WA2002 Mastering JAX-RS REST Web Services and AJAX Clients - Websphere 8.0 / RAD 8.0

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - WebSphere Workspace Configuration

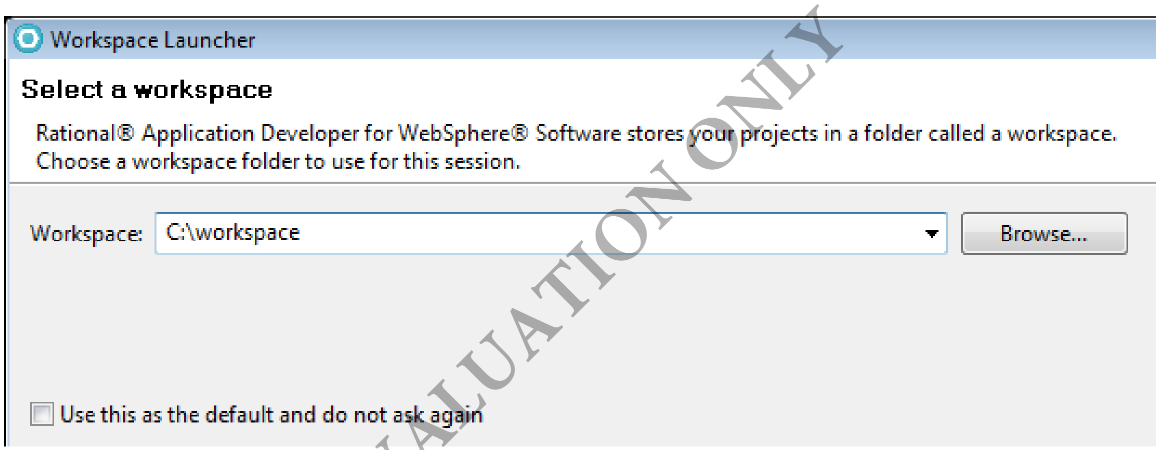
In this lab you will configure the RAD workspace to work with a WebSphere test environment server. This will include creating a WebSphere "profile" which will have the server definition. It is sometimes useful to have different profile configurations for testing purposes so this process is good to be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



Select a workspace

Rational® Application Developer for WebSphere® Software stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\workspace

Use this as the default and do not ask again

__3. Click **OK**.

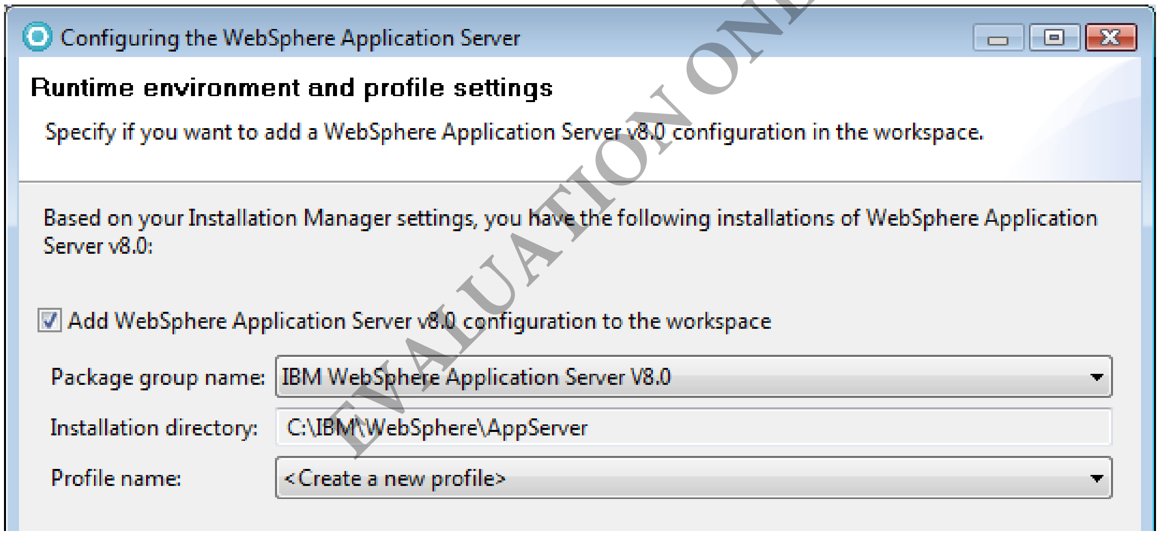
RAD 8.0.4 will start.

