WA2214 Mastering JAX-RS REST Web Services and AJAX Clients - JBoss / Eclipse

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Setup the Development Environment....3
Lab 2 - Develop a Simple RESTful Service....16
Lab 3 - Extracting Information from a HTTP Request....23
Lab 4 - Designing a RESTful Service....30
Lab 5 - Add Support for XML Format....37
Lab 6 - Add Support for JSON....46
Lab 7 - Build the Order Web Service....52
Lab 8 - Complete the Order Service....62
Lab 9 - Developing a JAX-RS Client....72
Lab 10 - Introduction to AJAX....81
Lab 11 - Making POST Requests....93
Lab 12 - Basic DOM API....99
Lab 13 - AJAX Client using XML 104
Lab 14 - AJAX Client Using JSON....112
Lab 15 - Securing JAX-RS REST Services....115
Lab 16 - AJAX Client Security....127

Lab 1 - Setup the Development Environment

In this lab you will setup the environment for developing Web Services for JBoss. This includes installing JBoss Eclipse plug-ins.

Part 1 - Verify Java Install

Since this course uses Java 7, which is still relatively new, you will do a few quick steps just to verify everything is setup properly. Java 7 should have been installed as part of the class setup but these steps will help verify and avoid other issues later when they are harder to debug.

__1. Open a Windows command prompt. You can usually do this on most versions of Windows by selecting '**Start → Programs → Accessories → Command Prompt**'.

__2. Run the following command and make sure you get a Java "1.7.x" version. If you do not see this, the next steps will help figure out what setting is incorrect. Also, if you get an error about java not being "recognized" as a command the next steps will help figure out why.

```
java -version
```

```
C:\Documents and Settings\Student>java -version
java version "1.7.0_25"
Java(TM) SE Runtime Environment (build 1.7.0_25-b16)
Java HotSpot(TM) Client VM (build 23.25-b01, mixed mode, sharing)

C:\Documents and Settings\Student>_
```


__3. Run the following command to see the current value of the '**PATH**' environment variable. Check that you see the 'bin' directory of a Java 1.7 installation in the PATH. If you do not see Java 1.7 or if you see multiple versions of Java referenced this could be causing errors.

```
set PATH
```

```
C:\Documents and Settings\Student>set PATH
Path=C:\Progra~1\Java\jdk1.7.0_25\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\
System32\Wbem;C:\Program Files\ESTsoft\ALZip\;C:\Program Files\ESTsoft\ALZip\
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH

C:\Documents and Settings\Student>_
```

Note: Your actual value may be slightly different but it should be obvious that it is a Java (or "JDK") 1.7 version.

__4. Run the following command and look at the value of the '**JAVA_HOME**' environment variable. Make sure that it points to the root directory of a Java 1.7 installation. If you get a message about 'JAVA_HOME' not being defined this could also cause issues.

```
set JAVA_HOME
```

```
C:\Documents and Settings\Student>set JAVA_HOME
JAVA_HOME=C:\Progra~1\Java\jdk1.7.0_25

C:\Documents and Settings\Student>_
```

Note: Your actual value may be slightly different but it should be obvious that it is a Java (or "JDK") 1.7 version.

__5. If you have not seen output similar to that shown in the previous steps, inform your instructor. They can help determine if all students have the same (or similar) issue and the best way to fix it so it won't cause problems later.

__6. Close the command prompt.

Part 2 - Install JBoss Eclipse Plug-ins

Besides allowing us to use JBoss as a test server, the "JBoss Tools" Eclipse plug-ins have a few tools specific to web services. This section will install those tools. This was not done as part of the class setup because the steps are a little more involved. It will also be good for you to know how to install these tools if you want to obtain them for your own environment from:

http://www.jboss.org/tools/download

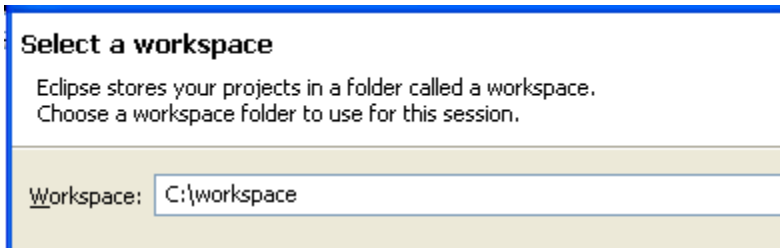
The JBoss Tools version you will use has already been downloaded and is compatible with the Eclipse version you are using.

__1. Start Eclipse by launching **C:\Software\eclipse\eclipse.exe**

Eclipse will launch.

You will be prompted to select a Workspace.

__2. Set the *Workspace* to **C:\workspace**

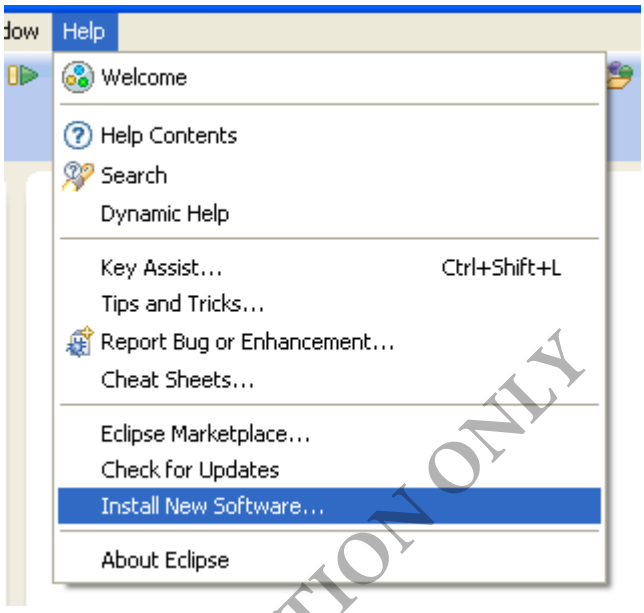


__3. Click **OK**.

Eclipse will start.

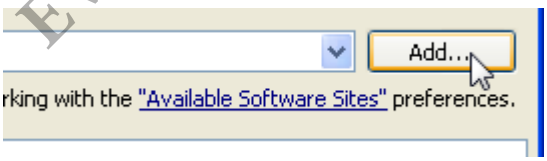
__4. Eclipse may be displaying the **Welcome** screen. Close the welcome screen by clicking the **x** in its tab.

