

REST API Guidebook

This guidebook provides an independent exercise that supports the *Representational State Transfer (REST) with Java, Beginner Part 1* course.

Table of Contents

Using the VMware Environment

Accessing the Environment	2
Setting Up Your Project Files.....	3

Importing an Existing Project	4
--	---

The Eclipse Interface

Top Bar	6
Project Explorer Pane.....	7

Setting Up For Testing

How to Test Your Exercises	11
---	----

Course 1: REST Exercise

Exercise: Connecting to a Live REST API	13
---	----

Accessing Additional Resources	29
---	----

Using the VMware Environment

VMware is a tool offered by VA ITWD that allows you to log in to a virtual machine loaded with all of the tools needed to create a REST API. You can follow the step-by-step directions listed in this guidebook to recreate the demonstrations shown during the course presentation to help you practice in a hands-on environment.

Accessing the VMware Environment

Upon registration for the *Introduction to Representational State Transfer (REST) with Java, Beginner Part 1* course, TMS ID 3878052, you will receive an email message with the link to the VMware virtual machine.

Depending upon your user profile, you may see icons for other tools that you have access to use in this virtual environment.



Setting Up Your Files in the Environment

After opening Eclipse, you need to set up a workspace to make sure your work is in a saved folder designated for you.

1. Select File Tab

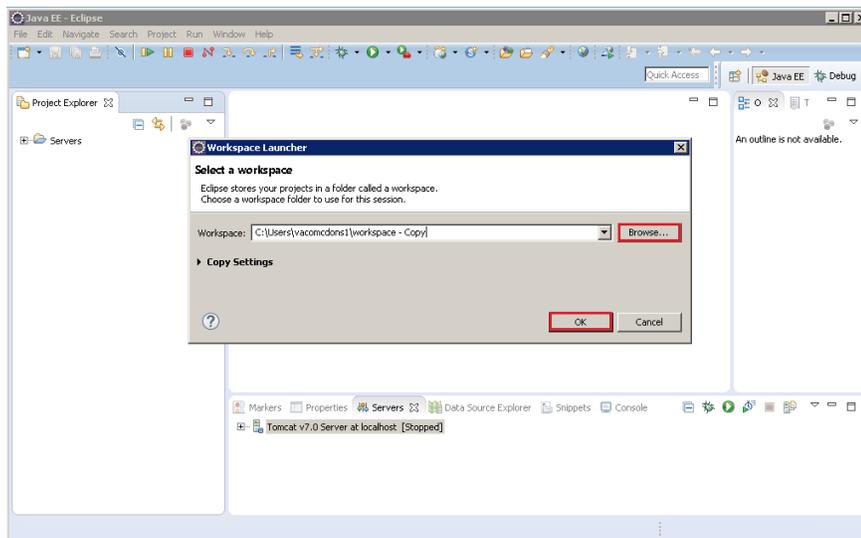
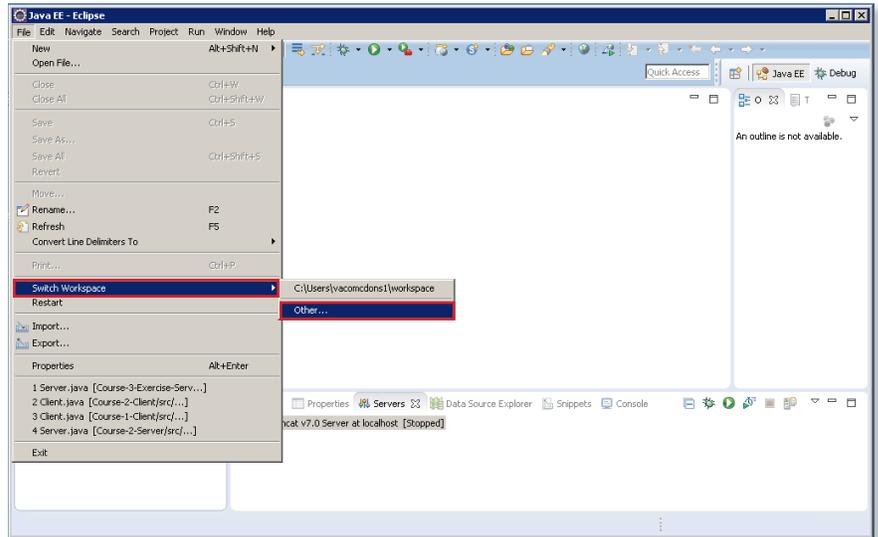
The **File** tab opens to display its menu.

2. Select Switch Workspace

Selecting **Switch Workspace** opens a sub-menu.

3. Select Other

Selecting **Other** opens the **Workspace Launcher** dialog box.



4. Browse to the Workspace

In the **Workspace** field of the **Workspace Launcher**, select **Browse** and navigate to the folder where you want to save your work.

Select or create a folder associated with your account, such as a folder on your Desktop or in **My Documents**.

Select **OK** to finish.

Your REST API project will now be saved in the workspace you designated. Your VM workspace will remain active for the two weeks following the course.

Importing an Existing Project

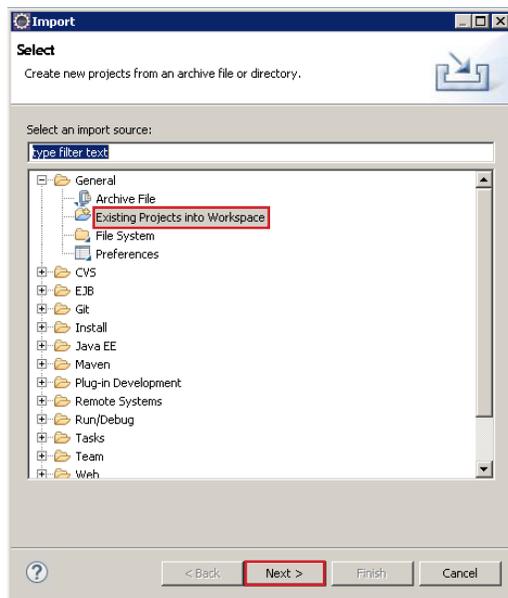
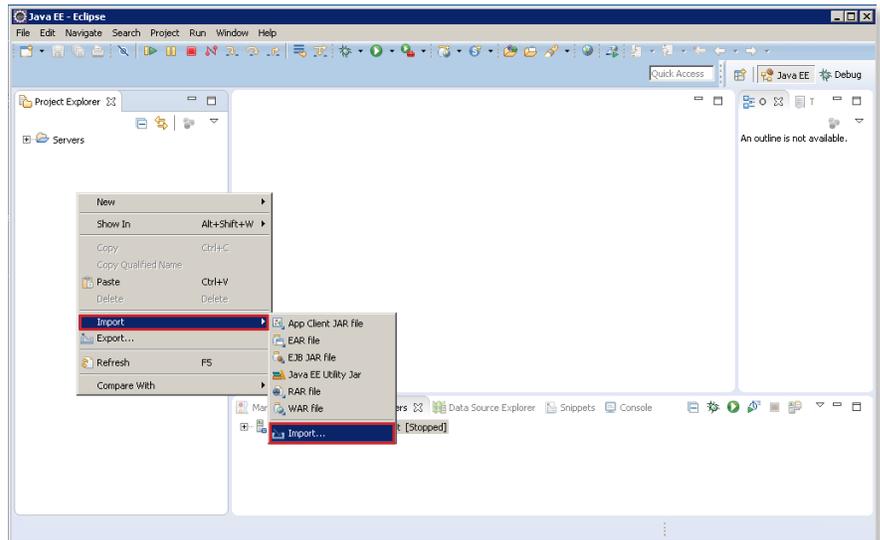
When you want to import an existing REST project into Eclipse, open Eclipse to start the import.

1. Right-Click on the Project Explorer Pane

A dialog box will open with several different options; navigate down to the Import option and select it.

2. Select Import

Once you've selected **Import**, you'll need to navigate down to the bottom of the sub-menu and select the **Import** option again. Selecting **Import** will open the **Import** dialog box shown below in step three.

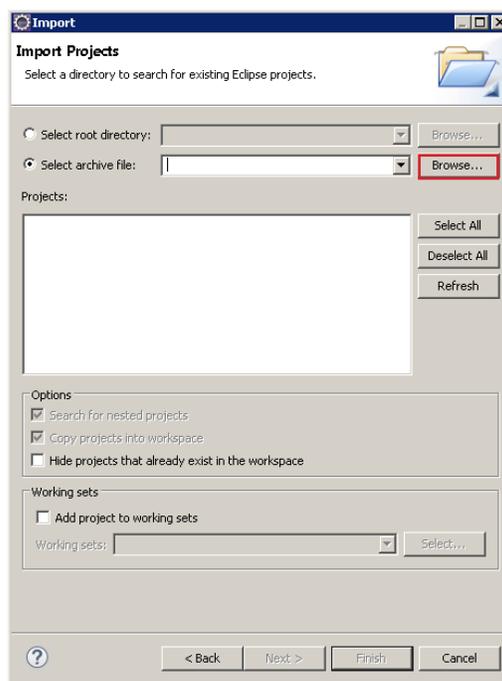


3. Select an Import Source

From the **Import** dialog box, select **General** and **Existing Projects** into Workspace. Select **Next** to continue.

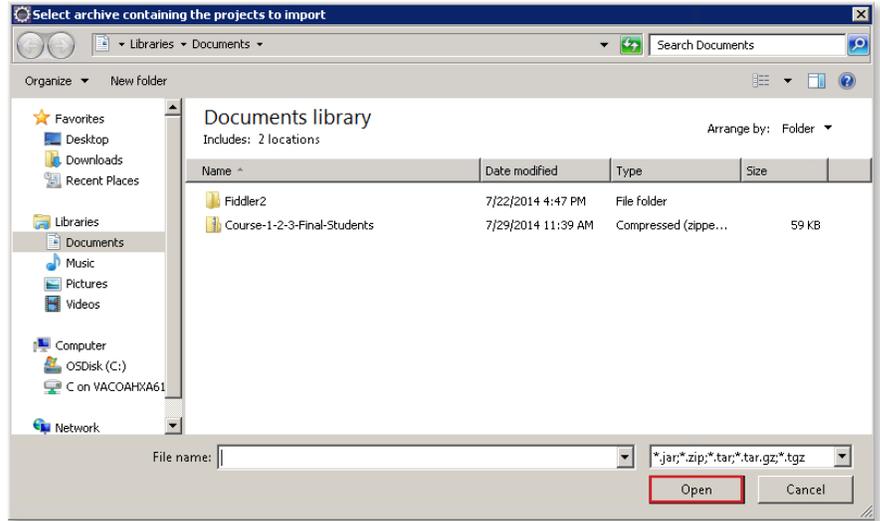
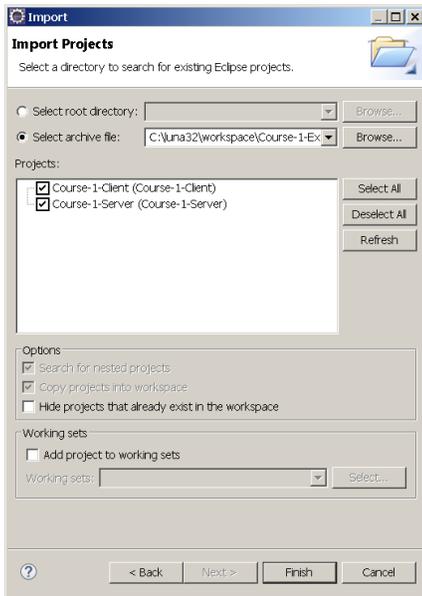
4. Select the Project to Import

On the **Import Project** dialog box, select the **Browse** button.

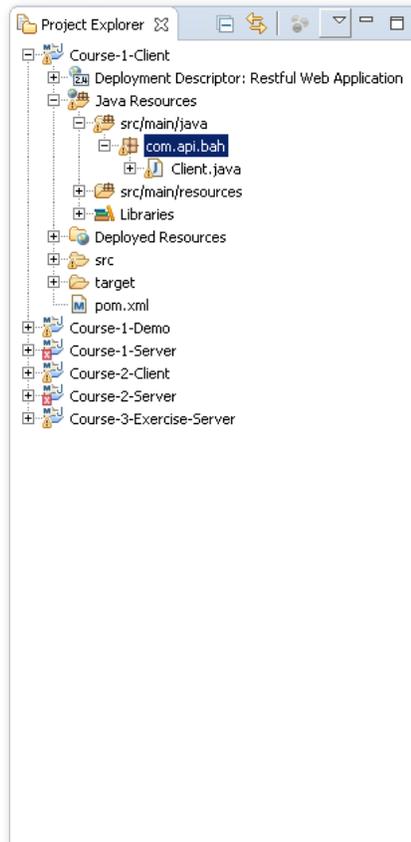


5. Locate on Your Computer Where the Project is Stored

Navigate to and select the zip file that contains the project, then select **Open**.



This will show you your projects in the zip file.

