

**WA2090 Programming REST Web
Services with JAX-RS 1.1 -
WebSphere 8.0 / RAD 8.0**

Student Labs

Web Age Solutions Inc.

EVALUATION ONLY

Table of Contents

Lab 1 - WebSphere Workspace Configuration.....	3
Lab 2 - Develop a Simple RESTful Service.....	9
Lab 3 - Extracting Information from a HTTP Request.....	16
Lab 4 - Designing a RESTful Service.....	23
Lab 5 - Add Support for XML Format.....	30
Lab 6 - Add Support for JSON.....	38
Lab 7 - Build the Order Web Service.....	44
Lab 8 - Complete the Order Service.....	54
Lab 9 - Developing a JAX-RS Client.....	64
Lab 10 - Securing RESTful Services.....	71

EVALUATION ONLY

Lab 1 - WebSphere Workspace Configuration

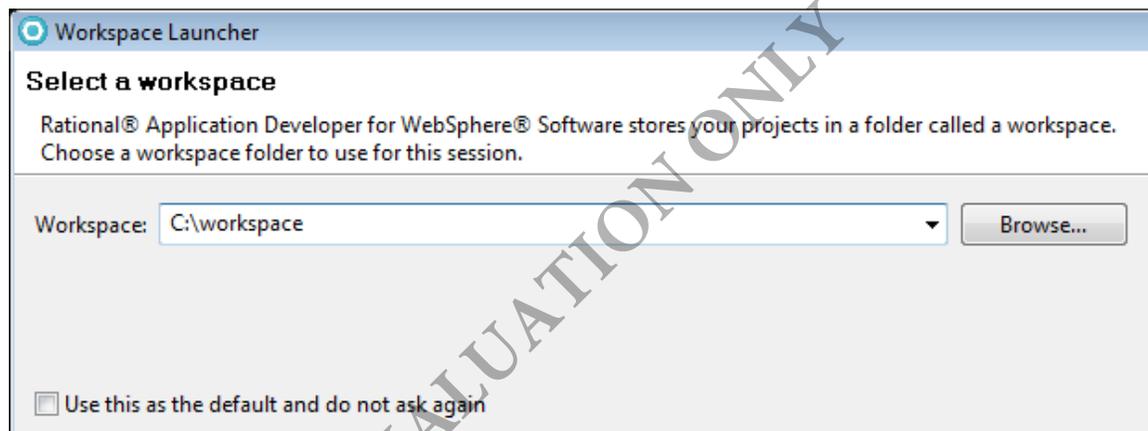
In this lab you will configure the RAD workspace to work with a WebSphere test environment server. This will include creating a WebSphere "profile" which will have the server definition. It is sometimes useful to have different profile configurations for testing purposes so this process is good to be familiar with.

Part 1 - Create Profile

The WebSphere Application Server software is installed along with RAD. In this section you will run a standard WebSphere tool, the Profile Management Tool, to create a profile.

__1. From the Windows Start menu select **Start** → **Programs** → **IBM Software Delivery Platform** → **IBM Rational Application Developer 8.0** → **Rational Application Developer**.