__5. From the Eclipse menus select **Help → Install New Software...**

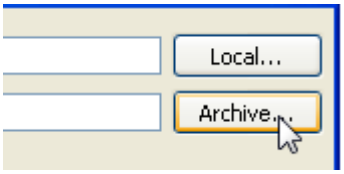


Note: If you get an error about some sites not being found it is because Eclipse can't connect to the Internet. Since you will be provided with a local copy of the updates you can ignore this error and click '**OK**'.

__6. Click the **Add** button on the upper right to add a new update site.



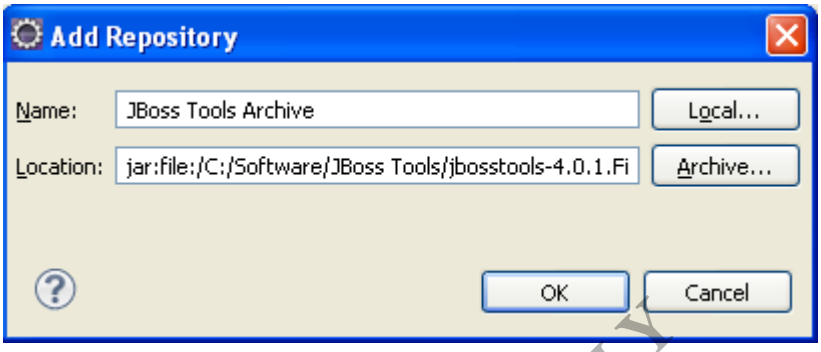
__7. Click the **Archive...** button.



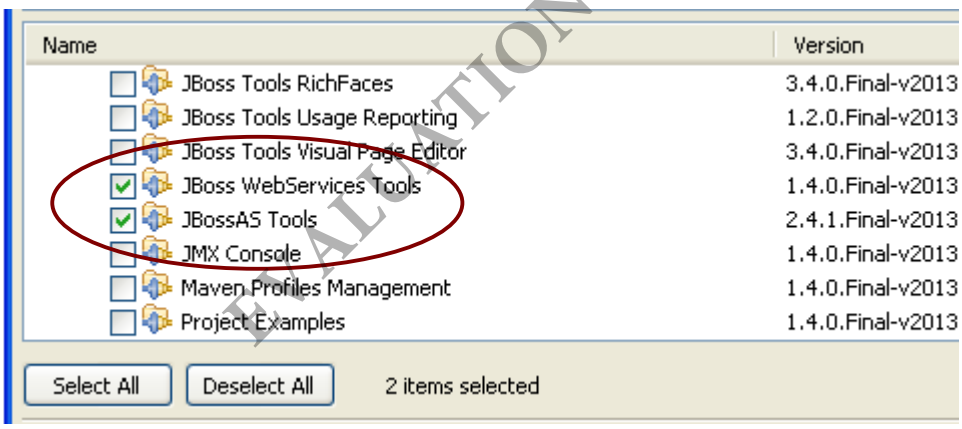
Note: You are using this option since the JBoss Tools are provided as a single Zip file with the class software.

__8. In the dialog to choose a file, find and open the file '**C:\Software\JBoss Tools\jbosstools-4.0.1.Final....zip**'. The end of the file name is not given as it may change but this should be the only Zip file which starts with that name at that location.

__9. Once you have chosen the appropriate Zip file, give the update site a name of '**JBoss Tools Archive**' and press the **OK** button when your options are similar to that shown below.

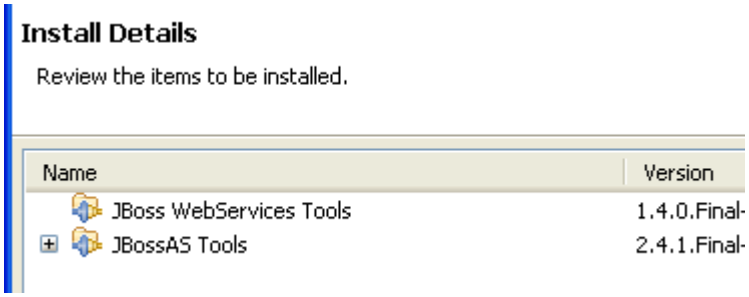


__10. In the list of available features expand the '**Abridged JBoss Tools 4.0**' category and check **ONLY** the '**JBoss WebServices Tools**' and the '**JBossAS Tools**' options as shown below and press the **Next** button.

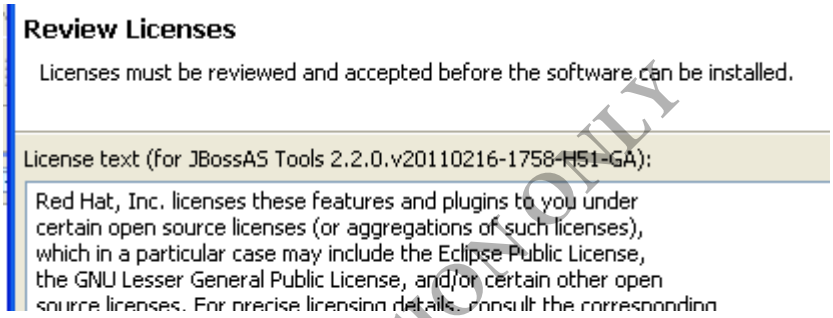


Note: Although JBoss Tools has other tools available we will only use these tools for the labs.

__11. You will see a list of software that will be installed as shown below. There should be no checkboxes next to any features. Press the **Next** button.

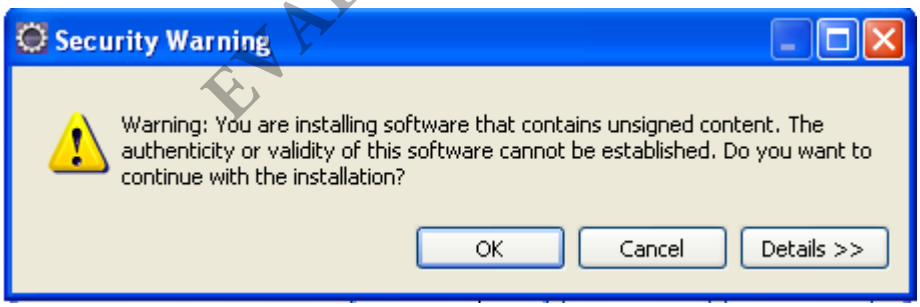


__12. On the '**Review Licenses**' screen, select the radio button in the lower right labeled '**I accept the terms of the license agreements**' and press the **Finish** button. This will begin installing the software.