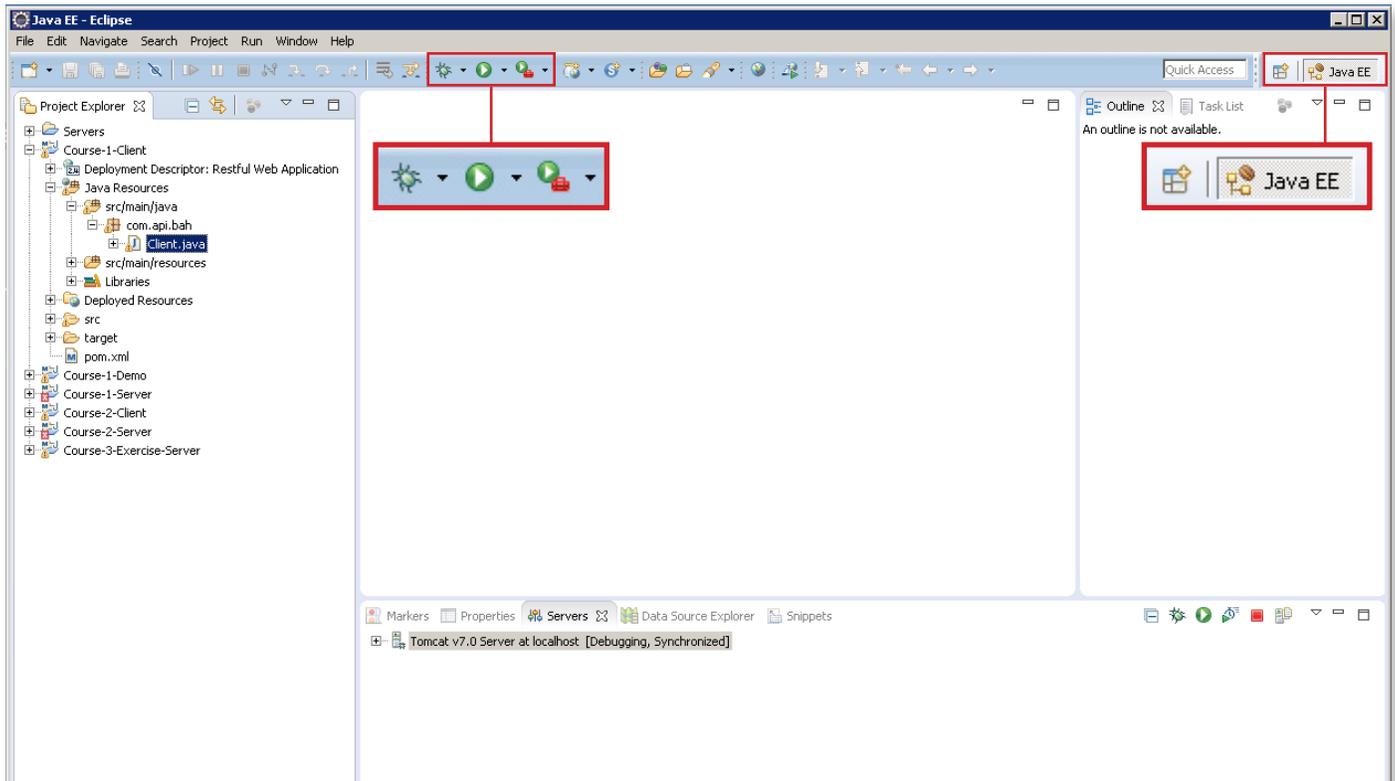


6. View the Imported Project in Project Explorer

Go to **Project Explorer** to locate the existing project that was just imported into your Eclipse. From there, you can add or edit the REST API project as you wish.

The Eclipse Interface

To help you become familiar with Eclipse's interface, we've highlighted some of its tools and features and provided the purpose and function of each.



1. Top bar

The buttons on this bar help run and edit your applications. The buttons we're going to highlight are the ones that will help you debug and run your assignments.

a. Debug Button

This button runs your app in debug mode by attaching the debugger. It enables you to discover and diagnose coding and other mistakes that may occur in your application. This is especially useful for determining run-time errors or errors that can only be detected once the application has launched. One of the many useful features of the debugger is the ability to step through your code and examine the contents of variables.

b. Run Button

This button runs your API without attaching the debugger. When developing your APIs, it is a best practice to avoid using this button because it does not show you where errors are, only that you have them. Once your system is in production, it should always be compiled without debug options enabled.

c. Open Perspective Button

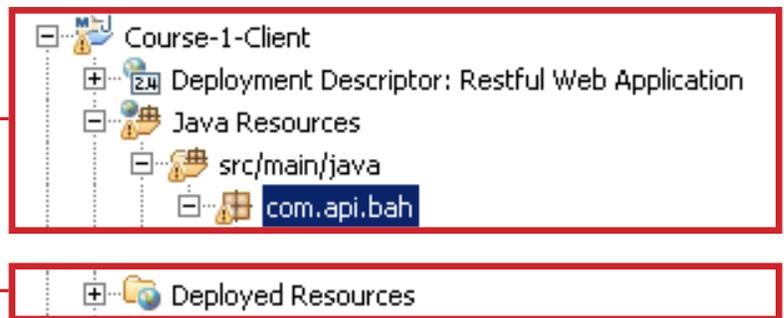
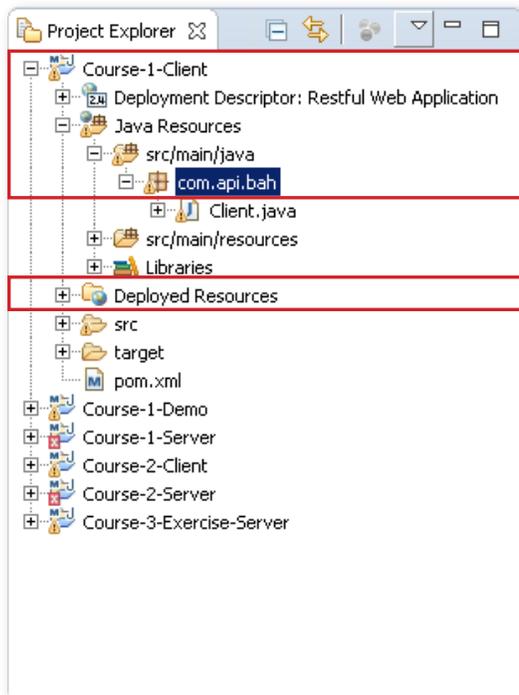
This button provides different views to assist you in completing specific steps while creating REST APIs. This button switches between different perspectives. The perspectives that will be most helpful in your assignments will be the Debug and Java.

d. Java EE Perspective Button

This button is another perspective button and it opens the **Java EE perspective**. It's used for Java projects and will be helpful if you get lost and need a shortcut to get back to your projects.

2. Project Explorer Pane

The purpose of this pane is to organize the files of the project you're working on. Files in this pane are displayed in a hierarchical view to help show how project files are arranged. The project files we're going to highlight will help you understand how a REST API project written in Java is organized in Eclipse.



a. Top Folder

The folder on the top level of a project holds all your files for a specific project. The tiny “M” and the “J” on the folder icon have a specific meaning to your project. The “M” stands for Apache Maven, which is a build automation tool. (Maven is outside of the scope of these assignments.) The “J” stands for Java, which tells you and Eclipse that this project is written for the Java platform.

b. Deployment Descriptor

As the project hierarchy is expanded, the next file you'll notice is the **Deployment Descriptor**. This file identifies the project as a RESTful web application.

c. Java Resources

The next folder in the same level of hierarchy as the **Deployment Descriptor** is the **Java Resources** folder. This folder holds content such as Java code and the file where you will edit your assignments.

src/main/java folder

As we expand down the Java Resources hierarchy one level, you'll see the **src/main/java** folder. This folder holds the file, **Server.java**, to edit your assignments. To locate this file, expand down the **Java Resources**.

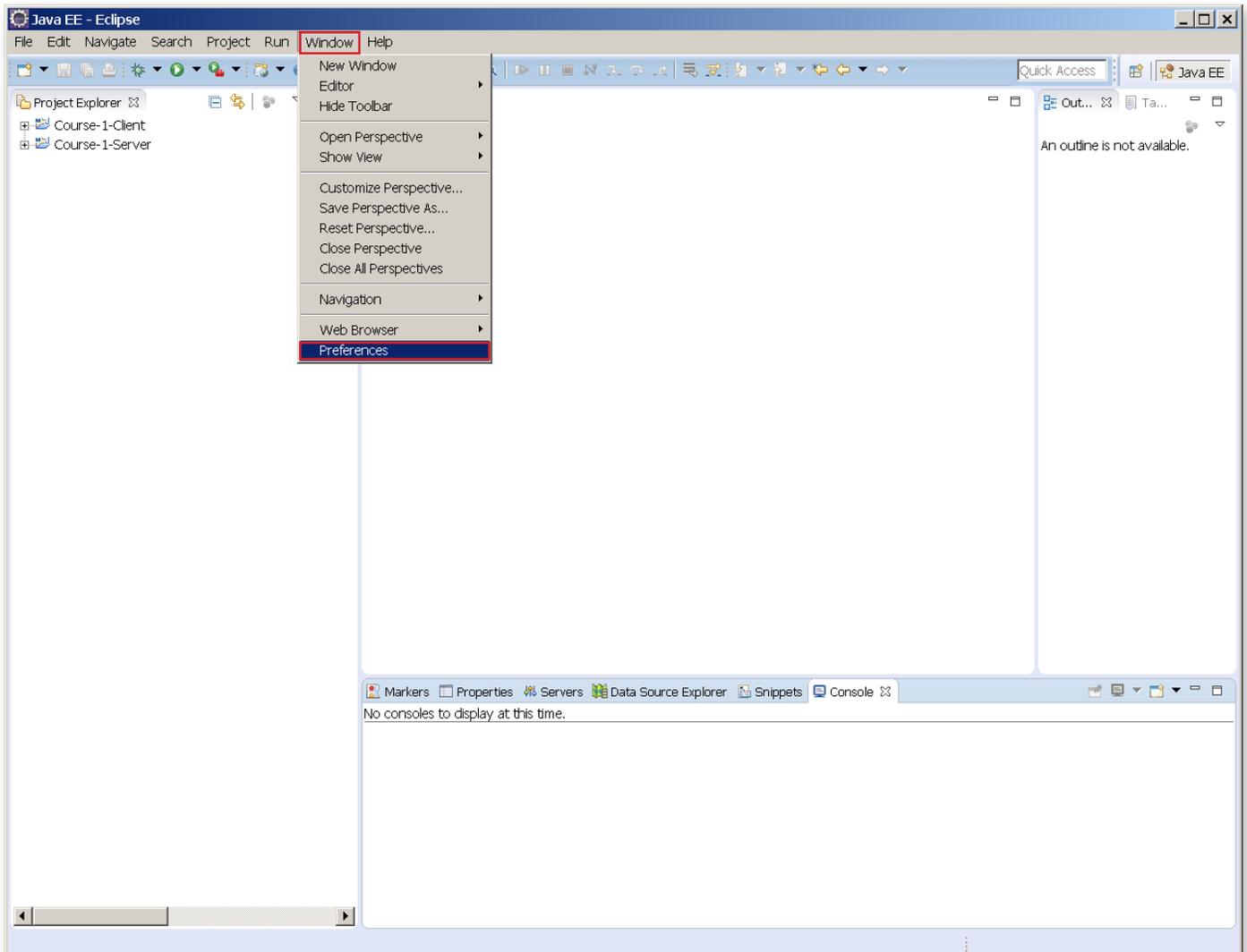
d. src and target folders

The **src** and **target** folders are in the same level of hierarchy as the **Java Resources** folder and shouldn't be edited or removed. The files in these folders were created to point to specific files necessary for running your program.

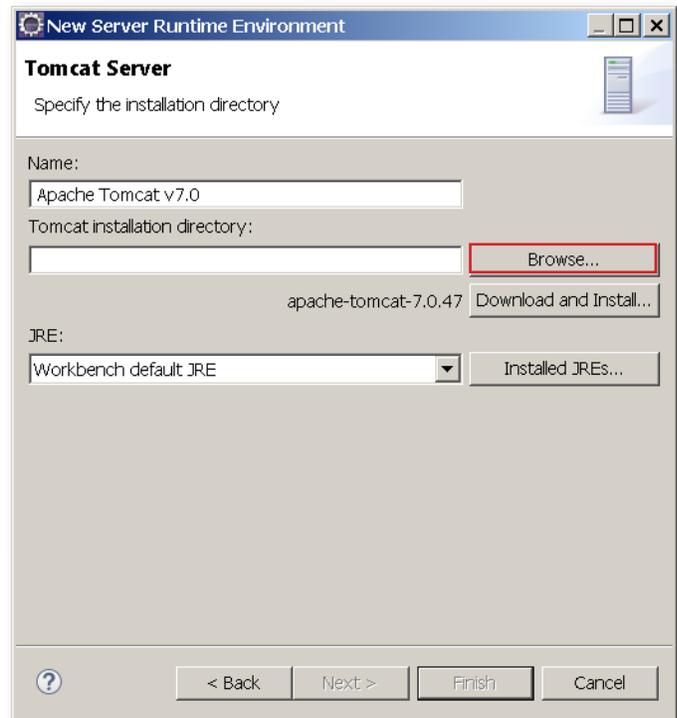
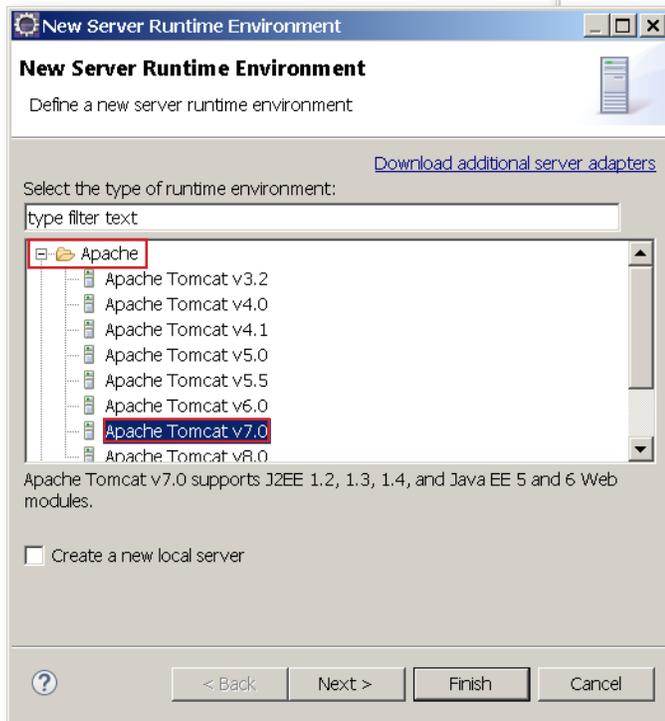
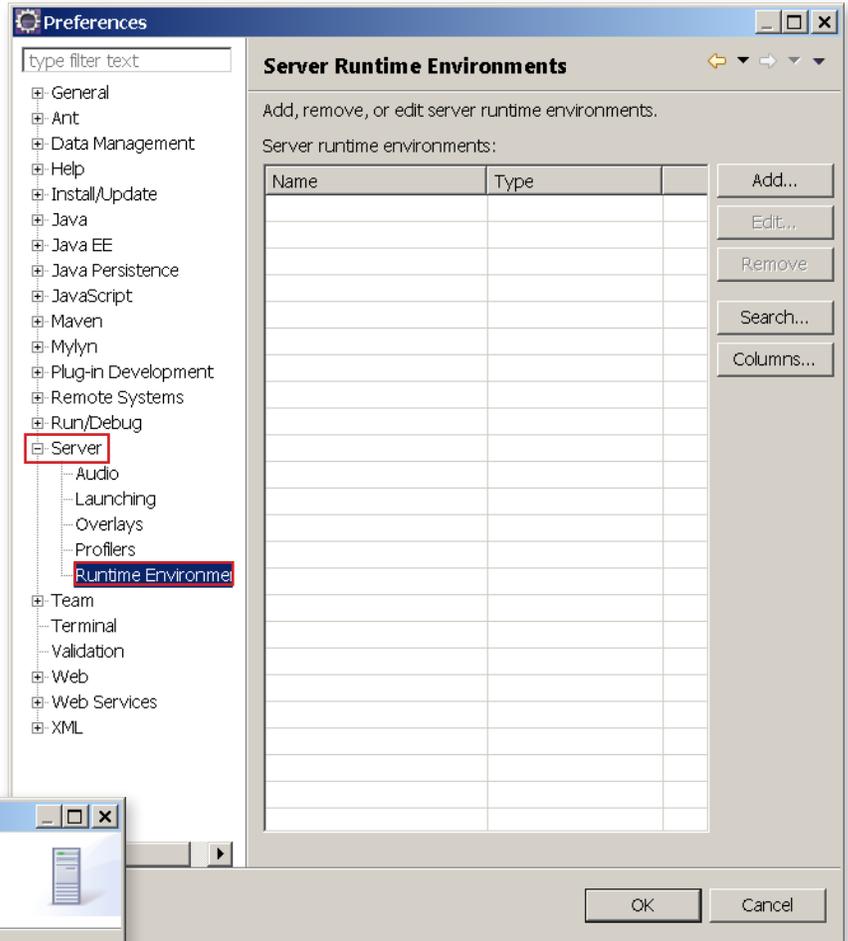
Setting Up For Testing