__2. In the workspace window, change the **Workspace** to **C:\workspace**

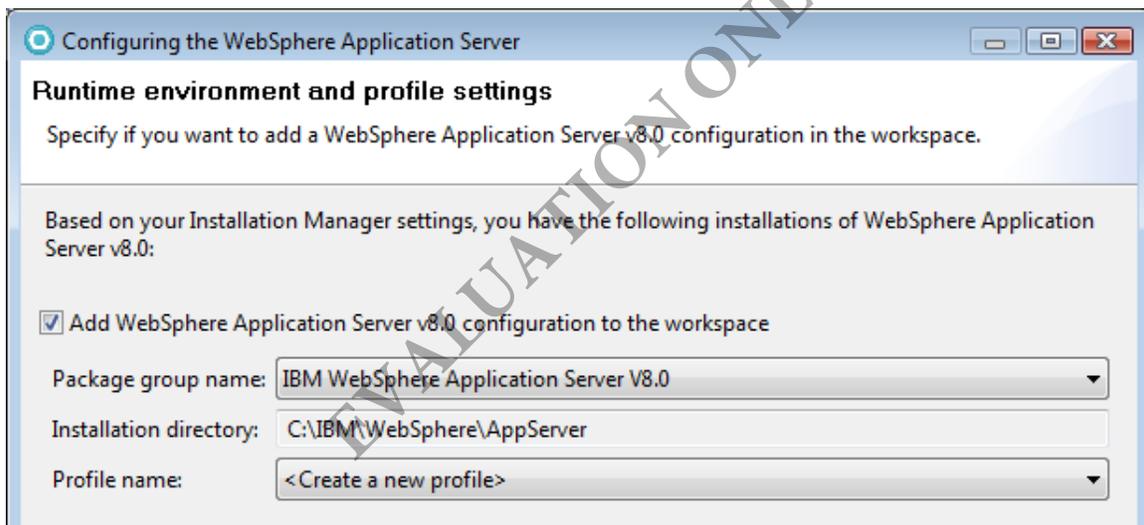


__3. Click **OK**.

RAD 8.0.4 will start.



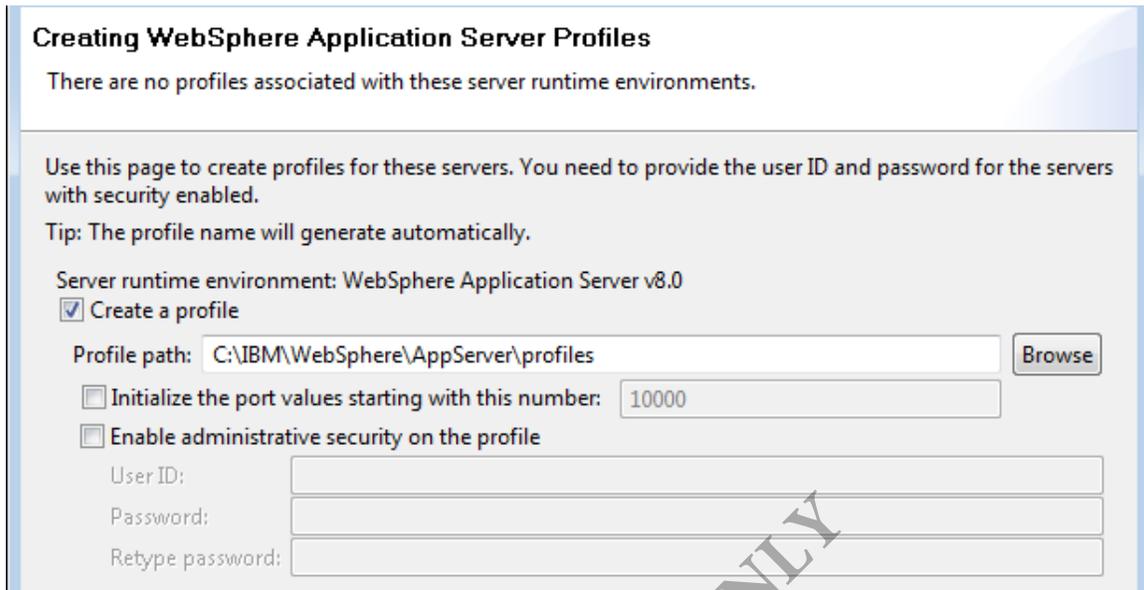
4. In the *Runtime environment and profile settings* dialog that appears check that the option to '<Create a new profile>' is available and click the **Next** button.



Note: If you **DO NOT** get this prompt it is possible that a profile was created for this workspace already. Since it is always best for the user who will be using the workspace to create the profile restart RAD and use a different workspace with RAD. Inform your instructor of this to see if it is a common problem in the class.