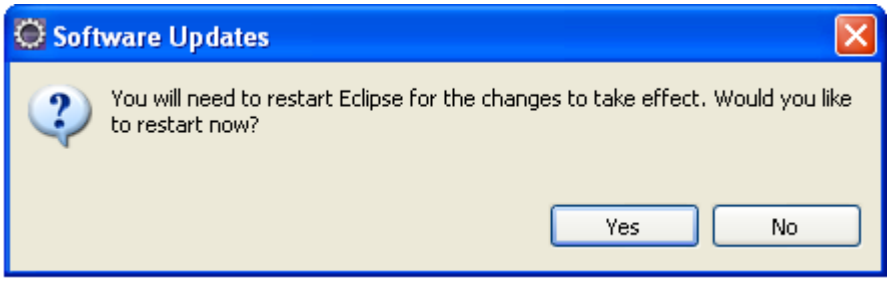


__13. Leave the dialog box that appears running so that you can see when the installation process completes.

__14. When you get a prompt about unsigned plugins click the **OK** button.

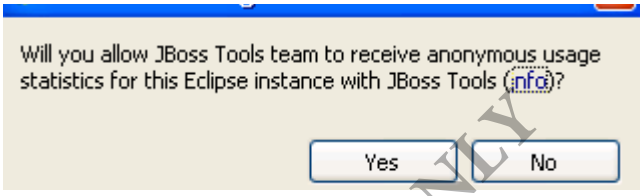


__15. Once the installation is complete you will get a prompt to restart Eclipse. Press the **Yes** button to restart.



__16. Open Eclipse in the same workspace when it restarts.

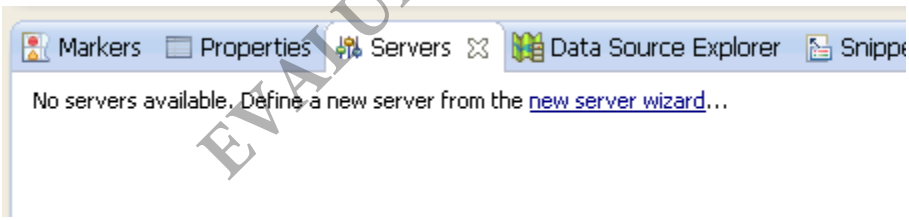
__17. Click **No** if the JBoss Tools Usage dialog opens.



Part 3 - Configure JBoss Test Server in Eclipse

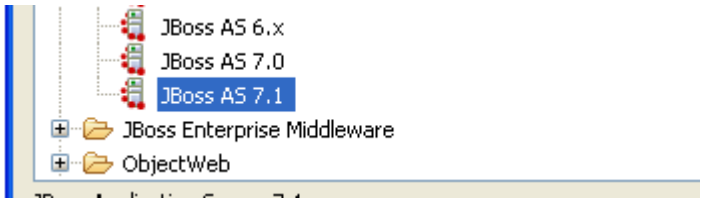
Now that the JBoss Eclipse tools are installed the next step is to define a JBoss test server. This will let you deploy and test applications directly from Eclipse.

__1. Click on the **Servers** view located in the bottom of the Eclipse window.



__2. Right click in the empty area and select **New -> Server**.

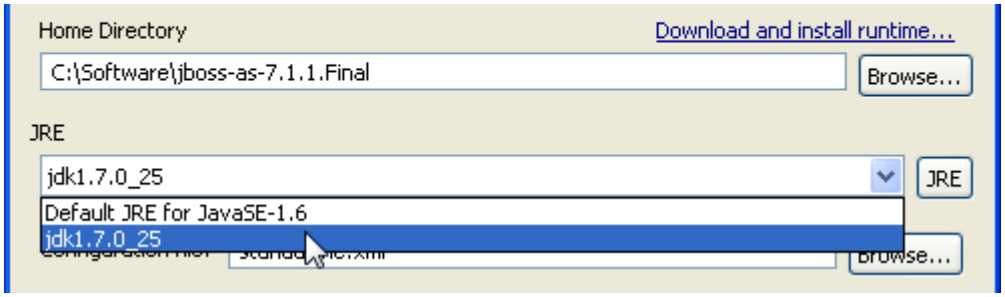
__3. From the list expand **JBoss Community** and select **JBoss AS 7.1**



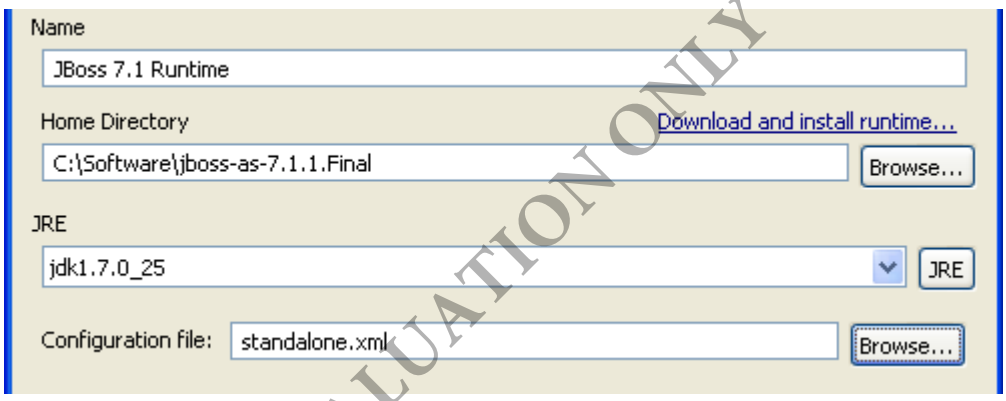
__4. Click **Next**.

__5. Click the first **Browse** button next to *Home Directory* and navigate to **C:\Software\jboss-as-7.1.1.Final** and then click **OK**. Stay on the same page in the main wizard.