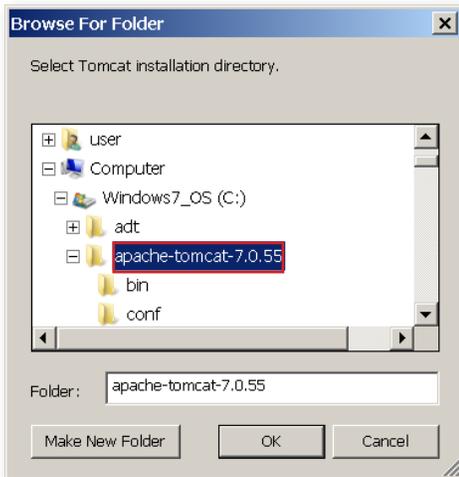
Before you can test your server project, you will need to define the server runtime executables that Eclipse will need to load. In our case, we are using Apache Tomcat version 7.

1. Click on **Window** in the top menu and go to **Preferences**.



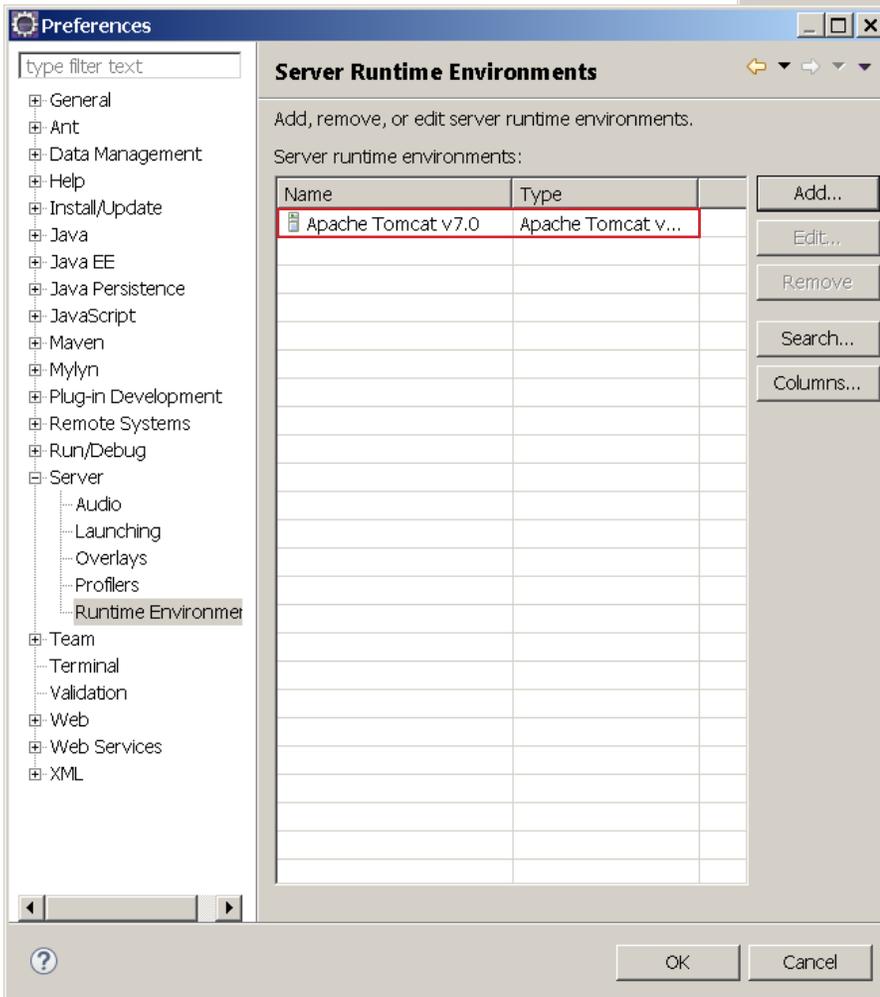
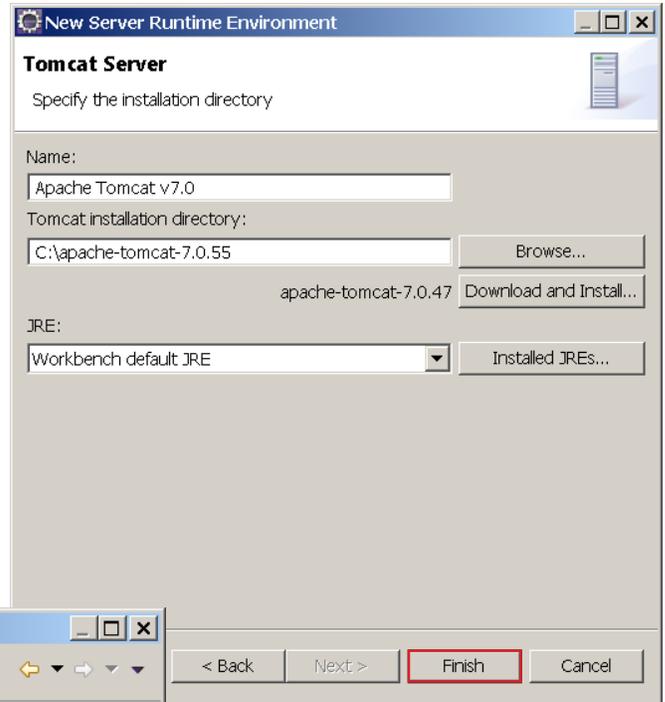
2. Expand the **Server** tab on the left, and select **Runtime Environment**.
3. Expand the **Apache** folder and select **Apache Tomcat 7.0**. Click **Next**.
4. Click **Browse** to select the **Apache Tomcat 7.0** folder.





5. Select the `c:\apache-tomcat-7.0.55` folder and click **OK**.

6. Click **Finish**.



7. You will see that the **Apache Tomcat 7.0** runtime environment has been linked to Eclipse.

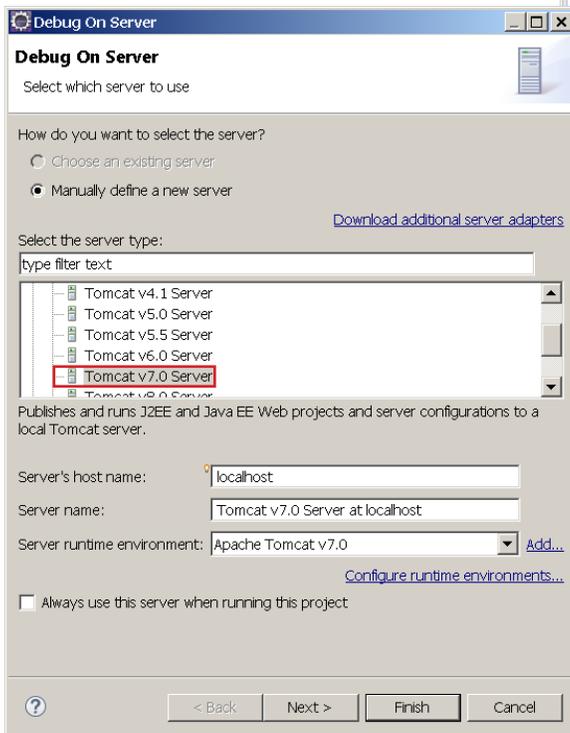
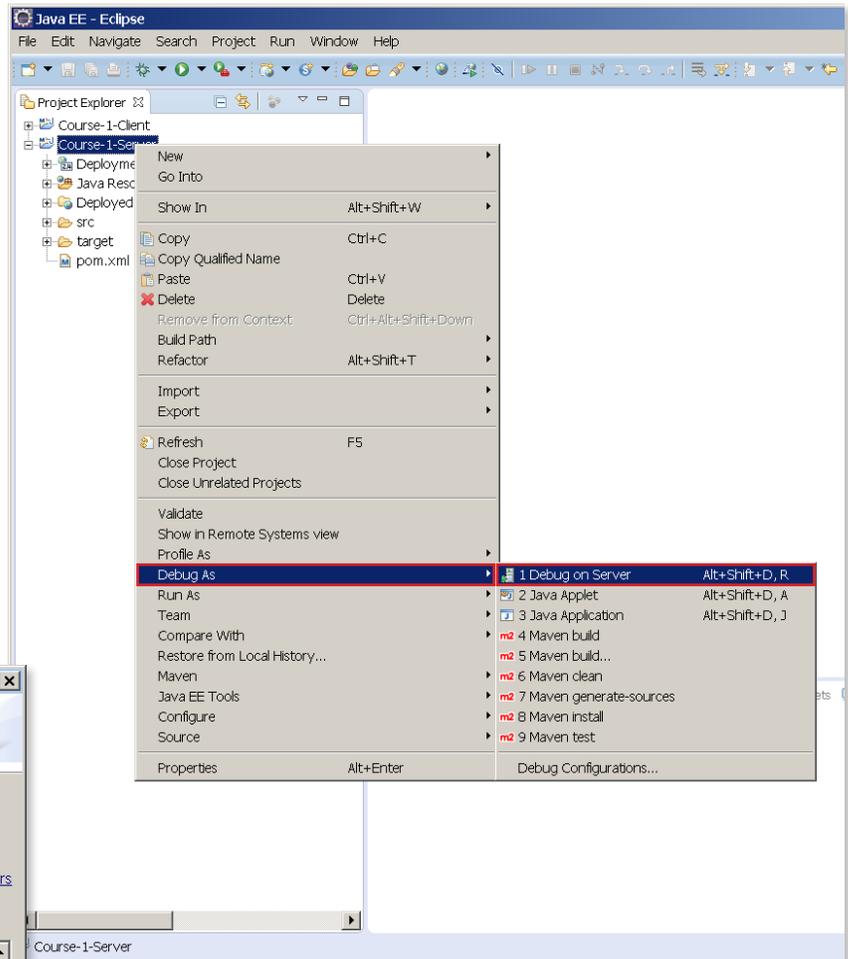
That's it—now Eclipse knows how to run your web server applications.

How to Test Your Exercises

Once you have completed the exercises in the next section of this guidebook, and you have resolved all the errors, you are ready to test. To test your exercises, you will need to select the context (click on the project folder), and then debug your project. For the clients (and other projects that do not need to run in a server container), debug as Java Application. For server applications, you will need to launch the debugger from the **Project Explorer** pane. The steps below show you how to test your exercises.

1. Right-Click on the Top Folder of the project you're working on

When you right-click on the top folder a sub-menu will open. Navigate down to **Debug As** and select Java Application for non-server code. For server code, select the first option, **1. Debug on Server**. This starts the application running on the server. Selecting **1. Debug on Server** will open the dialog box shown below.

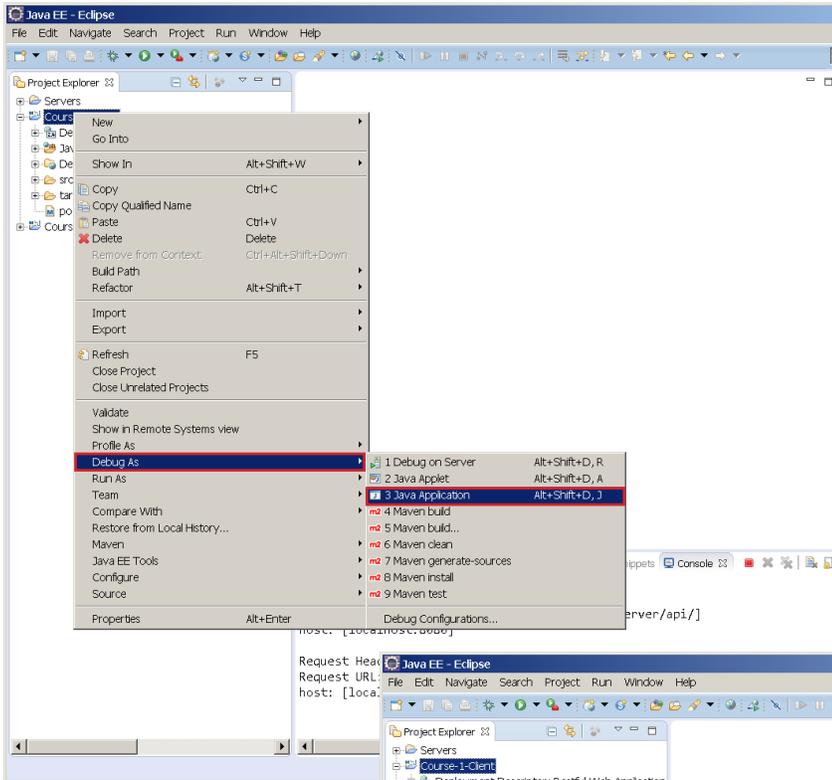


2. Debug On Server

A dialog box will open the **Debug on Server** screen. Expand the **Apache** folder and select **Tomcat v7.0 Server at localhost** and then select **Finish**. This will run the REST API that you're working on.