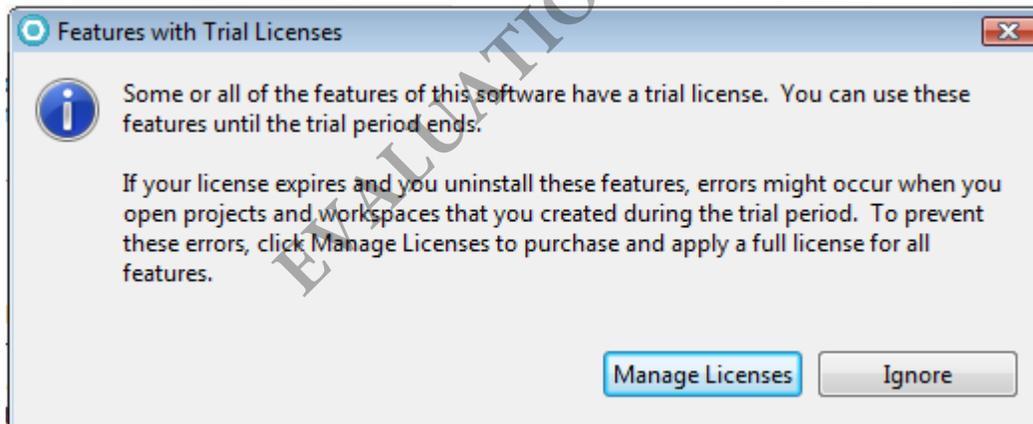
If there is an existing profile listed in the dialog a previous profile may already exist. Let your instructor know about this because although this profile can likely be used with your workspace, if it has applications deployed to it from another workspace, there may be errors when starting the server. It is also possible to run the 'Profile Management Tool' to create a new profile but this will be complex and should be guided by the instructor.

5. Uncheck the 'enable administrative security on the profile' option as shown below and click the **Finish** button.



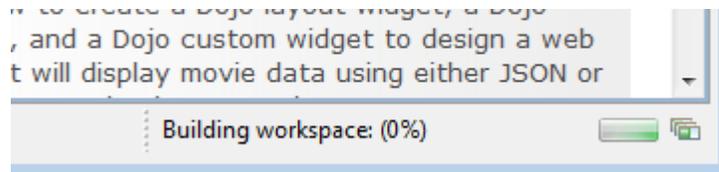
The **Features with Trial Licenses** dialog will open.

6. Click **Ignore**.

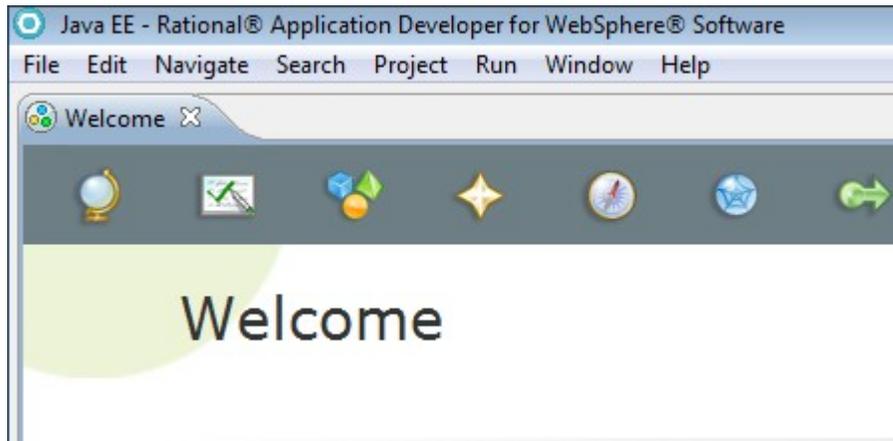


7. If prompted about help content select **Work without Help**.

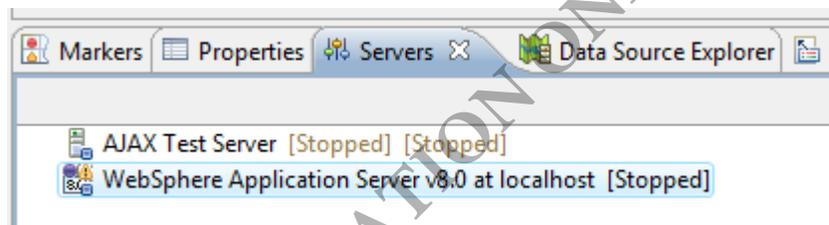
8. Wait until the messages in the lower right corner of RAD disappear to know the work of creating the server is complete.



- __9. If prompted by Windows firewall security hit the **Cancel** button.
- __10. Close the **Welcome** page if it appears.



- __11. Select the **Servers** view on the bottom of the *Java EE* perspective.
- __12. Check that a WebSphere Application Server v8.0 server is shown.