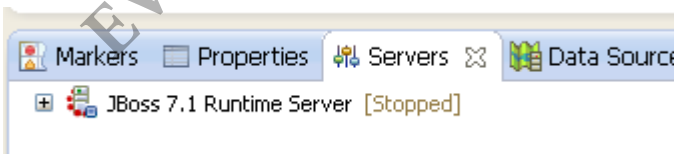
__6. For the **JRE** option use the drop-down to select the **JDK 1.7** that is installed.



__7. Click **Finish** once your dialog matches the settings shown below.

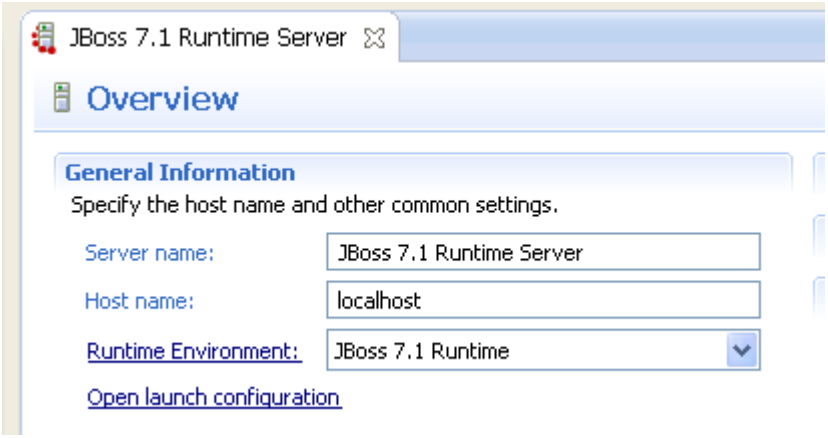


__8. The new server will appear in the Servers list.



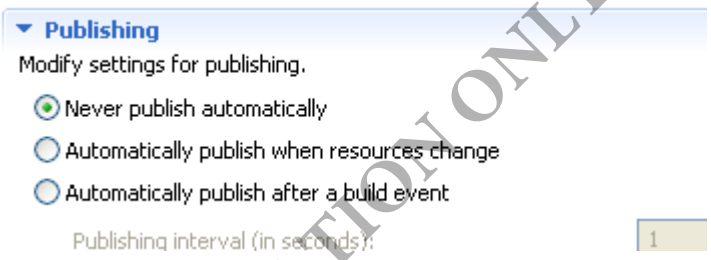
__9. Right click on the Server and select **Open**.

__10. The configuration page of the Server will open.

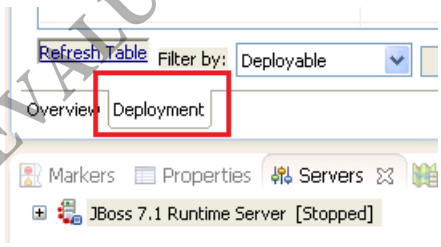


__11. Expand **Publishing** on the right side.

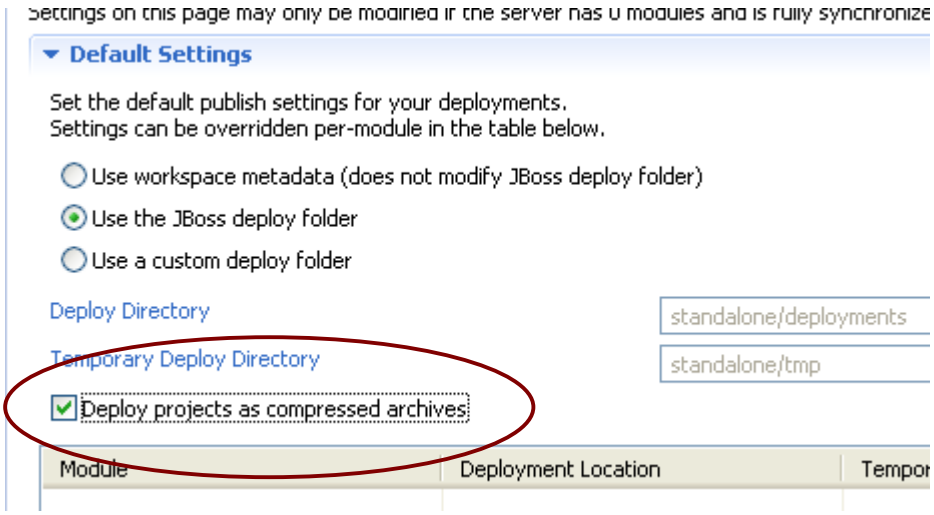
__12. Select **Never publish automatically**.



__13. At the bottom of the Eclipse server configuration editor click the '**Deployment**' tab.



__14. Check the '**Deploy projects as compressed archives**' option.



Note: This option is generally best to allow rapid deployment and testing of changes made in Eclipse. The default settings would deploy an "exploded" archive. Even though modifying and publishing an updated Java class would recompile the .class file copied to the server, the application would not be restarted and the previous version of the class loaded by the server would still be used.

By deploying a compressed archive, any change to any file in that project would recreate the entire archive and cause the entire application to be redeployed. This would guarantee the most recent code is being used. Since JBoss deployment is very quick, this method is preferred to avoid any confusing behavior where code changes have not been picked up by the server.

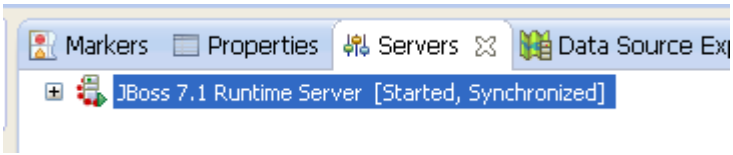
__15. Save the changes and close the window.

__16. In the **Servers** view, right click the '**JBoss 7.1 Runtime Server**' and select **Start**.

__17. Click the tab for the **Console** view and scroll back to find the message about JBoss Web Services starting. Notice that it is using the "CXF Stack" from Apache CXF. This will be important in advanced web service features in later labs.

```
ıg Naming Service
  Bound mail session [java:jboss/mail/Default]
  Starting Coyote HTTP/1.1 on http-localhost-127.0.0.1-8080
: thread 1-1) JBoss Web Services - Stack CXF Server 4.0.2.GA
ening on /127.0.0.1:9999
 JBAS015012: Started FileSystemDeploymentService for directory C:\
ening on localhost/127.0.0.1:4447
```

__18. Check that the server starts successfully. You may need to switch back to the **Servers** view.



__19. Leave the server running for the next section.

Part 4 - Configure JBoss Management User

By default the web administration console of JBoss is secure but doesn't have any users configured. This effectively locks it down until you configure users to access it. Since several labs may need to go into this administrative console to perform various tasks, this section will enable an administrative user.