Rational. Application Developer for WebSphere® Software
Version 8.0.4

Licensed Materials - Property of IBM Corp. © IBM Corporation and other(s) 2000, 2011. IBM, the IBM logo, Rational and WebSphere are trademarks of IBM Corp. in the United States, other countries or both. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates. Other company, product or service names may be trademarks or service marks of others.

Loading Workbench

__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.


Configuring the WebSphere Application Server

Runtime environment and profile settings

Specify if you want to add a WebSphere Application Server v8.0 configuration in the workspace.

Based on your Installation Manager settings, you have the following installations of WebSphere Application Server v8.0:

☑ Add WebSphere Application Server v8.0 configuration to the workspace

Package group name: IBM WebSphere Application Server V8.0

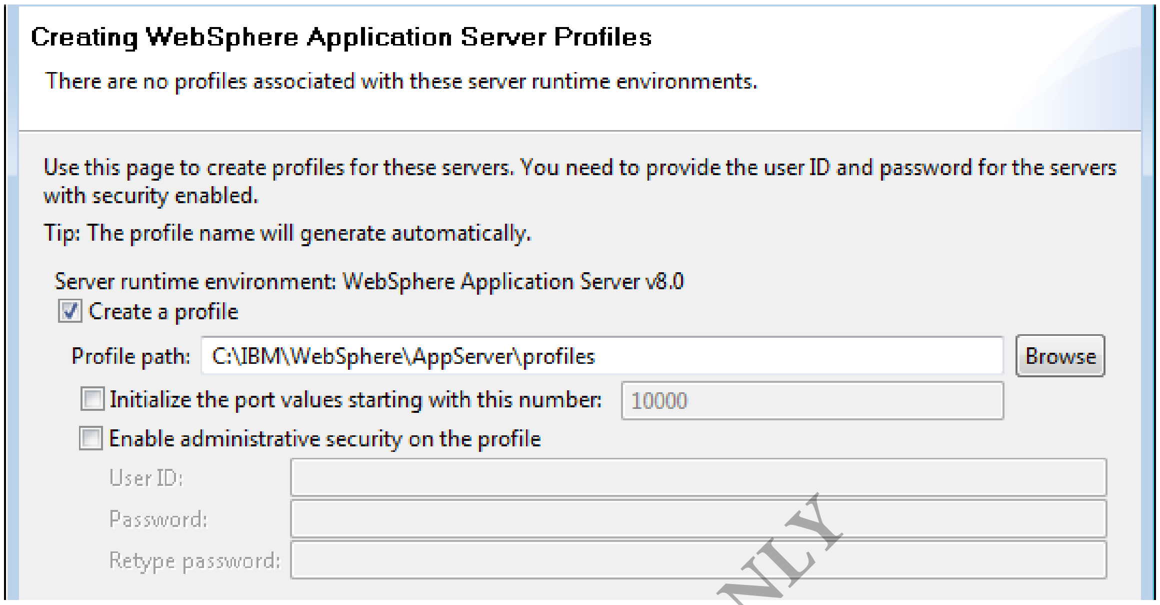
Installation directory: C:\IBM\WebSphere\AppServer

Profile name: <Create a new profile>

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.


Creating WebSphere Application Server Profiles

There are no profiles associated with these server runtime environments.

Use this page to create profiles for these servers. You need to provide the user ID and password for the servers with security enabled.

Tip: The profile name will generate automatically.

Server runtime environment: WebSphere Application Server v8.0

[✓] Create a profile

Profile path: C:\IBM\WebSphere\AppServer\profiles

[ ] Initialize the port values starting with this number: 10000

[ ] Enable administrative security on the profile

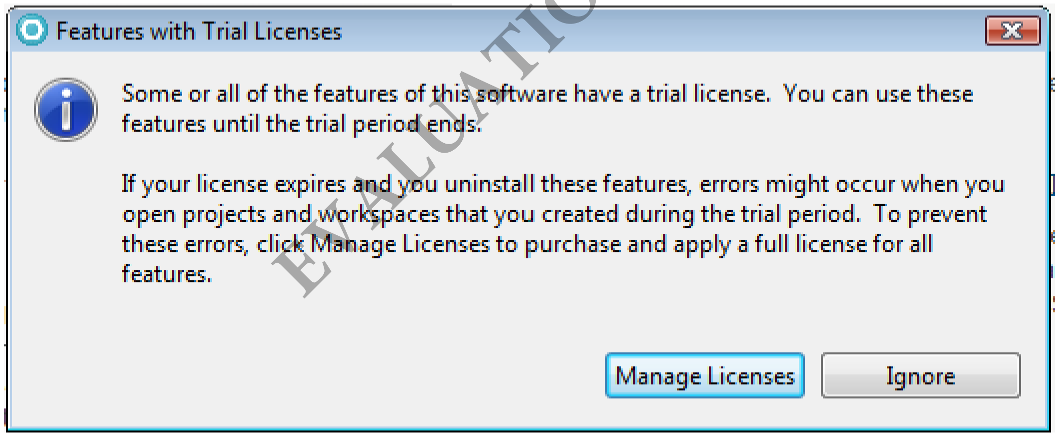
User ID:

Password:

Retype password:

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

__6. Click **Ignore**.

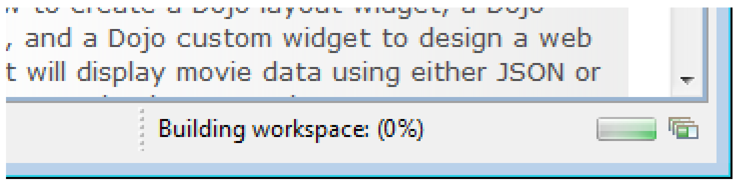

Features with Trial Licenses

Some or all of the features of this software have a trial license. You can use these features until the trial period ends.

If your license expires and you uninstall these features, errors might occur when you open projects and workspaces that you created during the trial period. To prevent these errors, click Manage Licenses to purchase and apply a full license for all features.

__7. If prompted about help content select **Work without Help**.

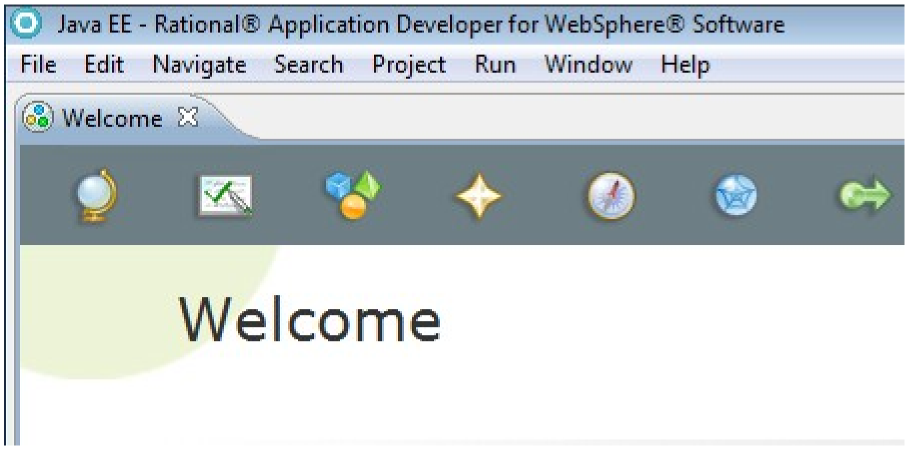
__8. Wait until the messages in the lower right corner of RAD disappear to know the work of creating the server is complete.


, and a Dojo custom widget to design a web
t will display movie data using either JSON or

Building workspace: (0%)

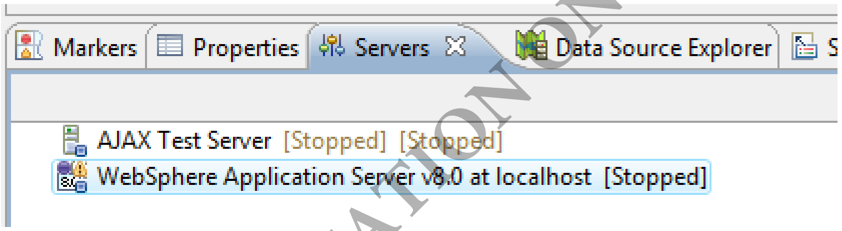
__9. If prompted by Windows firewall security hit the **Cancel** button.