- __13. Open the following file:

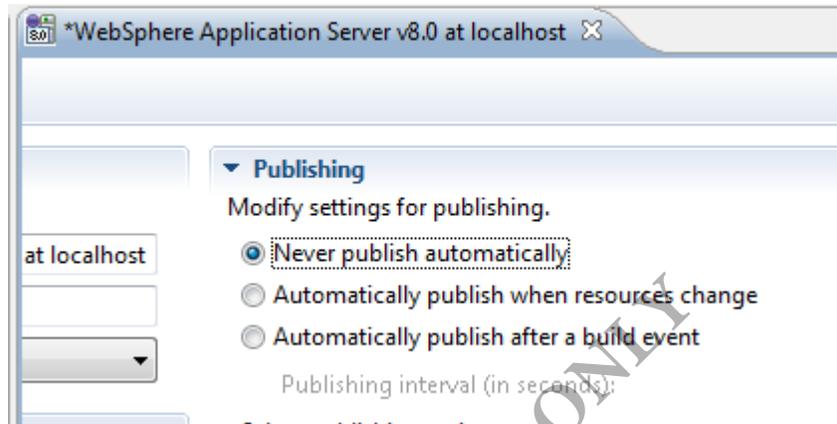
C:\IBM\WebSphere\AppServer\logs\manageprofiles\AppSrv1_create.log

- __14. Look at the end of the file for the message:

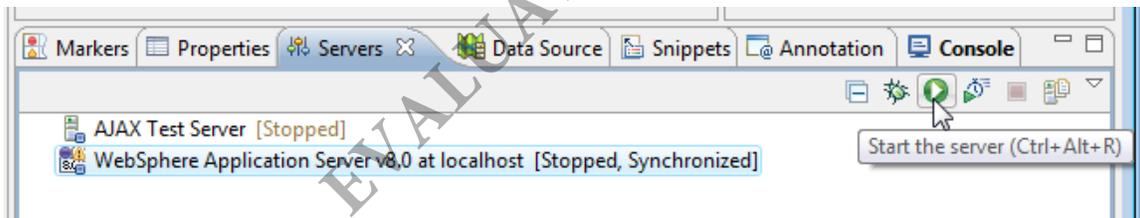
```
</record>
<record>
  <date>2012-07-08T04:28:29</date>
  <millis>1341779309892</millis>
  <sequence>4676</sequence>
  <logger>com.ibm.wsspi.profile.wsProfileCLI</logger>
  <level>INFO</level>
  <class>com.ibm.wsspi.profile.wsProfileCLI</class>
  <method>invokeWSProfile</method>
  <thread>0</thread>
  <message>Returning with return code: INSTCONFSUCCESS</message>
</record>
</log>
```

If you see the message INSTCONFSUCCESS, it means that the WebSphere Application Server profile was created correctly.

- __15. Close the log file.
- __16. Switch back to RAD and the **Servers** view.
- __17. Make sure the server is stopped.
- __18. Double click on the server to open the server properties.
- __19. On the right, expand **Publishing** and select the 'Never publish automatically' option.

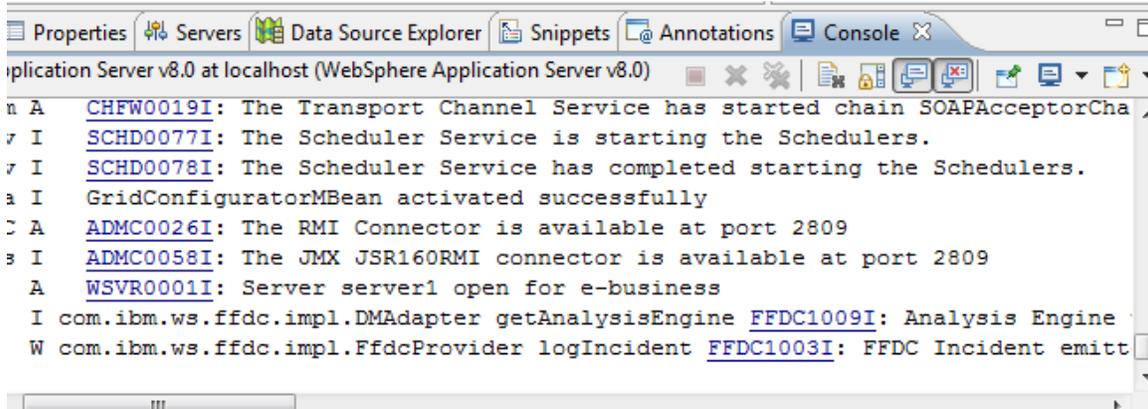


- __20. Save and close the server properties.
- __21. Select the **WebSphere Application Server v8.0** server that was created in the *Servers* view and click the **Start the server** button on the right.