__1. Open a Windows command prompt. You can usually do this on most versions of Windows by selecting '**Start → Programs → Accessories → Command Prompt**'.

__2. Use the following 'cd' command to switch to the JBoss 'bin' directory.

```
cd C:\Software\jboss-as-7.1.1.Final\bin
```

```
C:\Documents and Settings\Student>cd C:\Software\jboss-as-7.1.1.Final\bin

C:\Software\jboss-as-7.1.1.Final\bin>
```


__3. Run the following '**add-user**' command to start the interactive tool to add a user. How to respond to various prompts is given in the next steps.

```
add-user
```

```
C:\Software\jboss-as-7.1.1.Final\bin>add-user

What type of user do you wish to add?
```

__4. On the first prompt, hit the **<ENTER>** key to accept the default option of 'a' to create a "management user".

```
What type of user do you wish to add?
 a) Management User (mgmt-users.properties)
 b) Application User (application-users.properties)
(a):
```

__5. On the second prompt, hit the **<ENTER>** key to accept the default option of 'ManagementRealm' for the security realm. This is the name of the security realm already defined in the JBoss server so using this ensures we do not have to change any other JBoss configuration.

```
(a):

Enter the details of the new user to add.
Realm (ManagementRealm) : _
```

__6. Enter a username of '**admin**'.

```
Enter the details of the new user to add.
Realm (ManagementRealm) :
Username : admin
Password : _
```

__7. Type the following password for both prompts that ask you to enter a password. Note that the program will not even show stars while you type.

jbo$$ee6

```
Realm (ManagementRealm) :
Username : admin
Password :
Re-enter Password :
```

Note: This version of the tool requires a password that is at least 8 characters and has one number and one non-alphanumeric character.

__8. Type '**yes**' to accept the username 'admin' since it is "easy" to guess.

```
Re-enter Password :
The username 'admin' is easy to guess
Are you sure you want to add user 'admin' yes/no? yes
About to add user 'admin' for realm 'ManagementRealm'
```

__9. Type '**yes**' to confirm you want to add the 'admin' user to the 'ManagementRealm'.

```
Password :
Re-enter Password :
About to add user 'admin' for realm 'ManagementRealm'
Is this correct yes/no? yes
```

__10. Hit any key to let the program finish and return you to the command prompt.

```
About to add user 'admin' for realm 'ManagementRealm'
Is this correct yes/no? yes
Added user 'admin' to file 'C:\Software\jboss-as-7.1.1.Final\standalone\configur
ation\mgmt-users.properties'
Added user 'admin' to file 'C:\Software\jboss-as-7.1.1.Final\domain\configuratio
n\mgmt-users.properties'
Press any key to continue . . . _
```

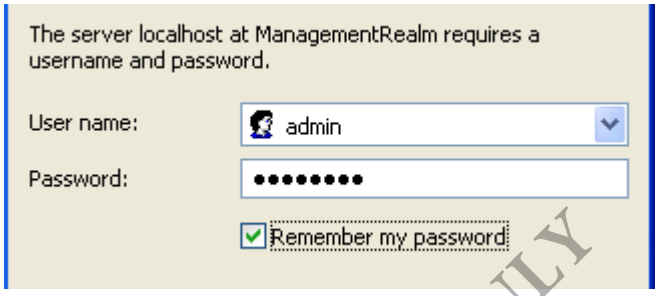
__11. Close the command prompt.

__12. Go back to the **Servers** view in Eclipse and make sure your JBoss server is running. Start it if it is not.

__13. Open a web browser to the following address which should bring up a security prompt.

```
http://localhost:9990/console
```

__14. In the security prompt, enter a username of '**admin**' and a password of '**jbo$$ee6**'. You can check the option to remember the password if offered. Click **OK**.



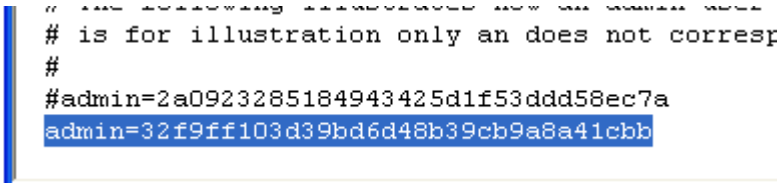
__15. Check that you see the overview page of the administration console. If it doesn't display try refreshing the page.

Note: If you get an authentication error it may be because somehow the password you entered in the browser is different than what you configured in the script tool. If you can't provide the correct password you can run the 'add-user' tool again but you first have to manually delete some lines from two text files.

Find the following two text files, open them with a text editor like WordPad, and delete the line that starts with 'admin=...' without a leading '#' as a comment.

C:\Software\jboss-as-7.1.1.Final\domain\configuration\mgmt-users.properties

C:\Software\jboss-as-7.1.1.Final\standalone\configuration\mgmt-users.properties



You need to delete this line from both files because even though your configuration is only looking at the 'standalone' properties file, you can't run the 'add-user' tool again with the same user if it already exists in either one of these files.

__16. Close the browser you are using to access the administration console.

__17. Stop the server in the Eclipse **Servers** view.

Part 5 - Review

In this lab you installed JBoss Eclipse plug-ins for the rest of the labs. You could do these steps in your own development environment after downloading the proper JBoss Tools downloads. You also did several configuration steps that will let you work with JBoss from Eclipse and access the administration console if required. You also saw how to install a different version of JBoss Web Services in a JBoss server. Although we just installed the Spring modules for later and used the same JBossWS-CXF version, you could also download a different version in your own environment.

Lab 2 - Develop a Simple RESTful Service

Over the next few labs, our goal will be to learn the fundamentals of the JAX-RS API.

In this lab, we will develop a very simple RESTful web service. This will let us focus on the development process using eclipse and the JBoss server. We will also learn about the basic JAX-RS annotations.

The main goal of this lab is to understand how to create a web service project and develop a few basic REST services.

Part 1 - Create the Web Service Project

In eclipse, the project to use to develop JAX-RS services is just a regular "Dynamic Web" project. When you target a server, the JAR files for the JAX-RS API are already on the classpath.