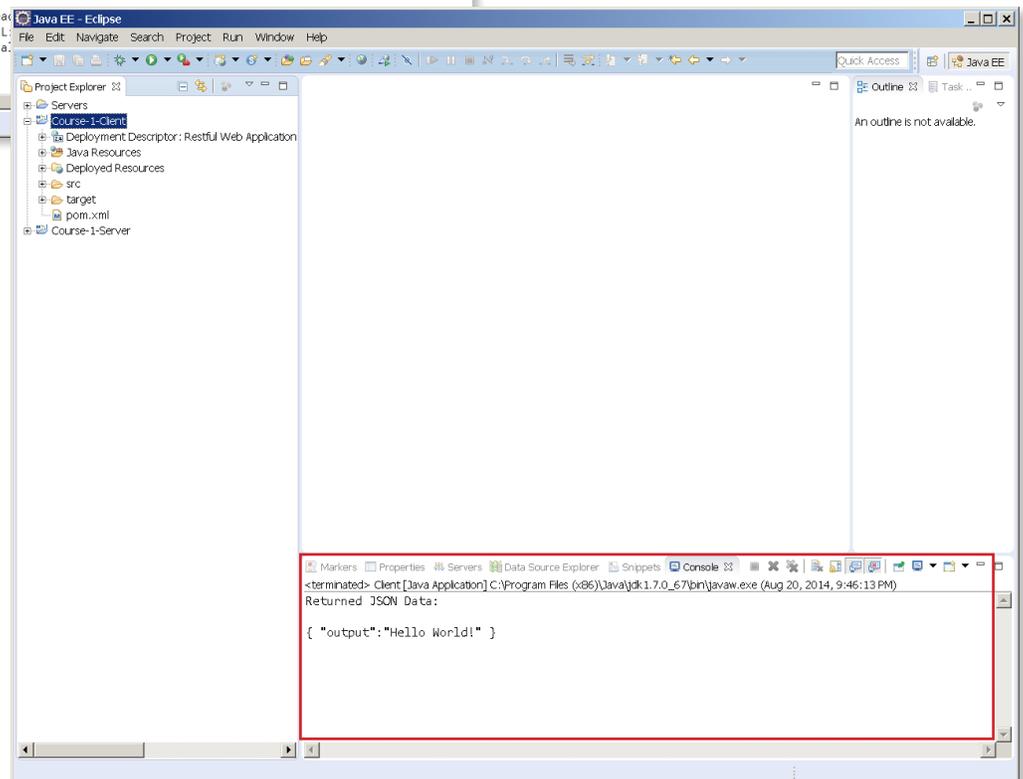
3. Debug As Java Application

Right-click on the top folder of the project you're working on to open a sub-menu. Navigate down to **Debug As** and select the third option, **3. Java Application**. This runs the client, and output will be displayed in the **Console** area.



4. View Results in the Console area

Once the results have been output, you will be able to view them in the console areas of Eclipse.



Course 1: REST Exercise

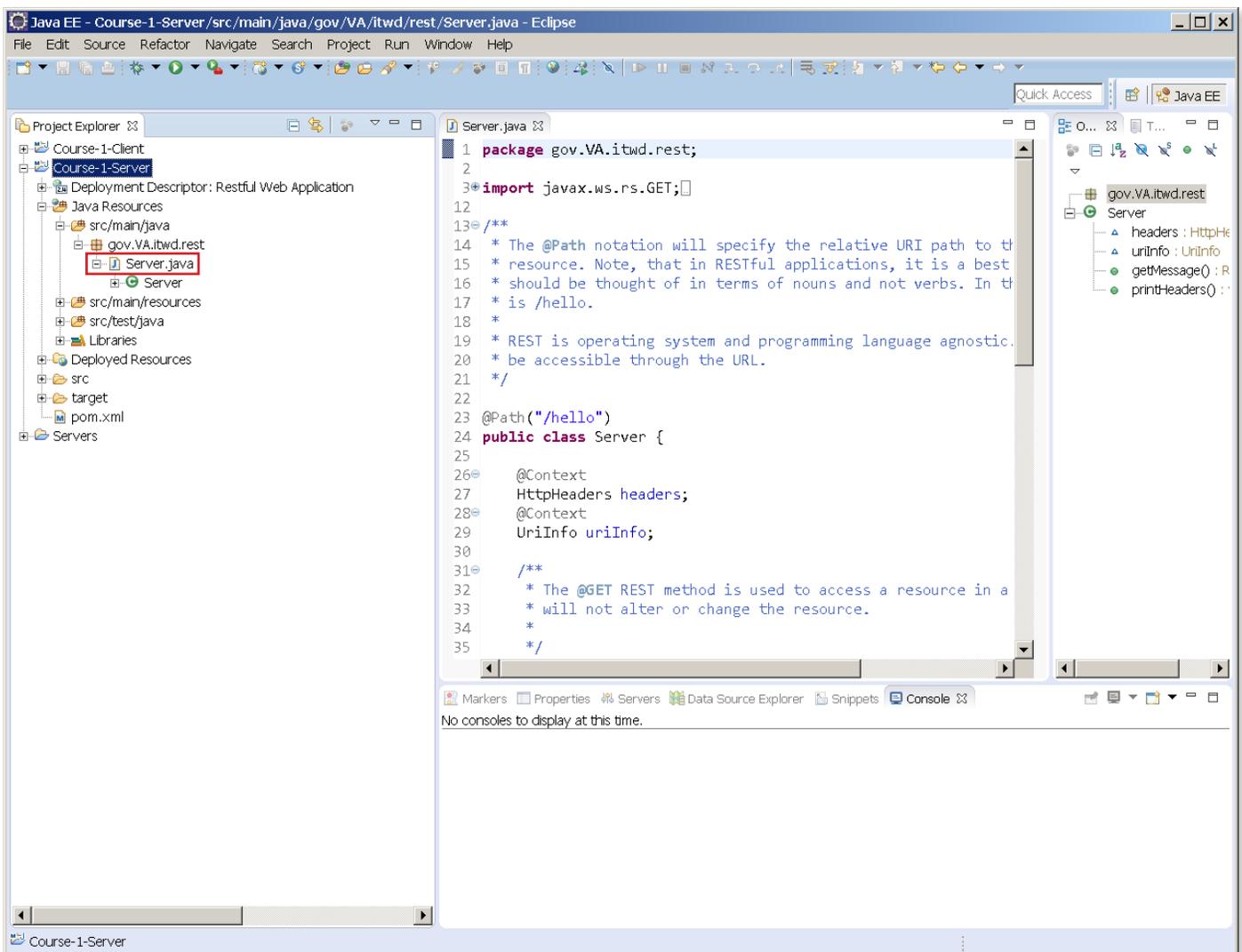
Exercise: Connecting to a Live REST API

In this first exercise, you're going to create a simple API that will produce JSON data. There's one way of accessing this API and it will respond the same way every time. This response is similar to a doorbell. Every time you push the doorbell, it rings with the exact same tone. Similar to a doorbell, the response in this API will never change. We will be using the Jersey library implementing the JAX-RS interfaces. To begin this exercise, you'll need to import the following existing REST API project: Course-1.zip.

Follow the steps to import an existing project, which are listed in the beginning of this guidebook. The steps to opening that project are listed below.

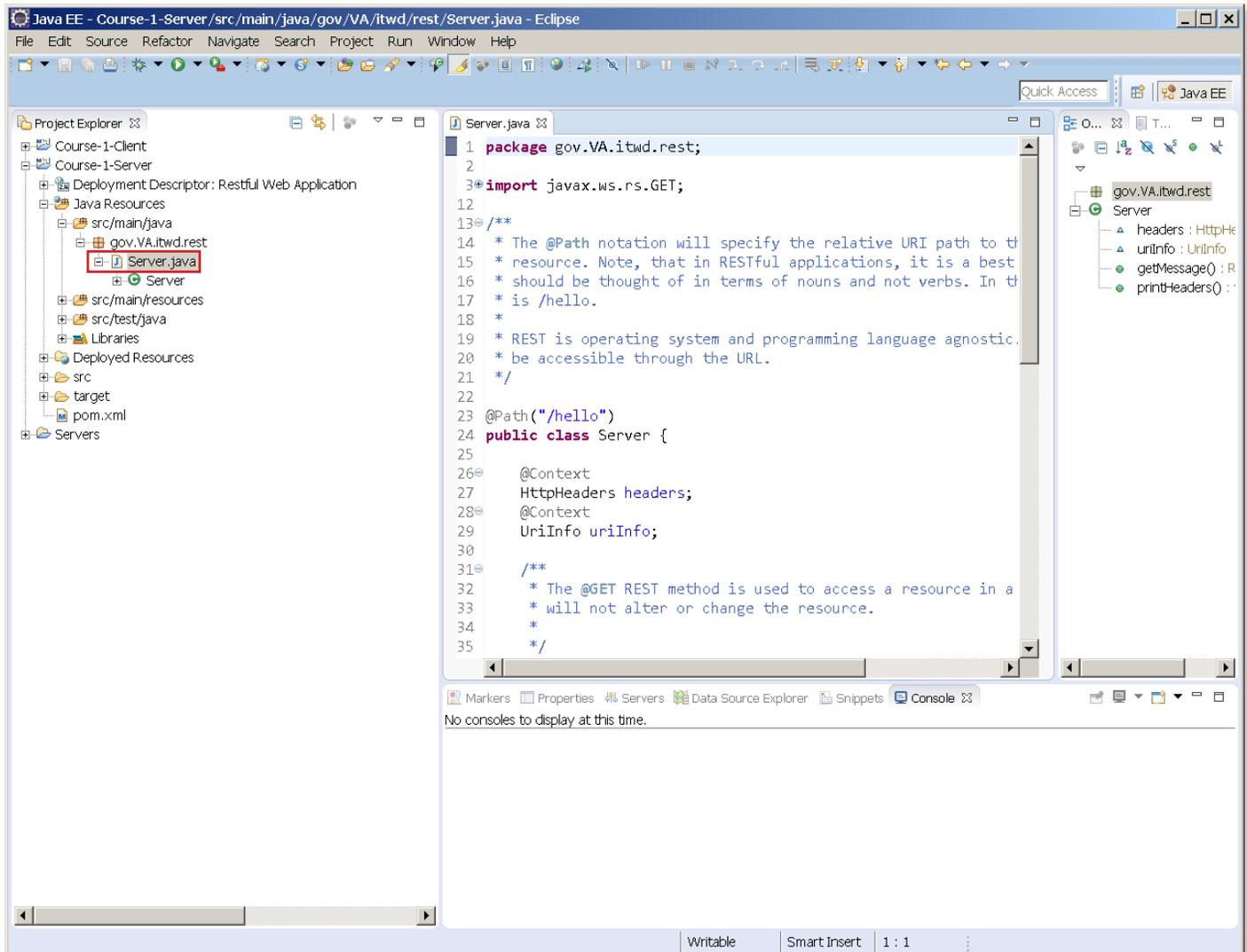
1. Import the Course 1 Files into the Package Explorer

Once you've imported the course files, your **Package Explorer** will look like the screenshot below. You'll need to expand your folders to locate the **Server.java** file.



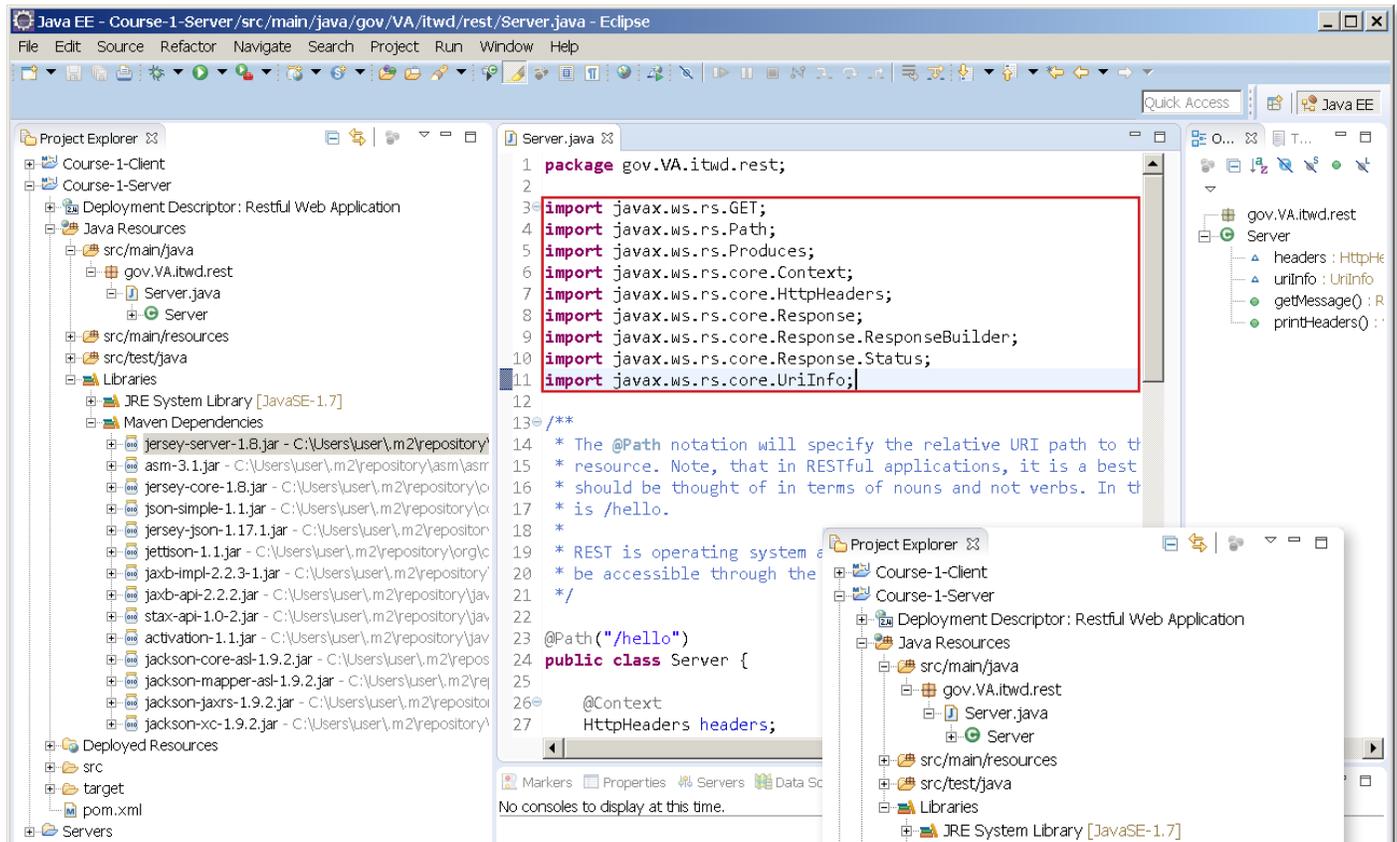
2. Expand the Java Resources Folder to the Client.java File

Next, you'll need to expand the **Java Resources** folder. Once you've expanded this folder, locate the **src/main/java** folder and expand it to open up the **gov.VA.itwd.rest** package. After expanding the package, locate the **Server.java** file and double-click it to open.