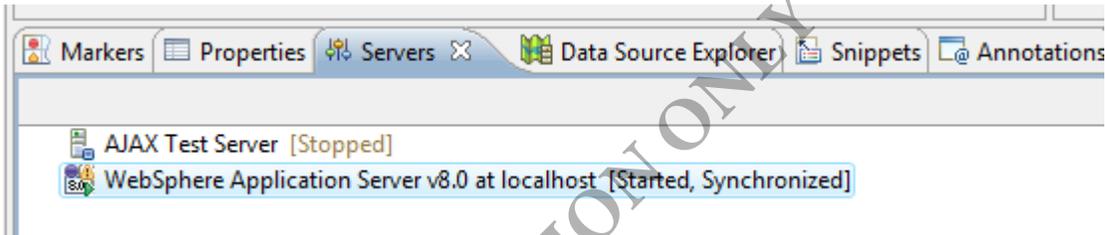


__22. From the menu, select **Window > Show View > Console**.

__23. Check the *Console* view as the server starts to look for errors.

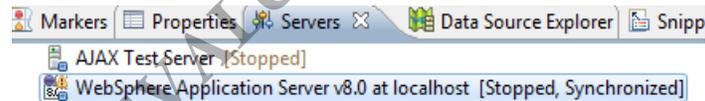


__24. Switch back to the *Servers* view and check that the server was started.



__25. Right click the server and select **Stop**.

__26. Check that the server was Stopped.



Part 2 - Review

You have now configured a test server to use with the RAD workspace.

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 RAD and the WebSphere 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.

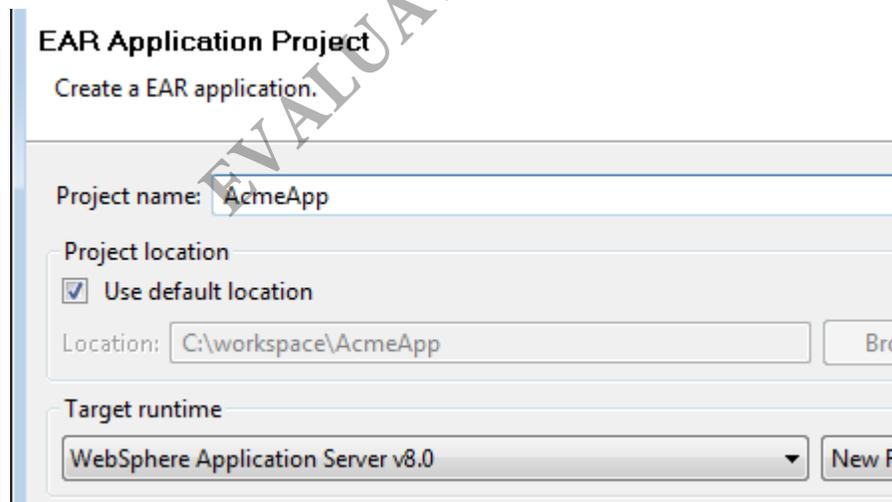
Part 1 - Create the Web Service Project

In RAD, the project to use to develop JAX-RS services is just a regular “Dynamic Web” project. When you target a WebSphere 8.0 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.