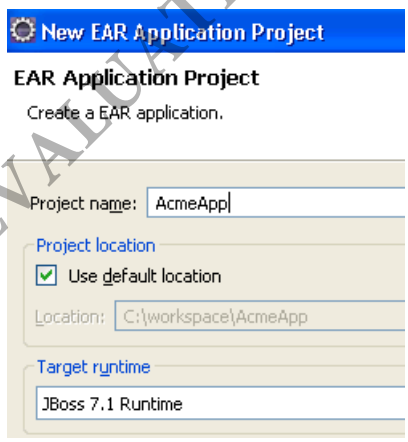
__1. From the menubar, select **File > New > Dynamic Web Project**.

__2. Enter **AcmeWeb** for the *Project name*.

__3. At the bottom of the window, in the *EAR Membership* section, click the button marked **New Project...**. (We will want to add this WAR file into an EAR file)

The *New EAR Application Project* screen will appear.

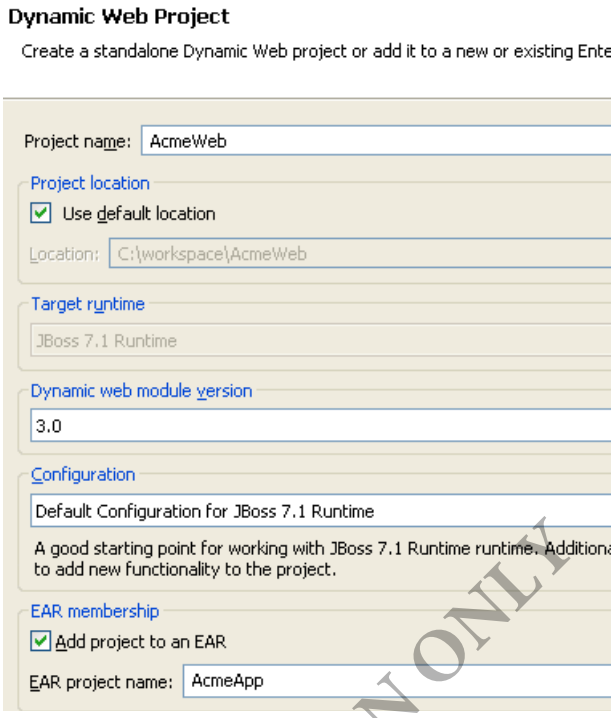
__4. Set the *Project name* to **AcmeApp** and make sure the target server is JBoss.



__5. Click **Finish** to create the EAR project

You will be returned to the *New Dynamic Web Project* screen.

Your screen should now look like the following:



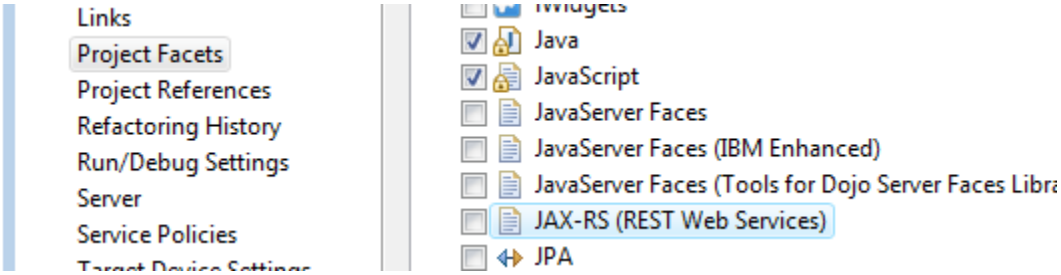
__6. Click **Finish**. Both the **EAR** and the **WAR** will be created. If prompted, don't switch to the web perspective by clicking **No**.

We will briefly review the project.

__7. Right click the newly created **AcmeWeb** project and select **Properties**.

__8. Now, select the **Project Facets** property.

__9. Note that the JAX-RS facet is not selected. This facet plays no role in developing JAX-RS services.



__10. Click **Cancel** to close the dialog.

Part 2 - Register the REST Application

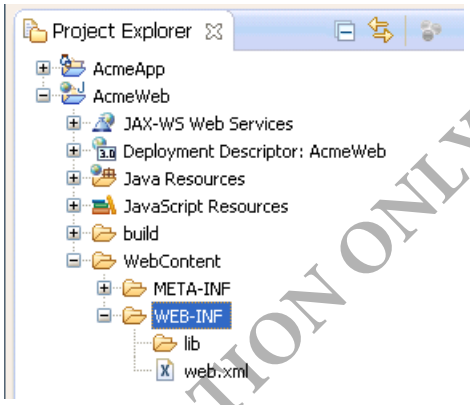
There are a few different ways to configure a REST application. This can depend on if you want a common URL prefix for all REST services. This might be useful if you have REST services within a project with other web components. If you want to register a common URL prefix you can do so in a web.xml file. To save time, this file is given to you. You will simply import it.

__1. Open Windows file explorer.

__2. Go to **C:\LabFiles**

__3. Copy **web.xml**

__4. In eclipse, paste the file inside the **WebContent > WEB-INF** folder of **AcmeWeb**.



__5. Open **web.xml** and study how the REST application is configured. Switch to the Source view and specifically, note the servlet mapping:

```
<servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>/svc/*</url-pattern>
</servlet-mapping>
```

This means, the URL for every REST request will start with http://host:port/AcmeWeb/svc/. You can choose any other path for the REST application. But, we will stick to the short and sweet "svc".

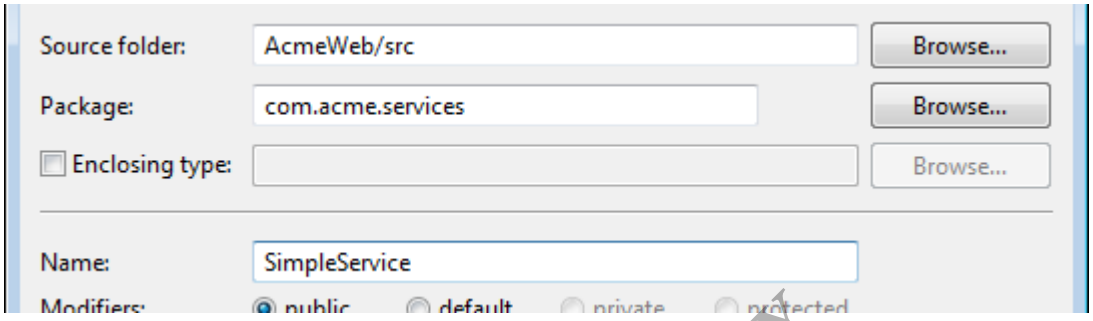
__6. Close the file.