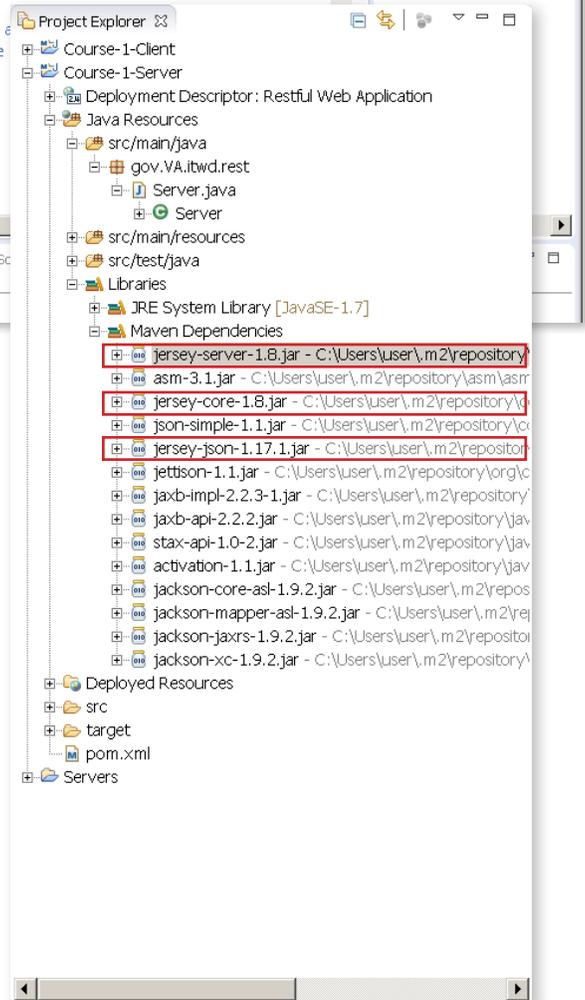
3. View the Jersey Library Imports

Notice the code at the top. Here, we are importing relevant members of the JAX-RS API and using Jersey as the implementation. JAX-RS provides an API for creating REST applications.



In the next screenshot, you can see the Jersey libraries listed in the Maven dependencies.

Tip: Do NOT alter the imports in this exercise, even though some are reported in warnings as not used (if you do the exercise right, those warnings will go away!).



4. Create @GET and @Produces Function

Scroll down in the code and locate the **@GET** block comments. Locate the line comments that ask you to complete the **@GET** and **@Produces** function, follow the directions, and create both functions.

```
Server.java
21  */
22
23  @Path("/hello")
24  public class Server {
25
26  @Context
27  HttpHeaders headers;
28  @Context
29  UriInfo uriInfo;
30
31  /**
32   * The @GET REST method is used to access a resource in a read-only manner. @GET
33   * will not alter or change the resource.
34   *
35   */
36  // Place an @GET here
37  // Place an @Produces("application/json") here
38  public Response getMessage() {
39      // Print out the HTTP Request
40      printHeaders();
41      /*
42       * Use the response builder to create a successful (200) status code In
43       * Java we use the ResponseBuilder which lets us attach multiple
44       * elements to the rest After adding the status to the response we now
45       * attach the user's data we found Add the following string to the
46       * entity "{ \"output\": \"Hello World!\" }" Build the response and send
47       * it back to complete the request via a return. Hint: ResponseBuilder
48       * is not invoked as a new object.
49       */
50      return null;
51  }
52
53  /**
54   * This prints out the headers from the URL so you can visualize what the
55   * request looks like from the console.
56   */
57  public void printHeaders() {
58      System.out.println("Request Headers -----");
59      System.out.println("Request URL: [" + uriInfo.getBaseUri() + "]");
60      for (String key : headers.getRequestHeaders().keySet()) {
61          System.out.println(key + ": " + headers.getRequestHeaders().get(key));
62      }
63  }
64  }
```

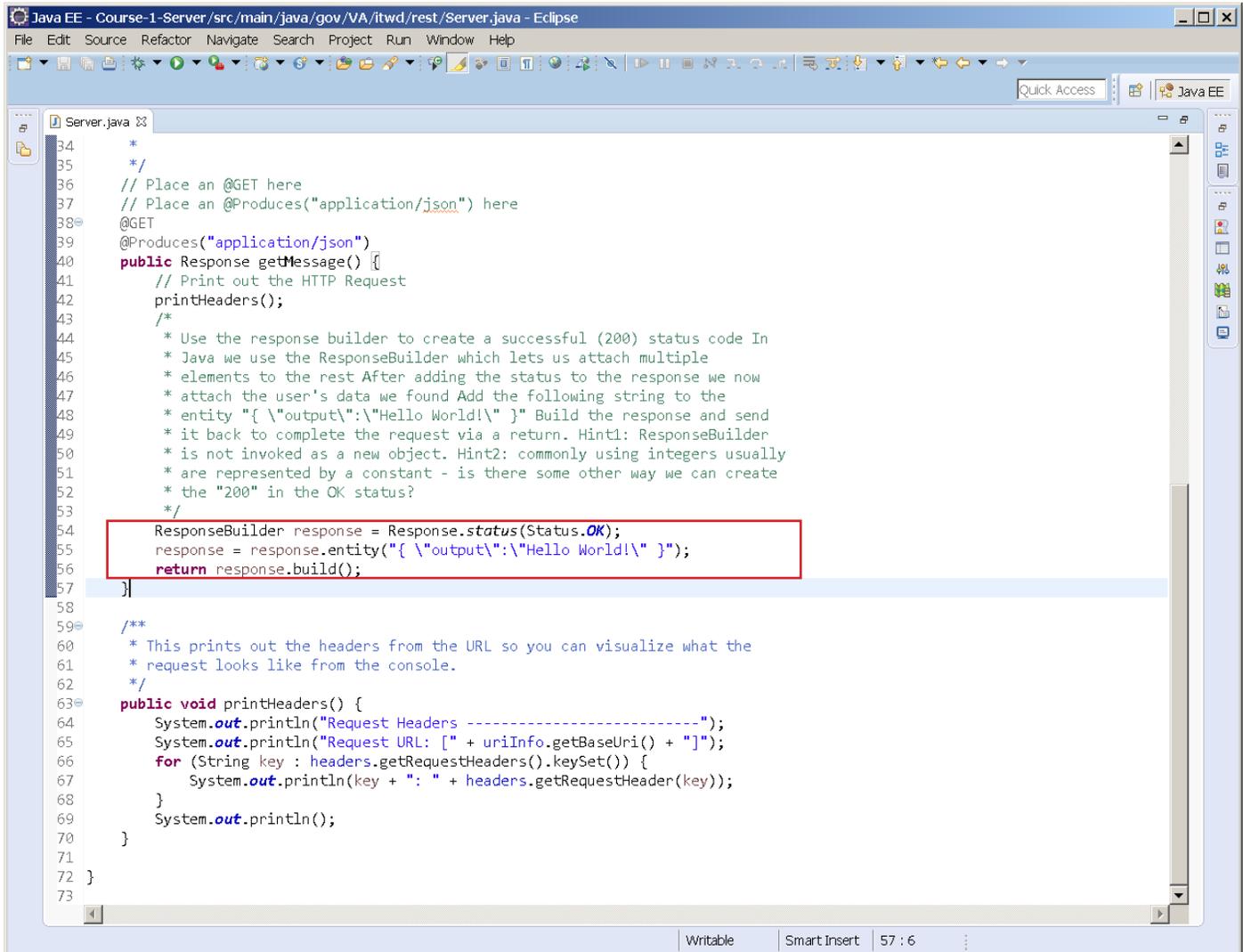
5. Use the Response Builder to Create a Successful Status Code, Add Data, and Build the Response to Complete the Return

First, locate the `printHeaders()` function. From the line comments, use the `ResponseBuilder` to create a successful (200) status code.

Hint: Well-known, often-used codes are usually constants in Java; see if you can find a way of representing 200(OK) from the Response library.

```
Server.java
21  +/
22
23  @Path("/hello")
24  public class Server {
25
26      @Context
27      HttpHeaders headers;
28      @Context
29      UriInfo uriInfo;
30
31      /**
32       * The @GET REST method is used to access a resource in a read-only manner. @GET
33       * will not alter or change the resource.
34       *
35       */
36      // Place an @GET here
37      // Place an @Produces("application/json") here
38      public Response getMessage() {
39          // Print out the HTTP Request
40          printHeaders();
41          /**
42           * Use the response builder to create a successful (200) status code In
43           * Java we use the ResponseBuilder which lets us attach multiple
44           * elements to the rest After adding the status to the response we now
45           * attach the user's data we found Add the following string to the
46           * entity "{ \"output\": \"Hello World!\" }" Build the response and send
47           * it back to complete the request via a return. Hint: ResponseBuilder
48           * is not invoked as a new object.
49           */
50          return null;
51      }
52
53      /**
54       * This prints out the headers from the URL so you can visualize what the
55       * request looks like from the console.
56       */
57      public void printHeaders() {
58          System.out.println("Request Headers -----");
59          System.out.println("Request URL: [" + uriInfo.getBaseUri() + " ]");
60          for (String key : headers.getRequestHeaders().keySet()) {
61              System.out.println(key + ": " + headers.getRequestHeaders().get(key));
62          }
63      }
64  }
```

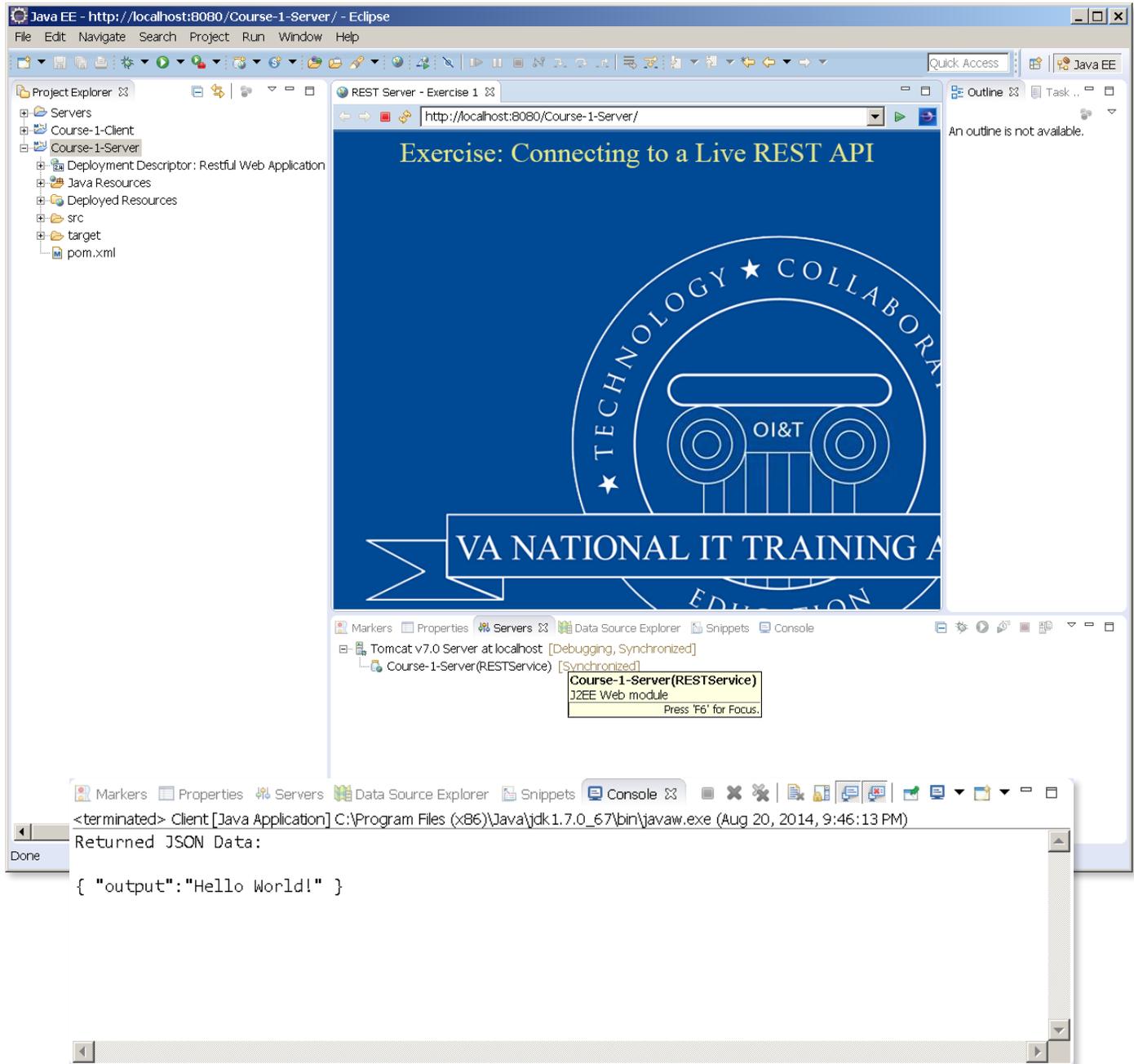
After the **ResponseBuilder** is initialized with a status code, attach the **JSON** in the comments before building the response. Build the response and send it back to complete the request by changing the return null to return your response at the end of the method block.