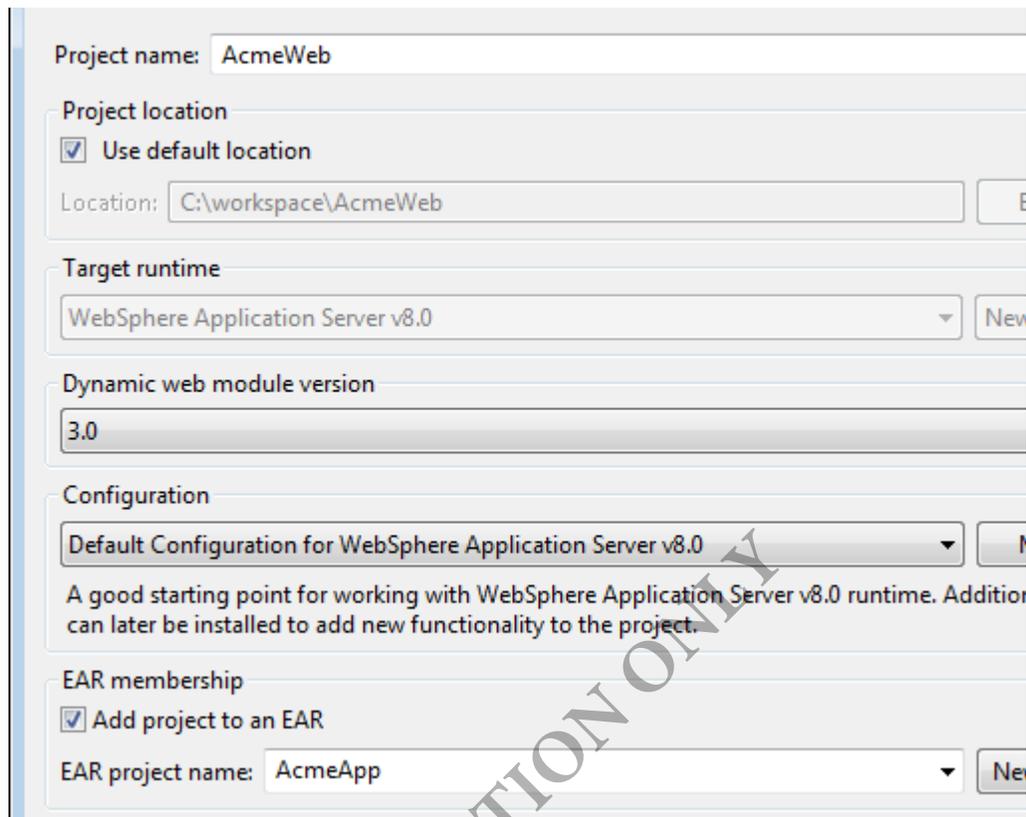
4. Set the *Project name* to **AcmeApp** and make sure the target server is WebSphere 8.0.



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**.

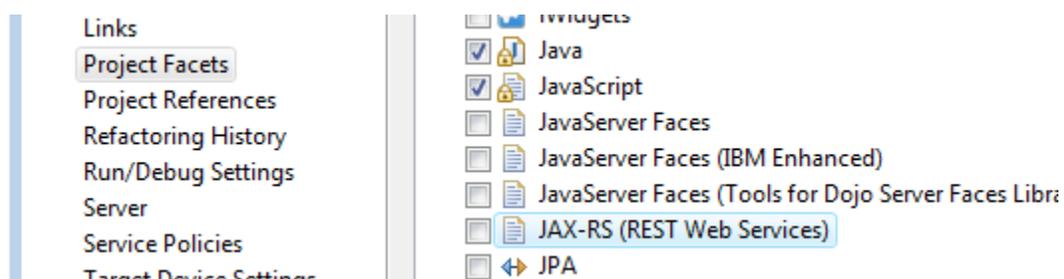
__7. Close the 'Technology Quickstarts' if it opens.

We will briefly review the project.

__8. Right click the newly created **AcmeWeb** project and select **Properties**.

__9. Now, select the **Project Facets** property.

__10. Note that the JAX-RS facet is not selected. This facet plays no role in developing JAX-RS services for WebSphere 8.0.

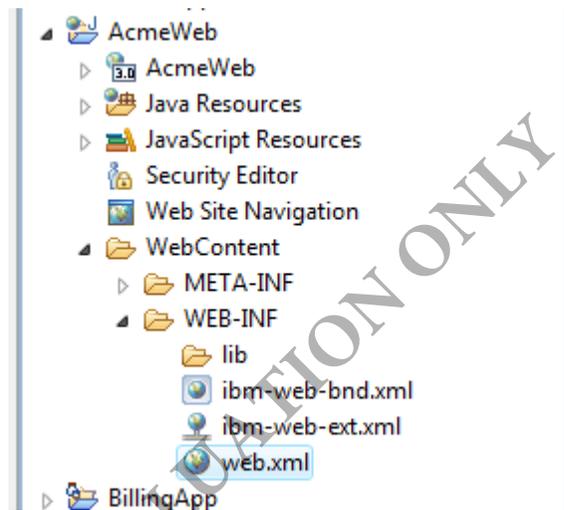


__11. Click **Cancel** to close the dialog.

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 RAD, 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".