Part 3 - Create the Resource Class

The Java class that implements a RESTful service is called a resource. We will now develop a Java class for a simple service.

__1. Right click **AcmeWeb** project and select **New > Class**.

__2. Enter **com.acme.services** as the package name and **SimpleService** as the class name.



__3. Click **Finish** to create the new class.

__4. Add a member variable that will help us do logging.

```
public class SimpleService {
          Logger logger = Logger.getLogger("SimpleService");
```

__5. Organize imports (Control+Shift+O) and select **java.util.logging.Logger**. Make sure you select the correct class as there are several 'Logger' classes on the classpath.

We will now add the testGET() method. It will not have any business logic. Later, we will map this method to a GET request.

__6. Add the method as follows.

```
public String testGET() {
          logger.info("Got a GET request");

          return "OK";
}
```

__7. Save changes.

Part 4 - Configure the Resource

We will now configure the URI and HTTP method for the service resource and its methods. We will use the "/simple" root URI for the service.

__1. Define the URI of the root resource, by adding the @Path annotation above the class.

```
@Path("/simple")
public class SimpleService {
```

The testGET() method does not need any path extension. All we have to do is set GET as the HTTP method.

We will also set "text/plain" as the content MIME type of the reply. That is good enough for this method. For XML data type, the MIME will be "text/xml".

__2. Set the HTTP method and reply MIME type for the testGET() method sub-resource as follows.

```
@GET
@Produces("text/plain")
public String testGET() {
```

__3. Organize imports. Select **javax.ws.rs.Produces**.

__4. Save changes.

Part 5 - Unit Test

We will now exercise the web service from a browser.

__1. Start the server.

__2. Right click the server and select **Add and Remove**.

__3. Add the **AcmeApp** project to the server.

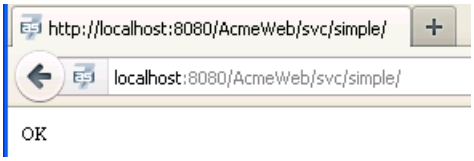
__4. Click **Finish**.

__5. Right click the server and select **Publish**. Do this every time you are requested to Publish the server.

__6. Open a web browser and enter the URL:

```
http://localhost:8080/AcmeWeb/svc/simple/
```

__7. You should see OK in the browser.



__8. The *Console* view will show the log output.

```
org.jboss.as.server.deployment] (MSC service thread 1-2) JBAS015876: Starting deployment of "
org.jboss.web] (MSC service thread 1-2) JBAS018210: Registering web context: /AcmeWeb
org.jboss.as.server] (DeploymentScanner-threads - 2) JBAS018559: Deployed "AcmeApp.ear"
SimpleService] (http-localhost-127.0.0.1-8080-1) Got a GET request
```

This proves that our project has been setup correctly and JAX-RS is working fine.

Part 6 - Use Path Extension for Sub-resource

A sub-resource – Java method – can be mapped to an URI extension path. Any HTTP request for that path will be handled by that method. We will now create a method that will respond to the /simple/**mypath** URI.

__1. First, add this method to the SimpleService class.

```
public String testGETWithPath() {
            logger.info("Got a GET request with path extension.");

            return "OK";
}
```

__2. Annotate the method as follows.

```
@GET
@Produces("text/plain")
@Path("/mypath")
public String testGETWithPath() {
```

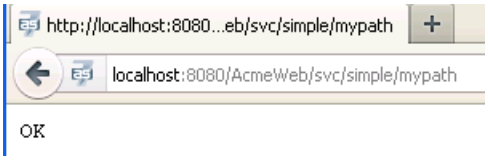
__3. Save changes.

Part 7 - Unit Test

__1. Right click the server and select **Publish**.

__2. In a web browser, enter the URL:

```
http://localhost:8080/AcmeWeb/svc/simple/mypath
```



__3. Make sure that the Console shows the log output.

```
:b] (MSC service thread 1-2) JBAS018210: Registering web context: /AcmeWeb
:.server] (DeploymentScanner-threads - 1) JBAS018565: Replaced deployment "AcmeApp.ear"
:e] (http-localhost-127.0.0.1-8080-1) Got a GET request with path extension.
```

__4. Open a new browser or tab and re-test the testGET method for regression using the URL:

```
http://localhost:8080/AcmeWeb/svc/simple/
```

__5. Close all browsers.

__6. Close all open files.

Part 8 - Review

In this lab, we created and configured a web service project. We created a very simple JAX-RS web service. At this point, you should know how to configure the URI of the root resource – the Java class - using the @Path annotation. Also, we configured the HTTP method of a sub-resource – a method – using the @GET annotation.

Lab 3 - Extracting Information from a HTTP Request

RESTful services are executed by sending HTTP requests. The request contains input data in various areas:

1. In the URI path. For example: /orders/**1051**.
2. As URL parameters. For example: /orders?**status=P**
3. Input data from a form submission.
4. Less commonly, from HTTP header and cookie.

We will now learn how to extract data from common locations.

Part 1 - Get Root Resource Path Parameters

Input data can be added to the path of the root resource as well as the path of a method (sub-resource). In REST, data is added to the URI to form an unique identifier.

First, we will add an input parameter in the root path of the SimpleService resource. The root URI will now look like /simple/**somedata**. The root URI will continue to execute the testGET() method since this method defines no path extension. To execute the testGETWithPath(), the URI will need to be /simple/**somedata**/mypath.

__1. Open **SimpleService.java** from the AcmeWeb project.

__2. Change the @Path annotation for the class and add a parameter there.

```
@Path("/simple/{myRootPathData}")
public class SimpleService {
```

Here, {myRootPathData} is a placeholder for a parameter. You can use anything as a name of the parameter. The name will play a role to obtain the value of the parameter.

The best place to capture input from a root path is a member variable of the resource. We will do that now.