__10. Close the **Welcome** page if it appears.


Java EE - Rational® Application Developer for WebSphere® Software
File Edit Navigate Search Project Run Window Help
Welcome
Welcome

__11. Select the **Servers** view on the bottom of the *Java EE* perspective.

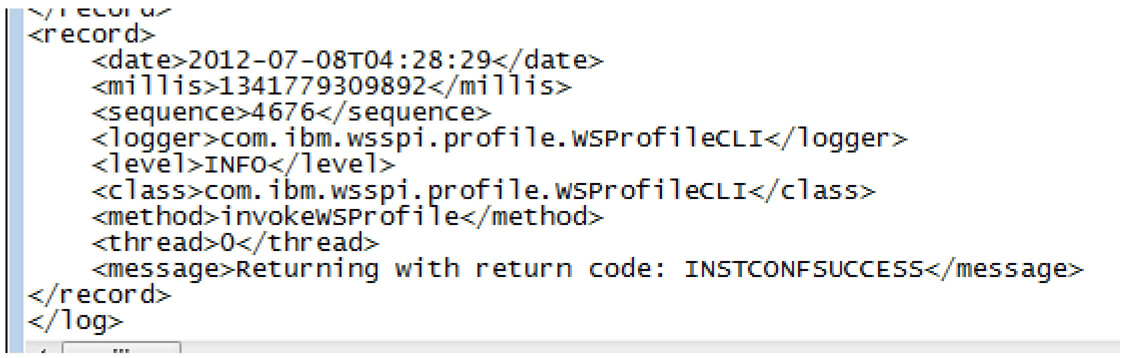
__12. Check that a WebSphere Application Server v8.0 server is shown.


Markers
Properties
Servers
Data Source Explorer
AJAX Test Server [Stopped] [Stopped]
WebSphere Application Server v8.0 at localhost [Stopped]

__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>
    <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.

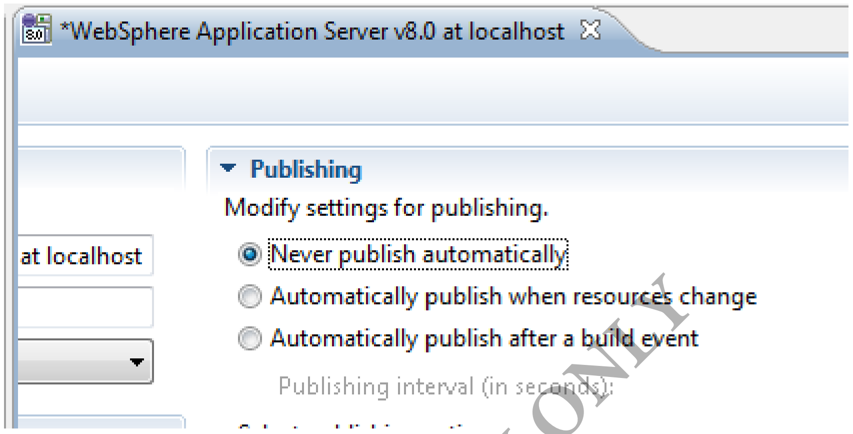
__15. Close the log file.

__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.


*WebSphere Application Server v8.0 at localhost

Publishing

Modify settings for publishing.

[✓ Never publish automatically]

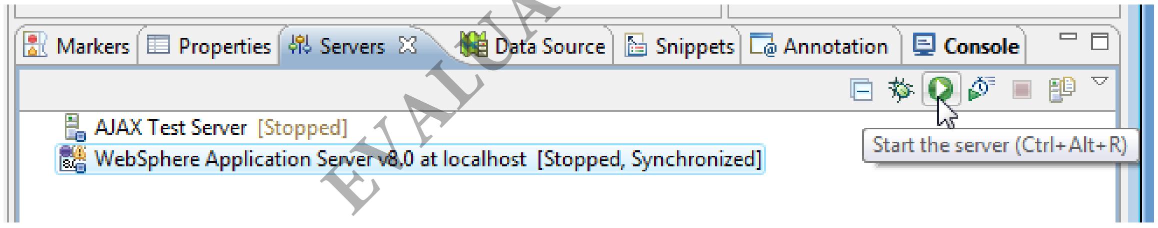
Automatically publish when resources change

Automatically publish after a build event

Publishing interval (in seconds):