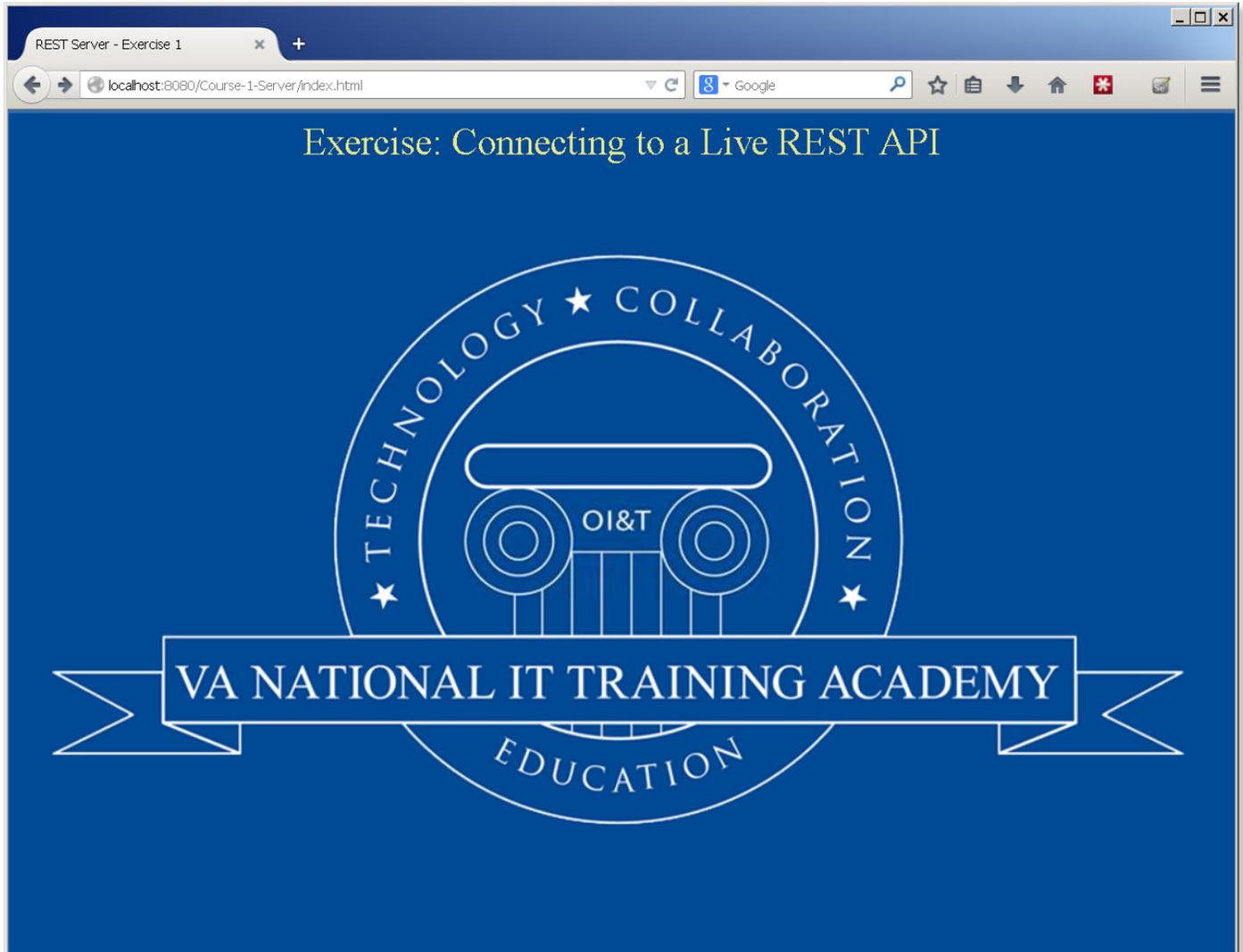
```
34     *
35     */
36     // Place an @GET here
37     // Place an @Produces("application/json") here
38     @GET
39     @Produces("application/json")
40     public Response getMessage() {
41         // Print out the HTTP Request
42         printHeaders();
43         /*
44         * Use the response builder to create a successful (200) status code In
45         * Java we use the ResponseBuilder which lets us attach multiple
46         * elements to the rest After adding the status to the response we now
47         * attach the user's data we found Add the following string to the
48         * entity "{ \"output\": \"Hello World!\" }" Build the response and send
49         * it back to complete the request via a return. Hint1: ResponseBuilder
50         * is not invoked as a new object. Hint2: commonly using integers usually
51         * are represented by a constant - is there some other way we can create
52         * the "200" in the OK status?
53         */
54         ResponseBuilder response = Response.status(Status.OK);
55         response = response.entity("{ \"output\": \"Hello World!\" }");
56         return response.build();
57     }
58
59     /**
60     * This prints out the headers from the URL so you can visualize what the
61     * request looks like from the console.
62     */
63     public void printHeaders() {
64         System.out.println("Request Headers -----");
65         System.out.println("Request URL: [" + uriInfo.getBaseUri() + "]");
66         for (String key : headers.getRequestHeaders().keySet()) {
67             System.out.println(key + ": " + headers.getRequestHeader(key));
68         }
69         System.out.println();
70     }
71 }
72 }
73 }
```

6. Debug and Test Your REST API

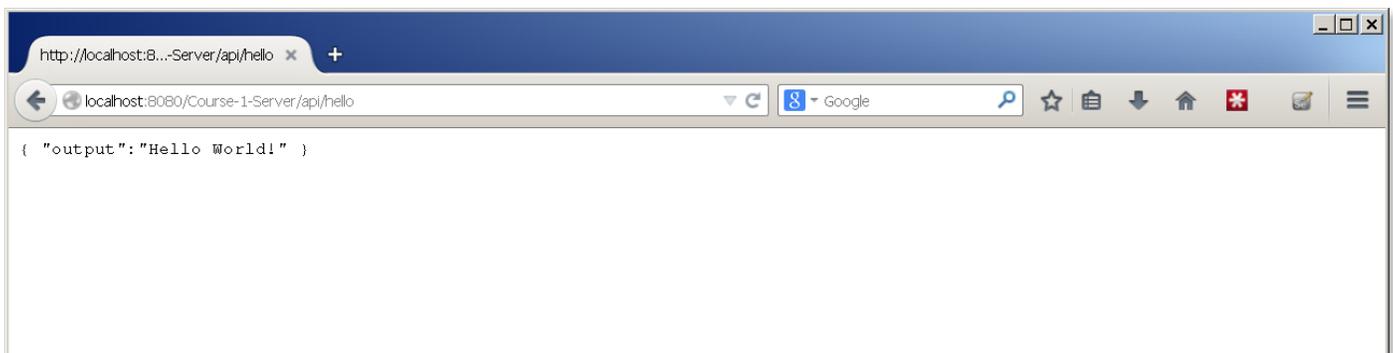
Follow the steps for How to Test Your REST APIs listed in this guidebook. Run your server code by using the **Debug as Server** and using the Tomcat 7.x container. Once that has initialized (you will know by the web page that pops up), run the client portion with **Debug As Java Application**.



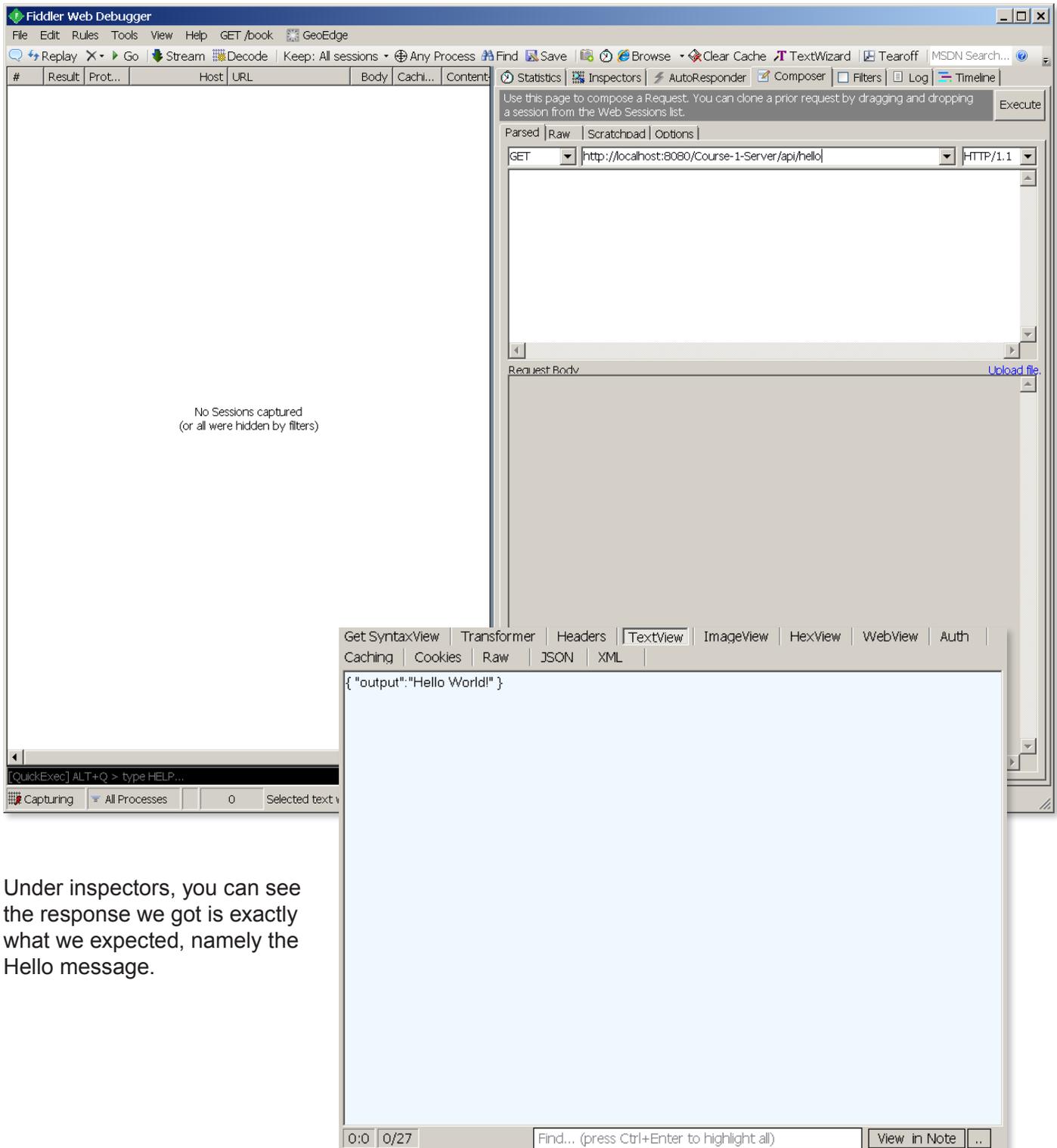
Since this is a web application, you can use any web browser to run it, not just the one built into Eclipse. Here, we are using Firefox. We included a basic index.html file so that the default view would have something to display. Of course, this is a RESTful application. Part of the power of REST is its simplicity.



If you just type in the access URL for our REST application into any browser, the app will respond, as expected and shown here.

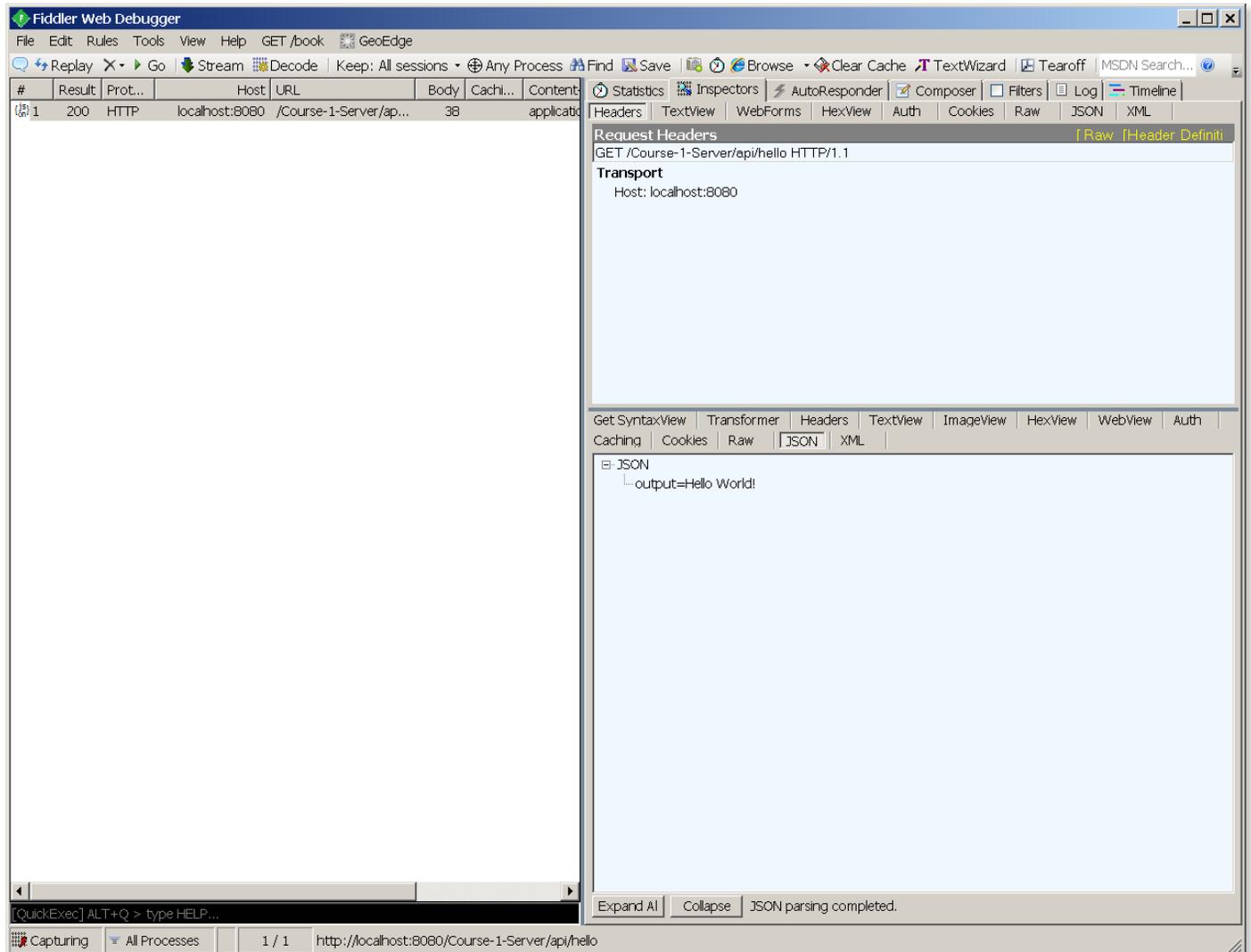


Additionally, you don't have to have a client, such as the one we ran. You can use third-party debugging tools, which we'll talk more about in the third course. However, as a preview, we launched the REST URL with Fiddler, a popular web service debugging tool that is freely available and approved on the TRM. By clicking **Execute**, it issues an HTTP GET for the resource defined by the URL.