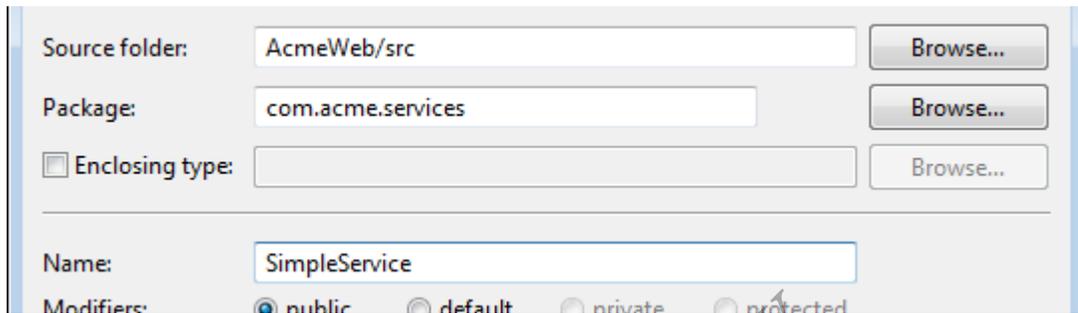
- __6. Close the file.

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.



The screenshot shows the 'New Class' dialog box. The 'Source folder' field contains 'AcmeWeb/src'. The 'Package' field contains 'com.acme.services'. The 'Enclosing type' checkbox is unchecked. The 'Name' field contains 'SimpleService'. The 'Modifiers' section shows 'public' selected, 'default' selected, 'private' unselected, and 'protected' unselected.

__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 WebSphere server.

__ 2. Right click the WebSphere server and select **Add and Remove**.

__ 3. Add the **AcmeApp** project to the server.

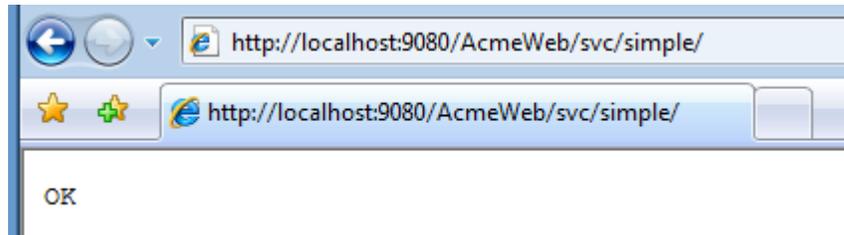
__ 4. Click **Finish**.

__ 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:9080/AcmeWeb/svc/simple/`

__7. You should see OK in the browser.



__8. The Console view in RAD will show the log output.

```
36 Providers      I org.apache.wink.server.internal.lo
36 servlet        I com.ibm.ws.webcontainer.servlet.Se
36 SimpleService I   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.

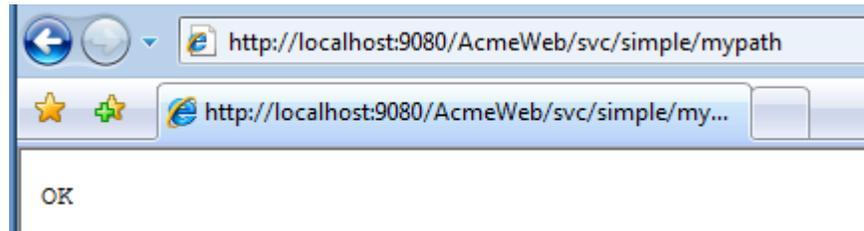
```
@GET
@Produces("text/plain")
@Path("/mypath")
public String testGETWithPath() {
```

__3. Save changes.

Part 7 - Unit Test

- __ 1. Right click the WebSphere server and select **Publish**.
- __ 2. In a web browser, enter the URL:

`http://localhost:9080/AcmeWeb/svc/simple/mypath`



- __ 3. Make sure that the Console shows the log output.

```
7 Providers      I org.apache.wink.server.internal.log.Providers log
7 servlet        I com.ibm.ws.webcontainer.servlet.ServletWrapper in
7 SimpleService I   Got a GET request with path extension.
```

- __ 4. Open a new browser and re-test the testGET method for regression using the URL:

`http://localhost:9080/AcmeWeb/svc/simple/`

- __ 5. Close all open files.
- __ 6. Close all browsers.

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.