__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.

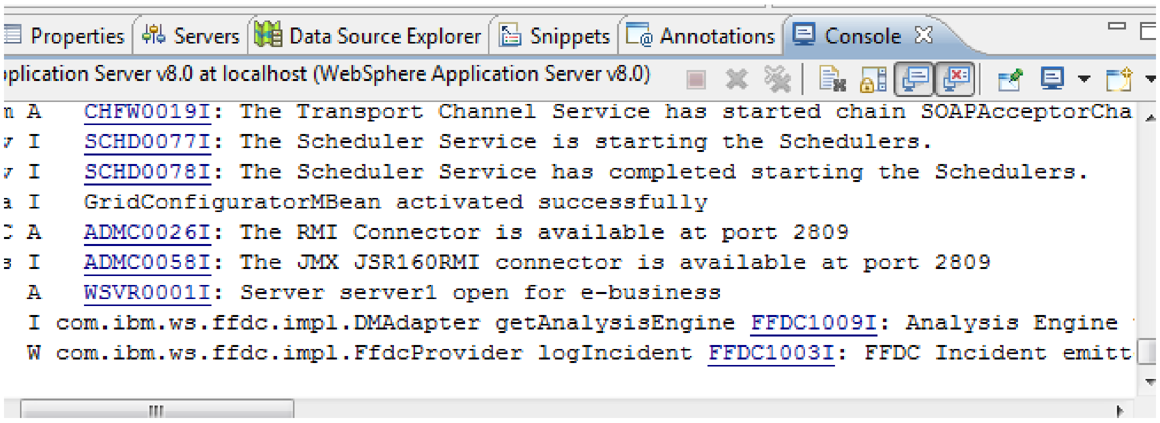


AJAX Test Server [Stopped]

WebSphere Application Server v8.0 at localhost [Stopped, Synchronized]

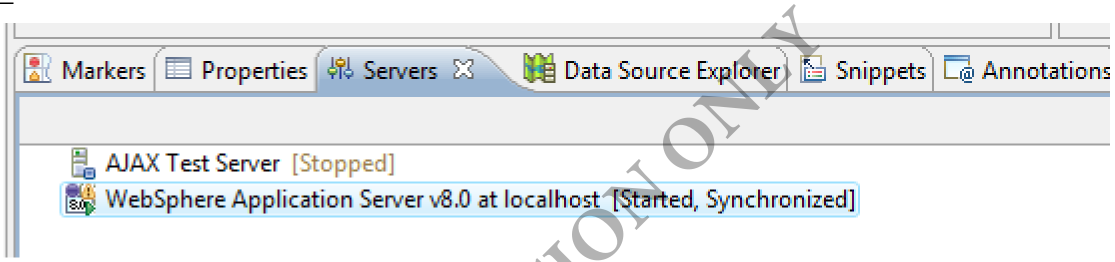
__22. From the menu, select **Window > Show View > Console**.

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



```
ıplication Server v8.0 at localhost (WebSphere Application Server v8.0)
n A   CHFW0019I: The Transport Channel Service has started chain SOAPAcceptorCha
r I   SCHD0077I: The Scheduler Service is starting the Schedulers.
r I   SCHD0078I: The Scheduler Service has completed starting the Schedulers.
a I   GridConfiguratorMBean activated successfully
C A   ADMC0026I: The RMI Connector is available at port 2809
3 I   ADMC0058I: The JMX JSR160RMI connector is available at port 2809
  A   WSVR0001I: Server server1 open for e-business
  I com.ibm.ws.ffdc.impl.DMAdapter getAnalysisEngine FFDC1009I: Analysis Engine
  W com.ibm.ws.ffdc.impl.FfdcProvider logIncident FFDC1003I: FFDC Incident emitt
```

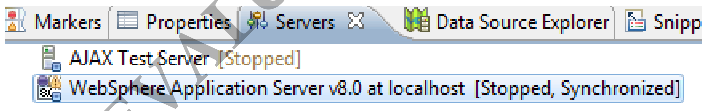
__24. Switch back to the *Servers* view and check that the server was started.



AJAX Test Server [Stopped]
WebSphere Application Server v8.0 at localhost [Started, Synchronized]

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

__26. Check that the server was Stopped.



AJAX Test Server [Stopped]
WebSphere Application Server v8.0 at localhost [Stopped, Synchronized]

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.

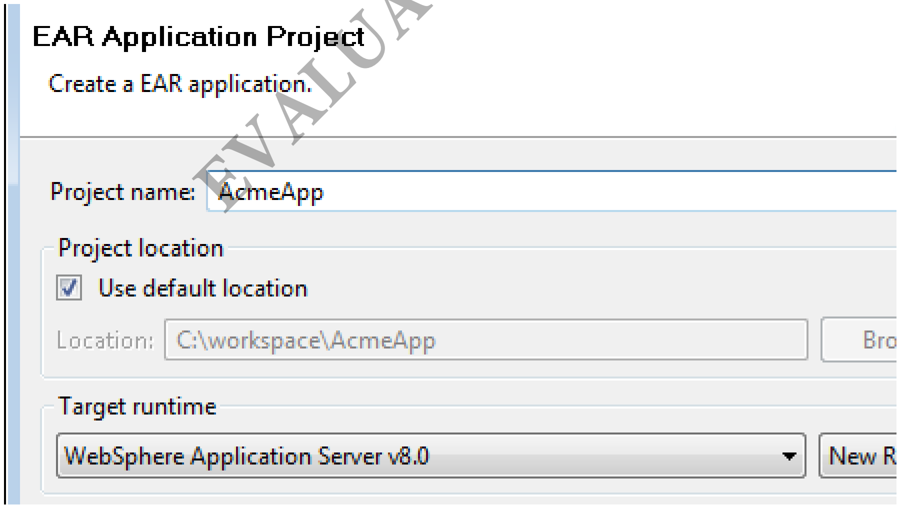
__1. From the menubar, select **File > New > Dynamic Web Project**.