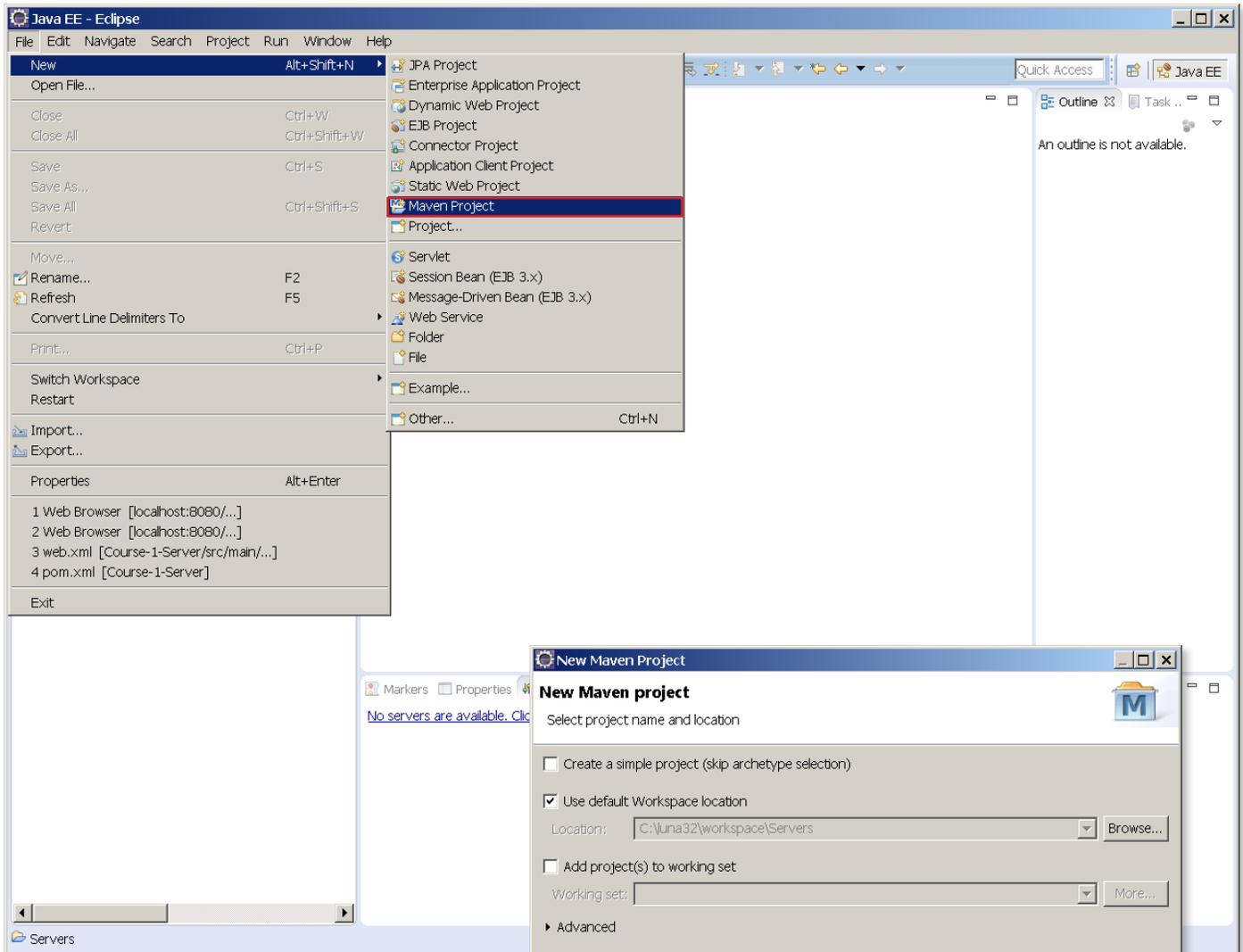
Under inspectors, you can see the response we got is exactly what we expected, namely the Hello message.

Notice how the **JSON** text gets parsed into a variable and a value. So we went from strings to variables. We'll talk more about this and debugging in general in the upcoming course, Part 2.



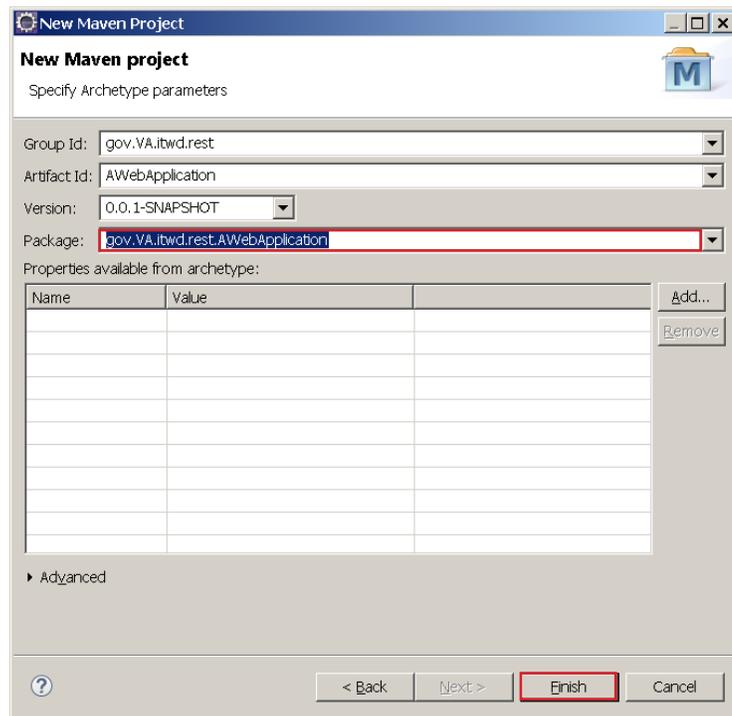
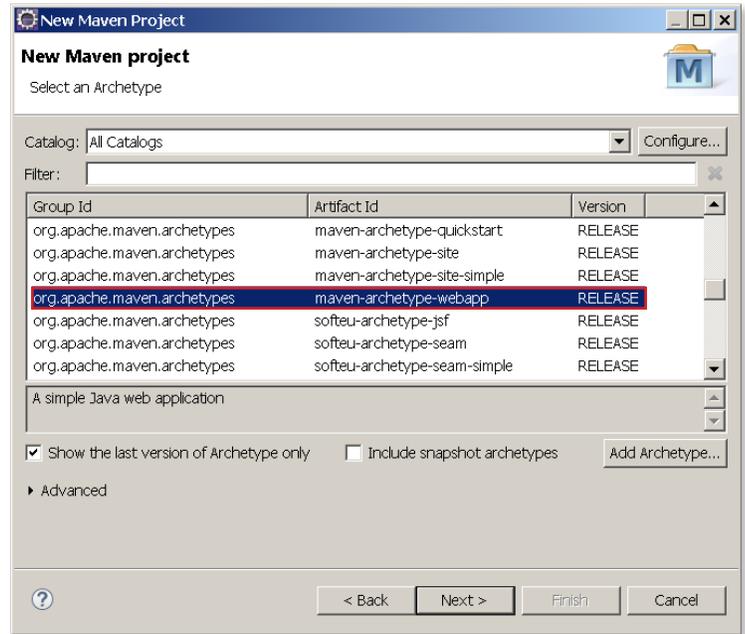
In case you would like to create a blank project to practice with, you can follow these steps to create your own blank project. Keep in mind, there is a lot of customization needed to make the project work as a RESTful web app, but this is the starting point.

1. Click on File, select new, and go to Maven Project.



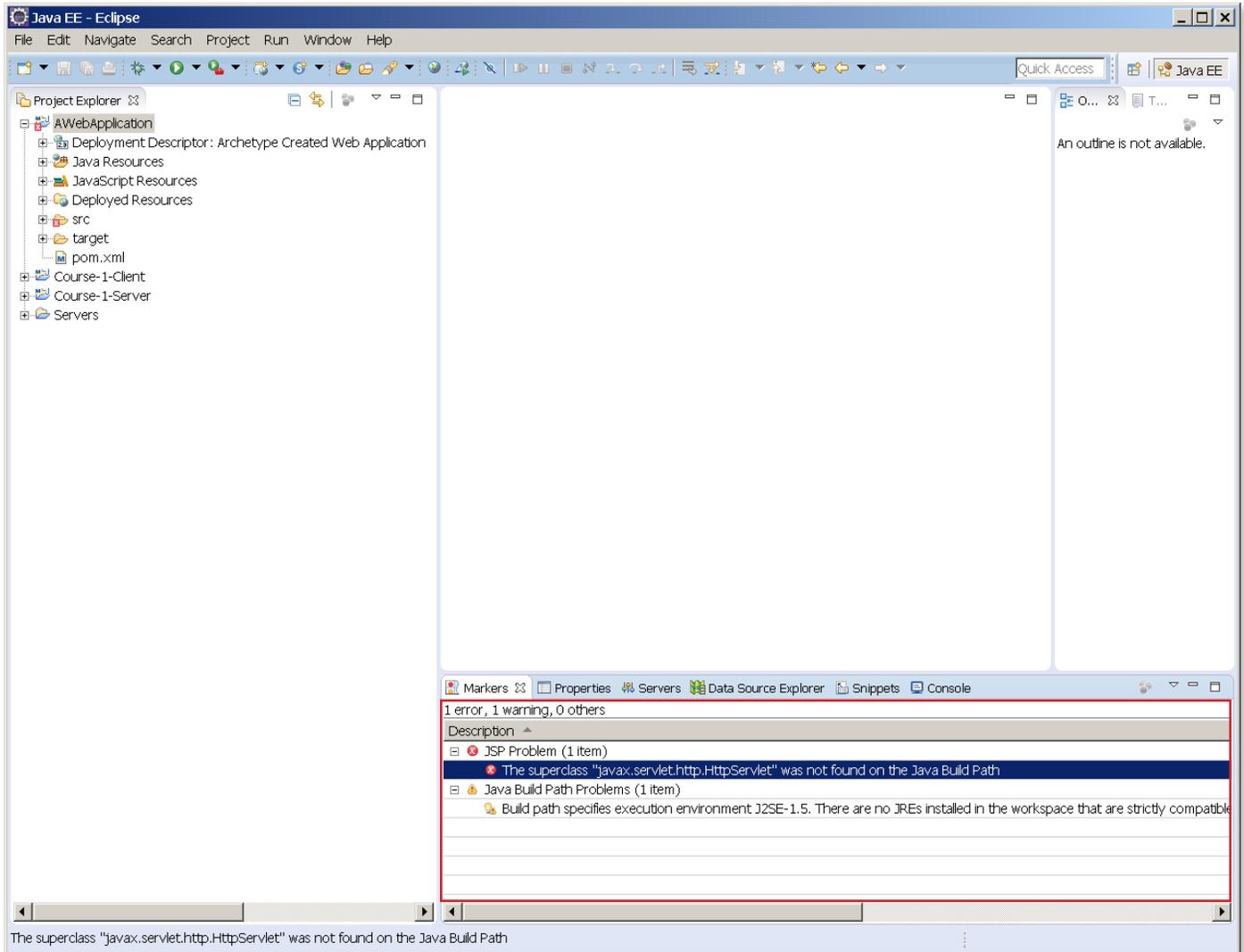
2. Keep the defaults and click Next.

3. Select the **maven-archetype-webapp** and click **Next**. Maven is a powerful tool. While we are going to use a Maven Archetype for web apps, many of its intricacies are outside the scope of these courses.

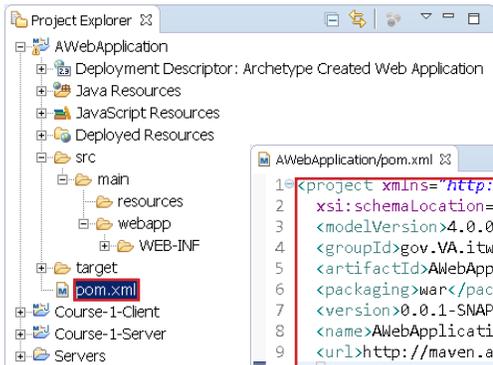


4. Type in a top level package name under **groupID**—all VA apps start with gov.VA. Type in your **application project package**, or **Artifact ID**. You can also choose settings for versioning under **version**, if you plan to distribute the app via Maven. Click on **Finish** to create the application.

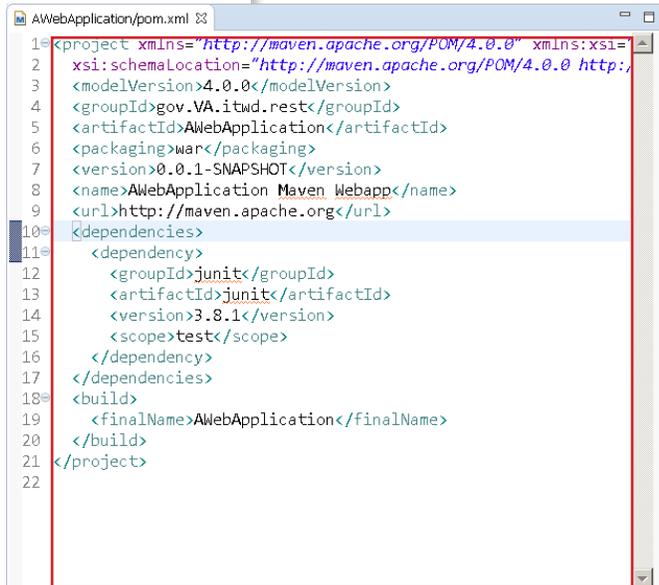
5. Right away, there are a few errors we can address—namely a build error and the need to deal with **JavaServer** pages.



6. Unless you are using JSPs, you can go ahead and delete the **index.jsp** file under **src, main, webapp** which will get rid of your first error.

