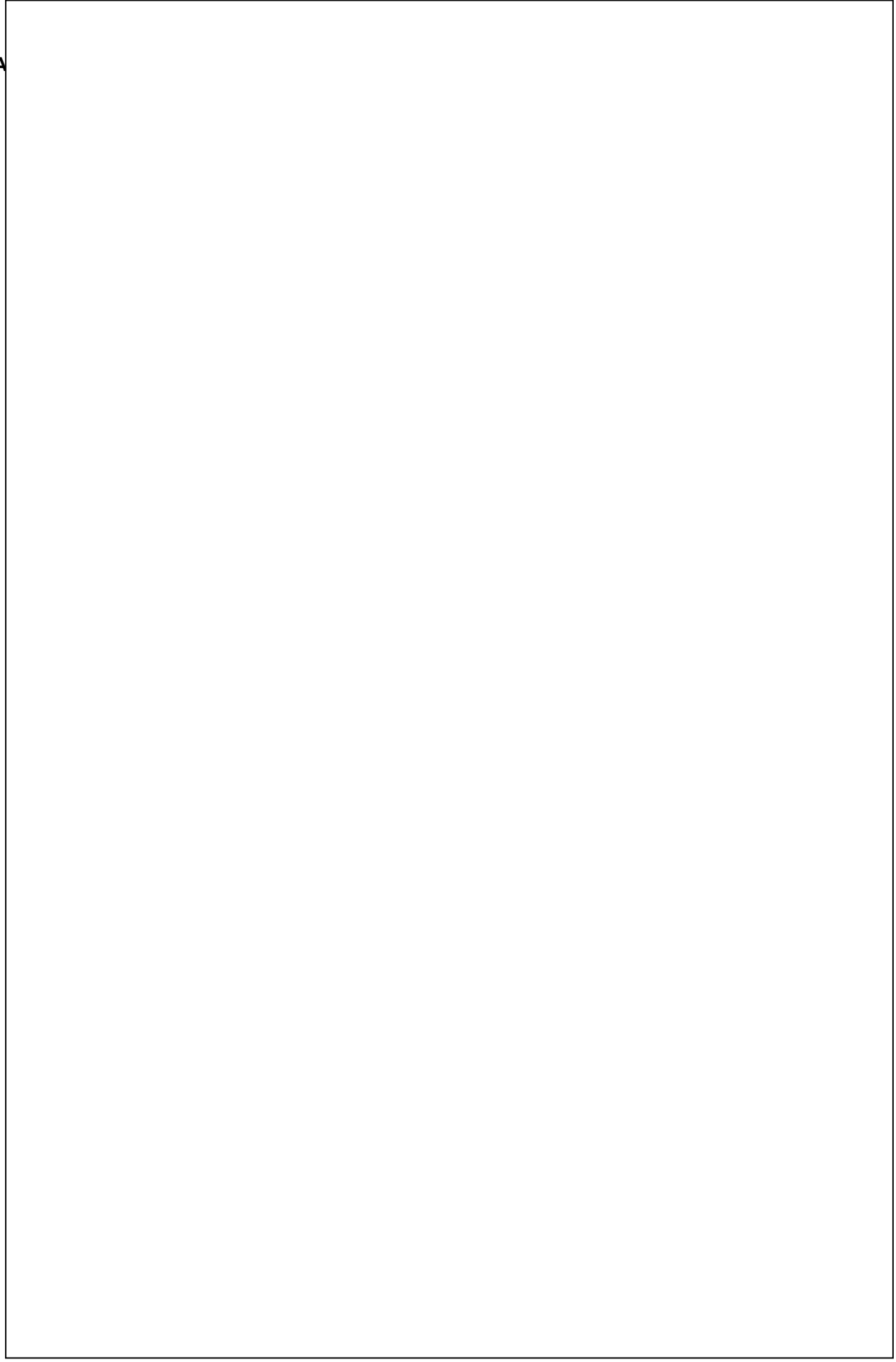
Click “Finish”

The jar file now is imported and the java files are compiled.

Note: Be aware that there currently seems to be a bug in Eclipse that results in error messages concerning the proxy files we will generate later. Those error messages do not have any impact on your project, because the code is compiled anyway and the class files are produced. You can just ignore these messages.

The proxy classes can now be used in the SimpleBean.java class. To get the flights information and display it in the JSP following steps are necessary:

- Import the bapiflight package into the SimpleBean class
- Add a public attribute “flightList” of the type BapisfldatTypeList
- Add a public method to the SimpleBean class that calls the BAPI by using the proxy classes and filling the flightList object with the resulting list of flights.
- Loop in the Simple.jsp page over the flightList object and display the flight information in a HTML table