__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.

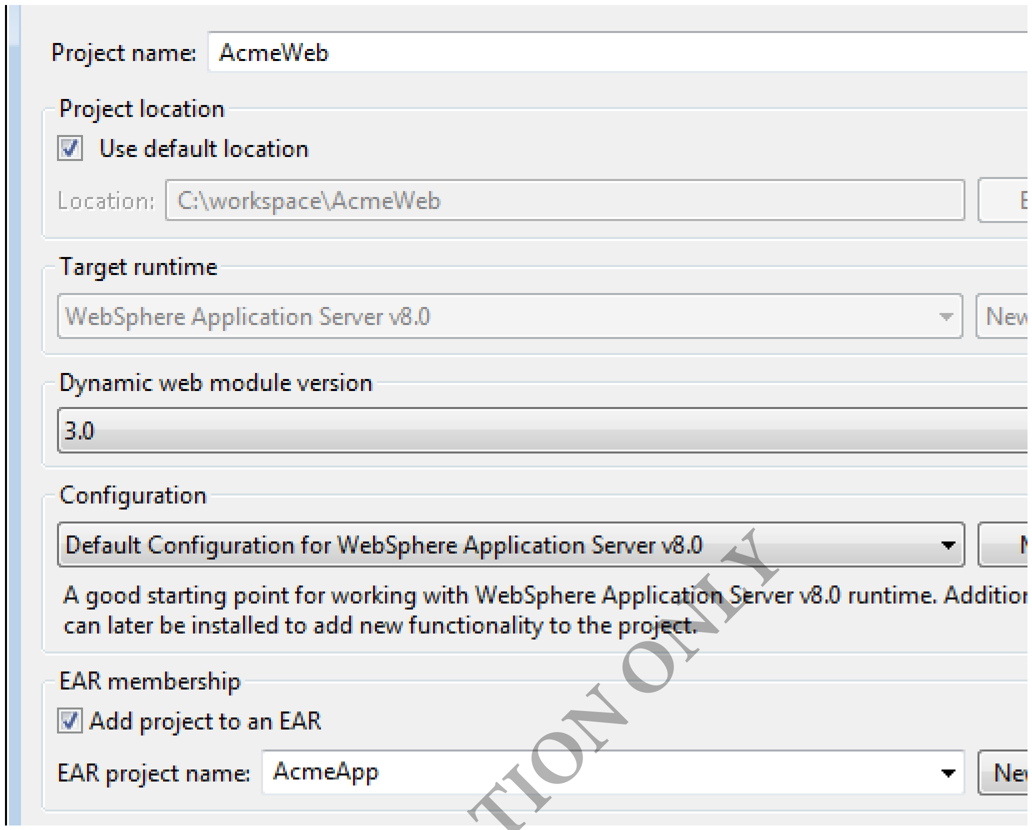
__4. Set the *Project name* to **AcmeApp** and make sure the target server is WebSphere 8.0.


EAR Application Project
Create a EAR application.
Project name: AcmeApp
Project location
☑ Use default location
Location: C:\workspace\AcmeApp
Target runtime
WebSphere Application Server v8.0

__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:


Project name: AcmeWeb

Project location

☑ Use default location

Location: C:\workspace\AcmeWeb

Target runtime

WebSphere Application Server v8.0

Dynamic web module version

3.0

Configuration

Default Configuration for WebSphere Application Server v8.0

A good starting point for working with WebSphere Application Server v8.0 runtime. Additio
can later be installed to add new functionality to the project.