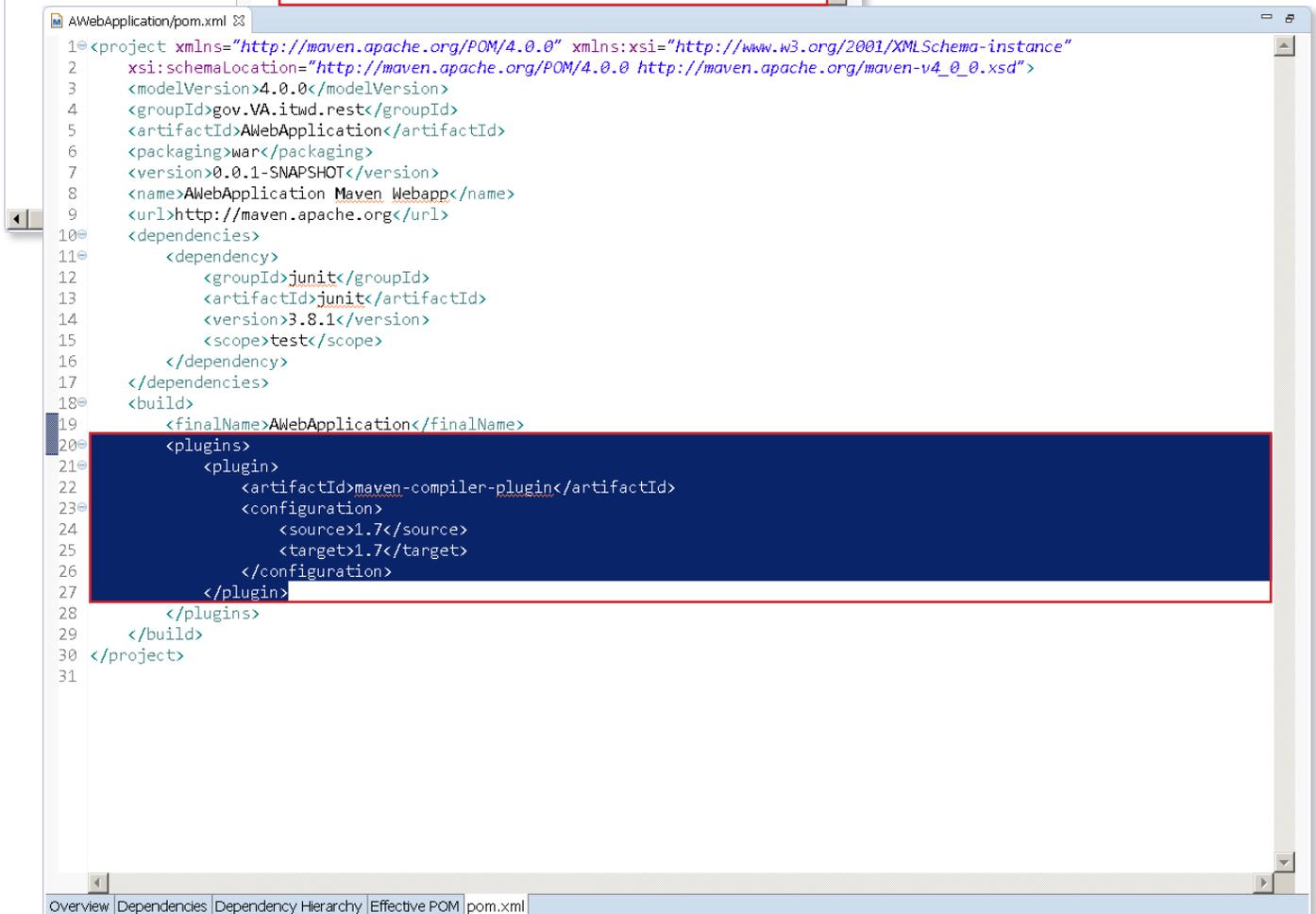


7. The next error is a build error because the Maven POM file has not defined the right target JRE. Locate the **pom.xml** file under the main project folder and double-click to open.

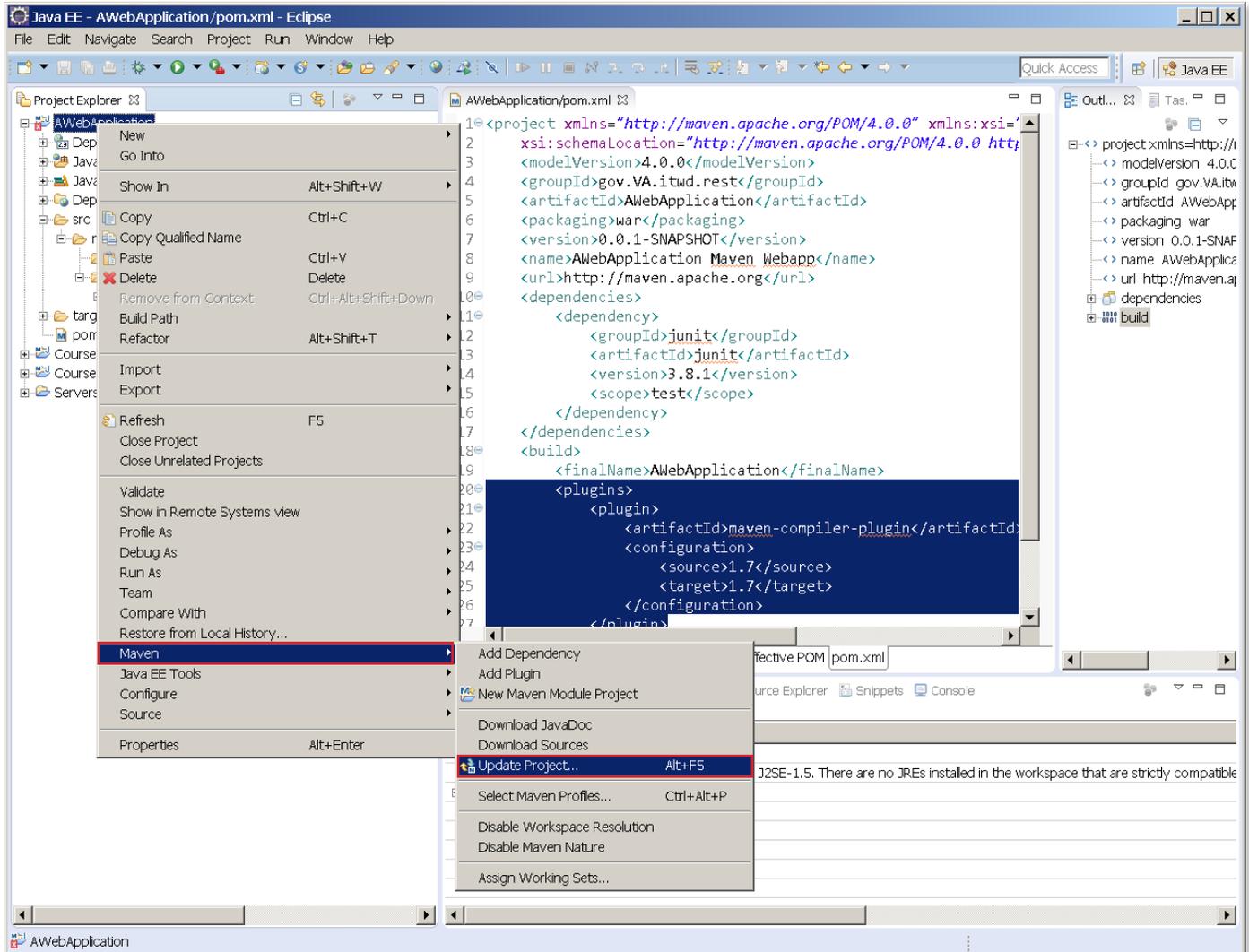


8. Select **pom.xml** at the bottom to see the XML directly. The other tabs provide a graphical editor for the POM, which may be useful to you.

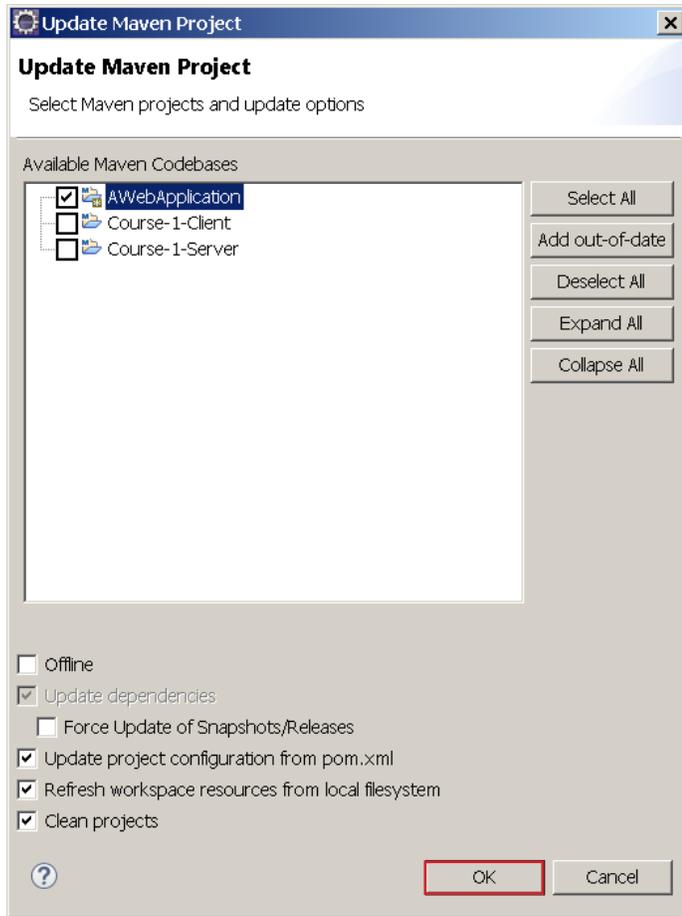
9. In the build tag at the bottom, we want to define the source and target JREs. Add the highlighted text to your **pom.xml** file and save your work.



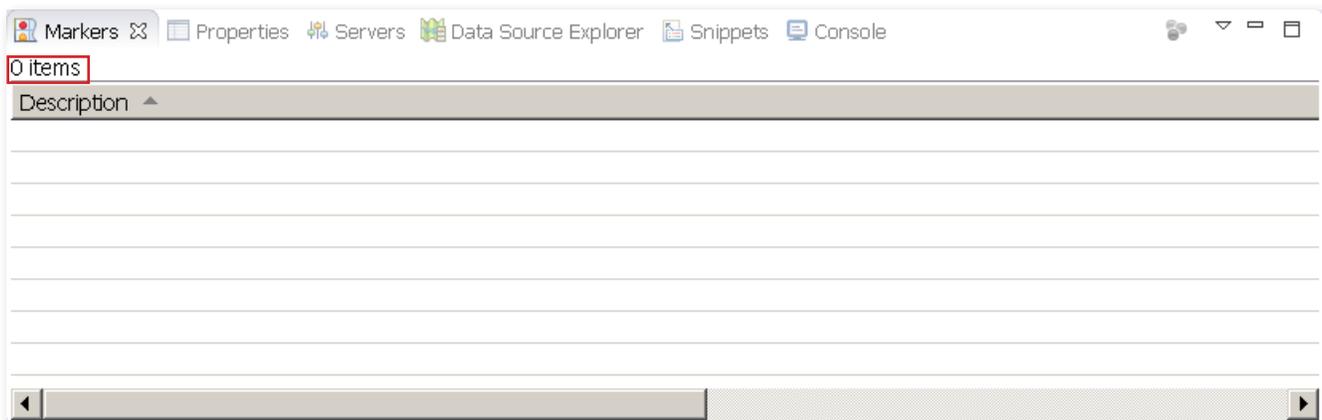
10. Now, we need to let Maven know a change occurred. Select your project, right-click, go to **Maven**, and click on **Update Project**. Alternatively, you can use **Alt+F5** on your keyboard.



11. The default options are ok. Click **OK** to update the project.



12. Notice once the project finishes linking and compiling, there are no errors or warnings. Feel free to practice with this project; you will probably want to add a servlet at a minimum to practice making REST applications.



Additional Resources

This section of the guidebook provides you with resources mentioned in the REST course. Several are websites and the others can be found in the Books 24x7 catalog of the Talent Management System (TMS).

Websites

The One-VA Technical Reference Manual (VA TRM)

<http://trm.oit.va.gov/TRMHomePage.asp> **Note:** This link is only available on VA's Intranet.

Search this site to stay within guidelines for REST vendors and guidelines that have been vetted and approved by VA. This site is also available to search hardware, software, testing tools, applications, etc., to see if that tool is approved, approved with constraints, or unapproved at VA. Do your research here before you start using tools to develop REST APIs.

Eclipse

To download Eclipse to your personal computer visit, <https://www.eclipse.org/>

Eclipse Reference Websites

Visit these user forums for references on how to use REST with Eclipse:

<http://marketplace.eclipse.org/content/rest-client>

<http://wiki.eclipse.org/EclipseLink/Examples/REST/GettingStarted/RestService>

Tomcat 7.0

To download Tomcat 7.0 to your personal computer, visit, <http://www.coreservlets.com/Apache-Tomcat-Tutorial/tomcat7-files/tomcat-7.0.34-preconfigured.zip>.

1. Unzip the the downloaded file into the root folder of the C drive (C:\apache-tomcat-7.0.34).
2. In Eclipse, go to **Open Window**, down to **Preferences**, and select **Server** and then **Installed Runtimes** to create a Tomcat installed runtime.
3. Click **Add** to open the **New Server Runtime** dialog.
4. Select your runtime under **Apache (v7.0)**.
5. Click **Next**.
6. Under **Tomcat Installation Directory** insert the path to your Tomcat installation (For example: C:\apache-tomcat-7.0.34).
7. Click **Finish**.

Books 24x7

There are several electronic books about developing REST APIs available to you in Books 24x7 on the TMS (TMS ID 30086). The list of books includes:

- *Cloud Optimized REST API Automation Framework*
- *The Agile Architecture Revolution: How Cloud Computing, REST-Based SOA, and Mobile Computing Are Changing Enterprise IT*
- *RESTful PHP Web Services: Learn the Basic Architectural Concepts and Steps Through Examples of Consuming and Creating RESTful Web Services in PHP Android Application Development for Dummies*

Five Steps for Accessing Books 24x7

1. Log in to the TMS at <https://www.tms.va.gov>.

2. Enter TMS ID 30086 in the **Browse** text box and select **Go**.

3. Select the **Books 24x7 Referenceware** title from the Catalog Search Results.

4. Select **Continue Course** (If this is the first time you have accessed Books 24x7, the button will read Start Course).

5. Enter your search terms (e.g., "REST, API") in the **Search** text box and select the **Go** button.