__3. Add a member variable as follows.

```
public class SimpleService {
          String rootPathData;
```

__4. Annotate the member variable to save the value of the myRootPathData parameter.

```
@PathParam("myRootPathData")
String rootPathData;
```

That's it. Now, JAX-RS will extract the value of the parameter from the URI and set it to the member variable right after the resource object is created. By default, a POJO resource instance is created for every HTTP request. Hence, every request can have a different parameter value in the path.

__5. Organize imports.

__6. Change the log statement of the testGET() method as follows.

```
logger.info("Got a GET request with root path data: " + rootPathData);
```

__7. Make a similar change to the testGETWithPath() method.

```
logger.info(
    "Got a GET request with path extension with root path data: " +
    rootPathData);
```

__8. Save changes.

Part 2 - Unit Test

__1. Publish the server.

__2. First, test the testGET() method by entering the URL:

```
http://localhost:8080/AcmeWeb/svc/simple/somedata/
```

__3. The log output will like this:

```
Got a GET request with root path data: somedata
```

__4. Now, test the testGETWithPath() method using the URL:

```
http://localhost:8080/AcmeWeb/svc/simple/somedata/mypath
```

__5. The log output will be:

```
Got a GET request with path extension with root path data: somedata
```

Part 3 - Get Sub-resource Path Parameters

Methods can also let us add parameters to its path. For example to get the billing address of an order #1051, we can use the URI: /orders/**1051**/address/**billing**. Never lose sight of the fact that a URI uniquely identifies an entity or a collection of entities. In this example, we are specifically pointing to the billing address used with an order. Here, the orderId as well as the type of address requested can be added as parameters to the path of the sub-resource. Let's try out this example.

__1. Add a basic method as follows.

```
public String getAddress(
          int oId,
          String type) {
          logger.info("Order Id: " + oId);
          logger.info("Address type: " + type);

          return "OK";
}
```

__2. Annotate method with path and method.

```
@GET
@Produces("text/plain")
@Path("/{orderId}/address/{addressType}")
public String getAddress(
```

__3. Now, extract the path parameters and save them in the two input argument variables of the method. The syntax for the @PathParam annotation on a method parameter is tricky so be careful.

```
public String getAddress(
          @PathParam("orderId")
          int oId,
          @PathParam("addressType")
          String type) {
```

__4. Save changes.

__5. Publish the server.

__6. Test the method by entering the URL:

```
http://localhost:8080/AcmeWeb/svc/simple/somedata/1051/address/billing
```

```
ds - 2) JBAS018565: Replaced deploy
-1) Order Id: 1051
-1) Address type: billing
```

Part 4 - Extract Query Parameters

Query parameters are available from the requested URI as:

```
GET /path?param1=value&param2=value2 HTTP/1.1
```

You can capture them either using resource class member variable or method argument variable. The latter is generally recommended for better code readability. We will try that out now.

__1. Add the giveRaise() method as shown below.

```
@GET
@Produces("text/plain")
@Path("raise")
public String giveRaise(
    @QueryParam("name")
    String employeeName,
    @QueryParam("amount")
    double amount
    ) {
    logger.info("Giving raise to " + employeeName + " by " + amount);

    return "OK";
}
```

__2. Organize imports.

__3. Save changes.

__4. Publish the server.

__5. Test the change by entering the URL:

```
http://localhost:8080/AcmeWeb/svc/simple/somedata/raise?name=Daffy&amount=10000
```

__6. Make sure that the Console shows:

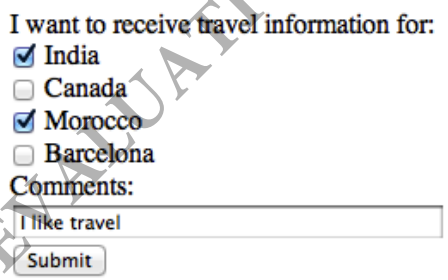
```
Giving raise to Daffy by 10000.0
```

Note, how JAX-RS converted the amount URL parameter from text to double.

Part 5 - Extracting Form Post Data

Form data is available in the HTTP request body. The data is encoded using MIME type application/x-www-form-urlencoded. Usually, such data is submitted using the POST method.

Note: The Java EE Servlet specification does not distinguish between URL query parameter and POST data in HTTP request body. They are both accessed using request.getParameter() method. JAX-RS, however, makes a distinction. Query parameter can be accessed using @QueryParam. Form POST data has to be accessed using @FormParam.



We will now build a service method that will accept input from the form above.

__1. Add this method.

```
@POST
@Produces("text/plain")
@Path("feedback")
public String submitFeed(
        @FormParam("interest")
        List<String> interestList,
        @FormParam("comments")
        String comments
    ) {
    for (String interest : interestList) {
        logger.info("Interest: " + interest);
    }
    logger.info("Comments: " + comments);

    return "OK";
}
```

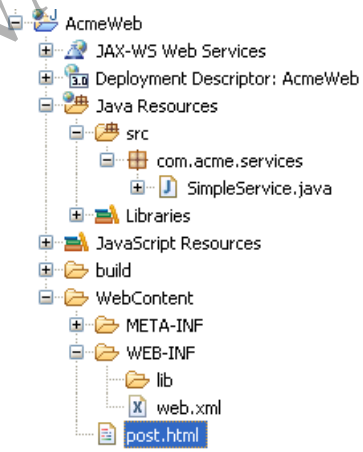
Note that the HTTP method is set to POST. Also, we expect multiple values for the "interest" parameter. That is why we have set the data type of the variable to java.util.List<String>.

__2. Organize imports. Select **java.util.List**.

__3. Save changes.

Part 6 - Unit Test

__1. We will import the form HTML page. Copy **C:\LabFiles\post.html** and paste it inside the **WebContent** folder of the **AcmeWeb** project.



__2. In the *Servers* view, right click the server and select **Publish**.

__3. Open a browser and enter the URL:

```
http://localhost:8080/AcmeWeb/post.html
```

__4. Fill out the form and submit it.

__5. Make sure that the console log shows the input values correctly.

```
reads - 1) JBAS018565: Replac
080-1) Interest: Canada
080-1) Comments: Blue Jays
```

__6. Close all open files.

__7. Close all open browsers.

Part 7 - Review

In this lab, we learned how to gather input data from a couple of common sources:

1. URI path of the root resource
2. URI path of the method sub-resource.
3. URL query parameter.
4. Form POST data.