EAR membership

☑ Add project to an EAR

EAR project name: AcmeApp

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

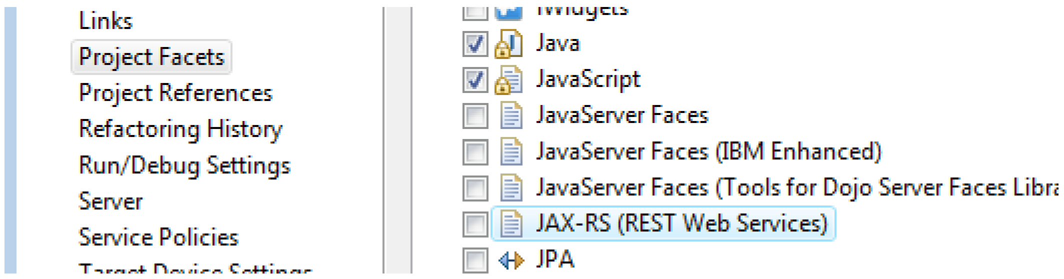
__7. Close the 'Technology Quickstarts' if it opens.

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.



☑ Java

☑ JavaScript

☐ JavaServer Faces

☐ JavaServer Faces (IBM Enhanced)

☐ JavaServer Faces (Tools for Dojo Server Faces Libr

☐ JAX-RS (REST Web Services)

☐ JPA

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

Part 2 - Register the REST Application

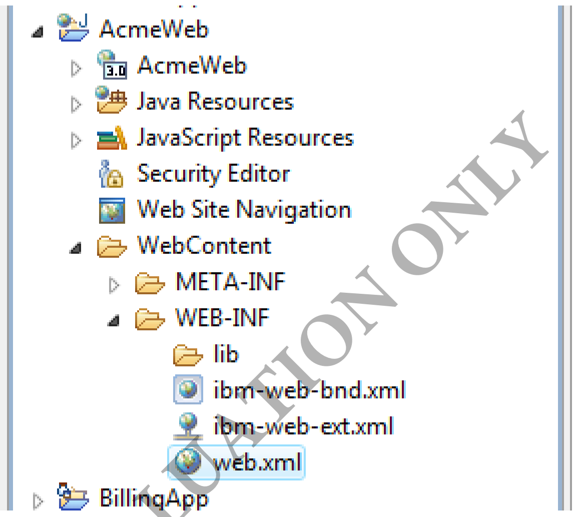
There are a few different ways to configure a REST application. This can depend on if you want a common URL prefix for all REST services. This might be useful if you have REST services within a project with other web components. If you want to register a common URL prefix you can do so in a web.xml file. 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".

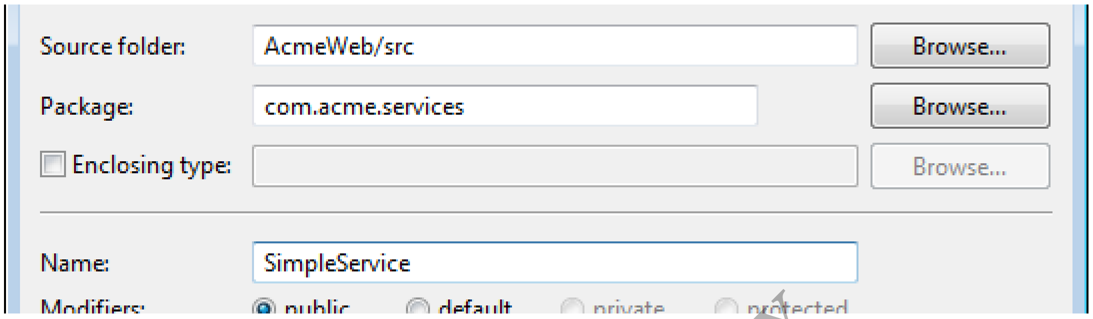
__6. Close the file.

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.



Source folder: AcmeWeb/src

Package: com.acme.services

Enclosing type:

Name: SimpleService

Modifiers: [✓ public] default private protected

__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.

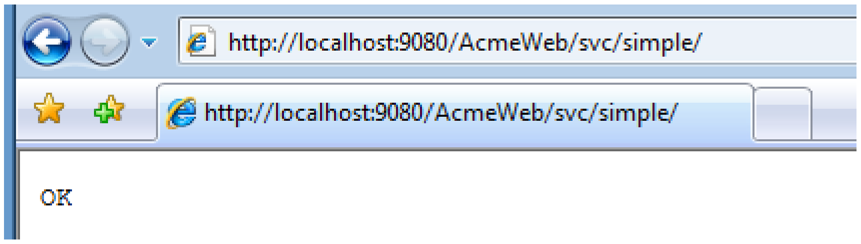
__4. Click **Finish**.