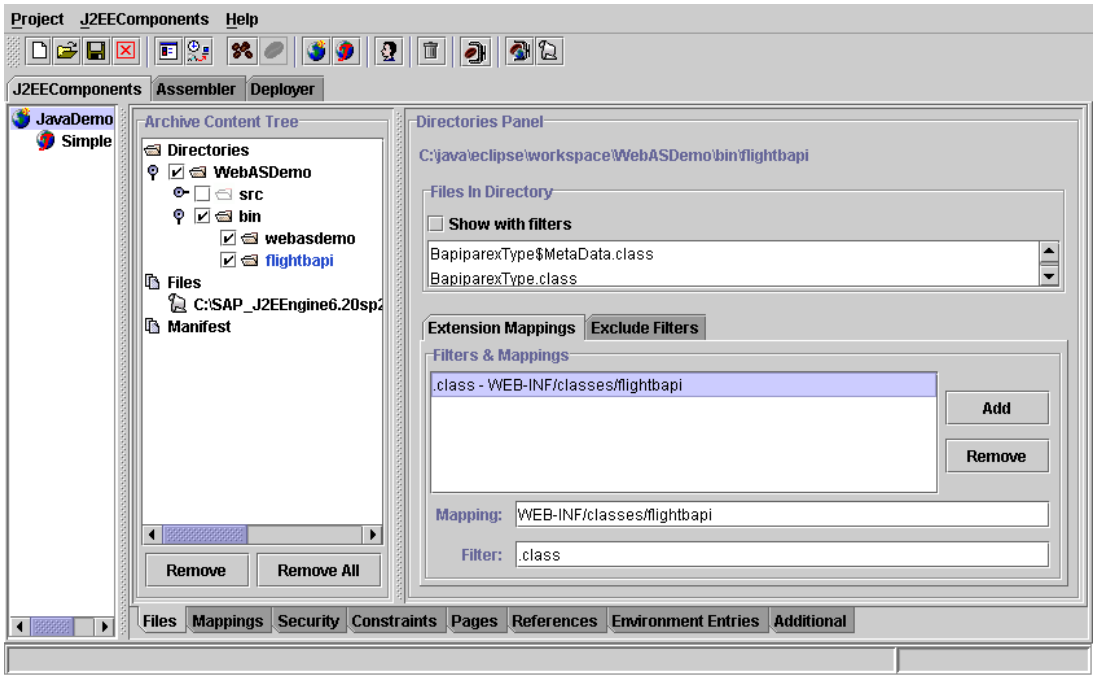
Add the highlighted code to the SimpleBean.java class:

To display the data the Simple.jsp can call the “getFlights()” method and access the public “flightList” attribute directly and use it in an HTML table

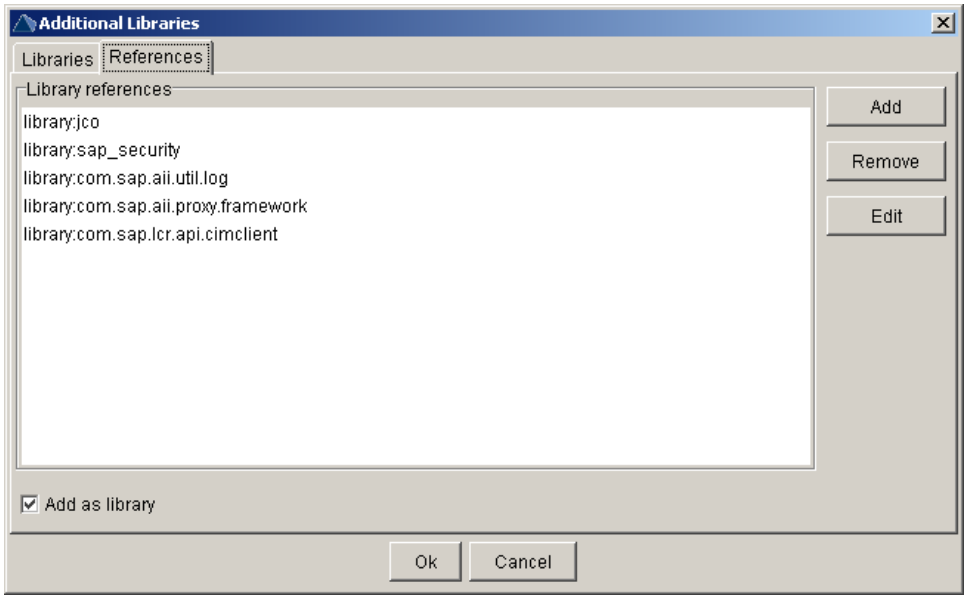
Add the highlighted code to your “Simple.jsp” page.

Save the project and open the deploy tool again.

```
<jsp:useBean id="SimpleBeanId" scope="session"
class="webasdemo.SimpleBean" />
<jsp:setProperty name="SimpleBeanId" property="*" />
<body>
<h1>
JBuilder Generated JSP
</h1>
<p>
Hello <%= request.getUserPrincipal().getName().toString()%>
<p>
<%= SimpleBeanId.openConnection( request )%> <br>
<%= SimpleBeanId.getFlights()%>
<%= SimpleBeanId.closeConnection()%>

<h2> Flight Data </h2>
<p>

<table border=1>
  <tr>
    <th> Carrier </th>
    <th> from </th>
    <th> to </th>
    <th> Date </th>
  </tr>
  <% for (int i=0; i < SimpleBeanId.flightList.size(); i++) {%>
    <tr>
      <td><%=
SimpleBeanId.flightList.getBapisfldatType(i).getAirline()%></td>
      <td><%=
SimpleBeanId.flightList.getBapisfldatType(i).getCityfrom()%></td>
      <td><%=
SimpleBeanId.flightList.getBapisfldatType(i).getCityto()%></td>
      <td><%=
SimpleBeanId.flightList.getBapisfldatType(i).getDeptime()%></td>
    </tr>
  <%}%>
</table>
</body>
</html>
```



Clicking on the war file should ask you to include the newly build directory. Answer this with yes. Again make sure that your mapping is right and that flightbapi maps to "WEB-INF/classes/flightbapi"



For the generated proxies several runtime classes have to be specified at the deploy time. Go to "Deploy" "Libraries", select the "References" tab and add the following library names:

```
com.sap.aii.util.log

com.sap.aii.proxy.framework

com.sap.lcr.api.cimclient
```

Deploy the application.

Go to your browser and start Simple.jsp