__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:

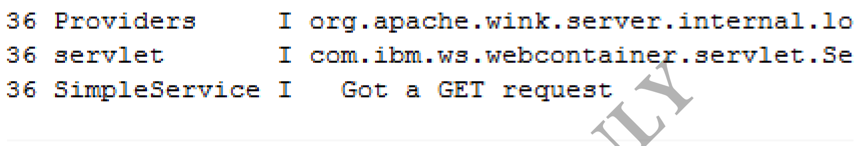
```
http://localhost:9080/AcmeWeb/svc/simple/
```

__7. You should see OK in the browser.


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

```
OK
```

__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() {
```

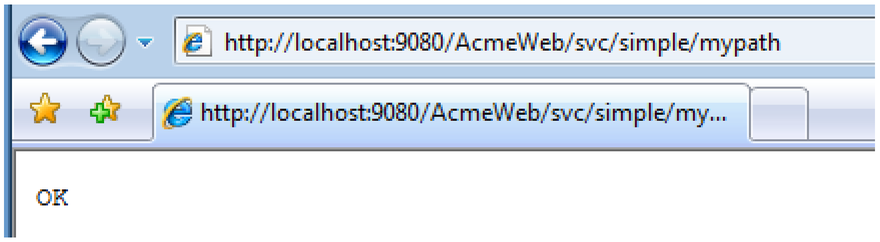
__3. Save changes.

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
```


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

http://localhost:9080/AcmeWeb/svc/simple/my...

```
OK
```

__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.

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:9080/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:9080/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.

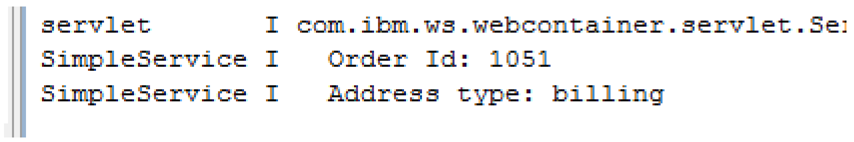
```
public String getAddress(
          @PathParam("orderId")
          int oId,
          @PathParam("addressType")
          String type) {
```

__4. Save changes.

__5. Publish the server.

__6. Test the method by entering the URL:

```
http://localhost:9080/AcmeWeb/svc/simple/somedata/1051/address/billing
```



```
servlet       I com.ibm.ws.webcontainer.servlet.Se
SimpleService I   Order Id: 1051
SimpleService I   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:9080/AcmeWeb/svc/simple/somedata/raise?name=Daffy&amount=10000
```

__6. Make sure that the Console shows:

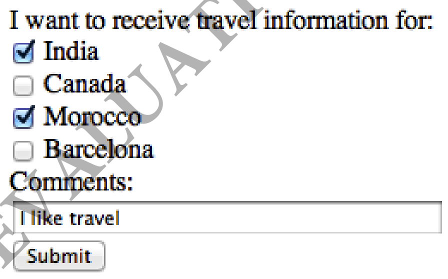
```
Giving raise to Daffy by 10000.0
```

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.


I want to receive travel information for:

- [✓] India
- [ ] Canada
- [✓] Morocco
- [ ] Barcelona

Comments:

I like travel

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";
}
```

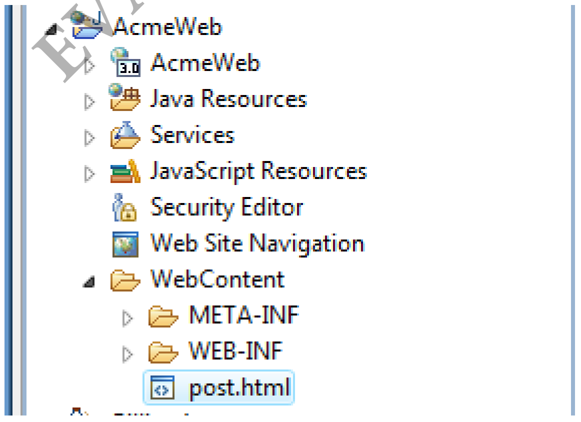
Note that the HTTP method is set to POST. Also, we expect multiple values for the "interest" parameter. That is why we have set the data type of the variable to java.util.List<String>.

__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.



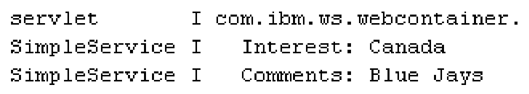
__2. In the *Servers* view, right click the server and select **Publish**.

__3. Open a browser and enter the URL:

```
http://localhost:9080/AcmeWeb/post.html
```

__4. Fill out the form and submit it.

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



```
servlet       I com.ibm.ws.webcontainer.
SimpleService I   Interest: Canada
SimpleService I   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.