

REST API Handbook

This guidebook provides an exercise that coordinates with the *Introduction to Representational State Transfer (REST) with Java, Beginner Part 2* 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

How to Test Your Exercises	8
---	---

Course 2: REST Exercise, 2

Exercise: Creating a REST API with Standard Actions	11
---	----

Accessing Additional Resources	15
---	----

Using the VMware Environment

VMware is a tool offered by VA ITWD that allows you to log into 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 2* course, TMS ID 3878053, 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

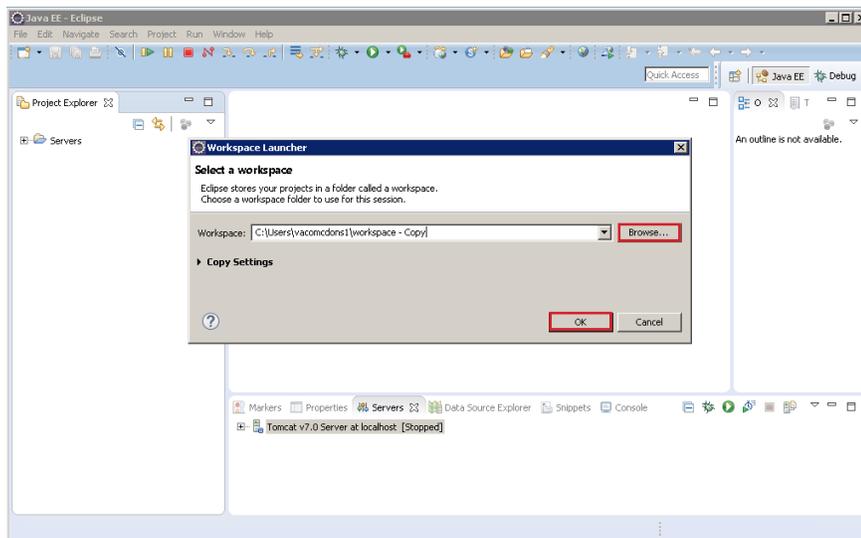
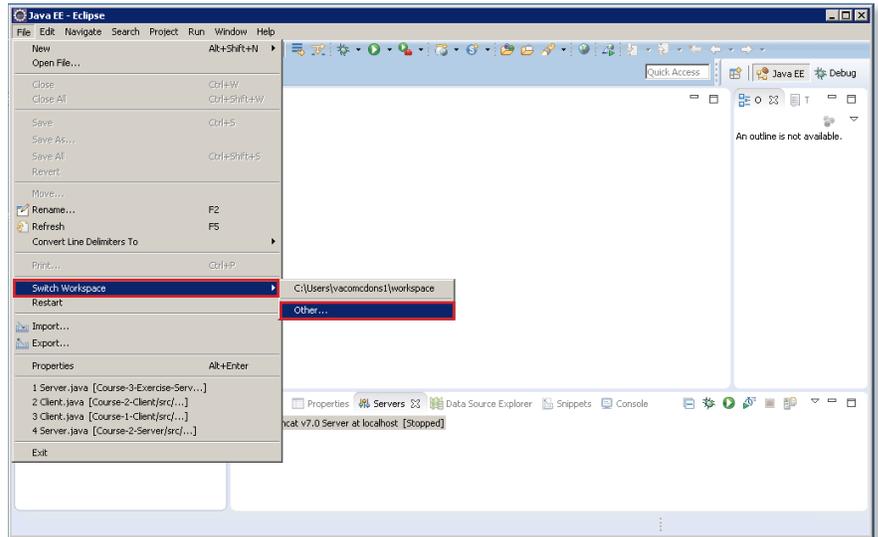
The **File** tab opens to display its menu.

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

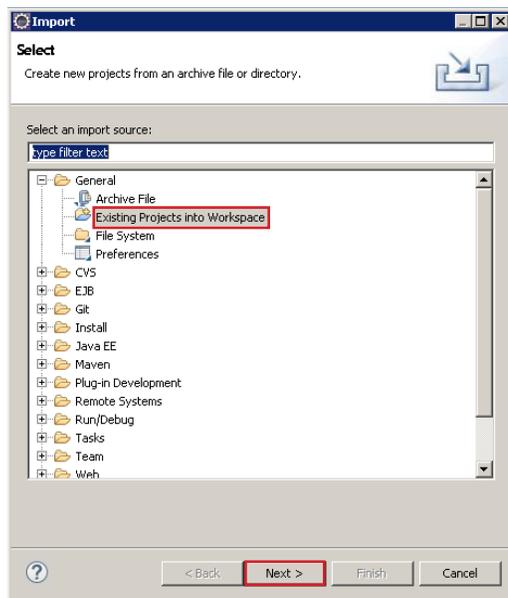
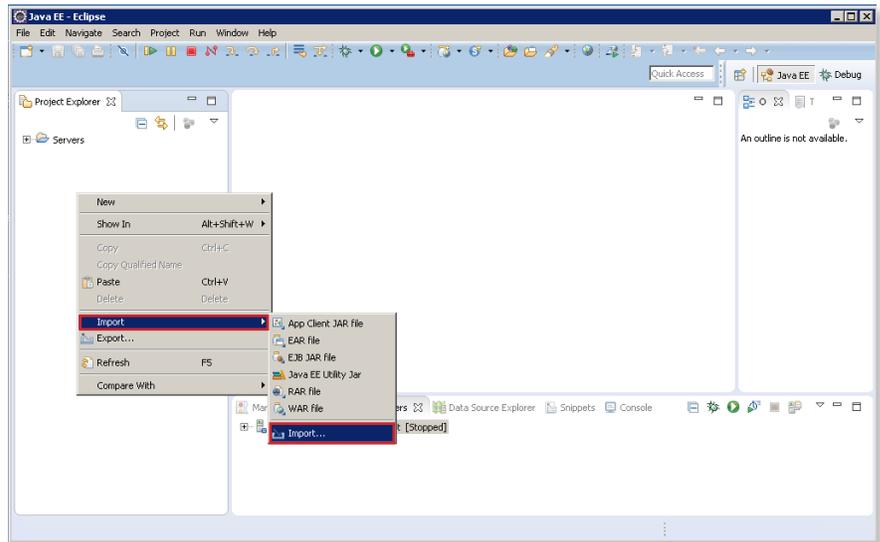
When you want to import an existing REST project into Eclipse, open Eclipse to start the import.

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.

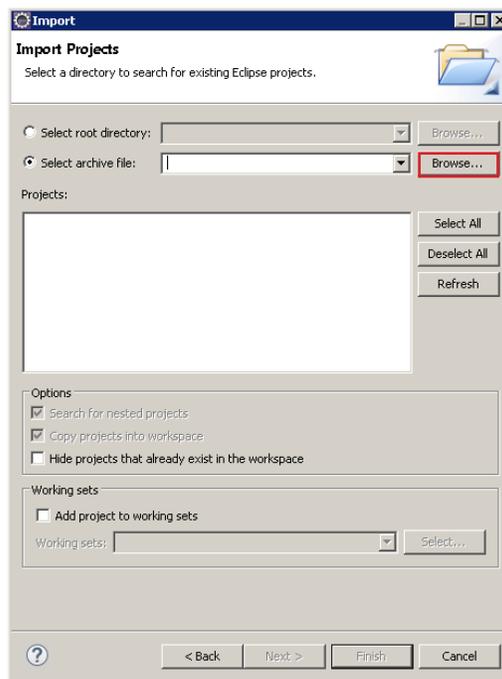


3. Select an Import Source

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

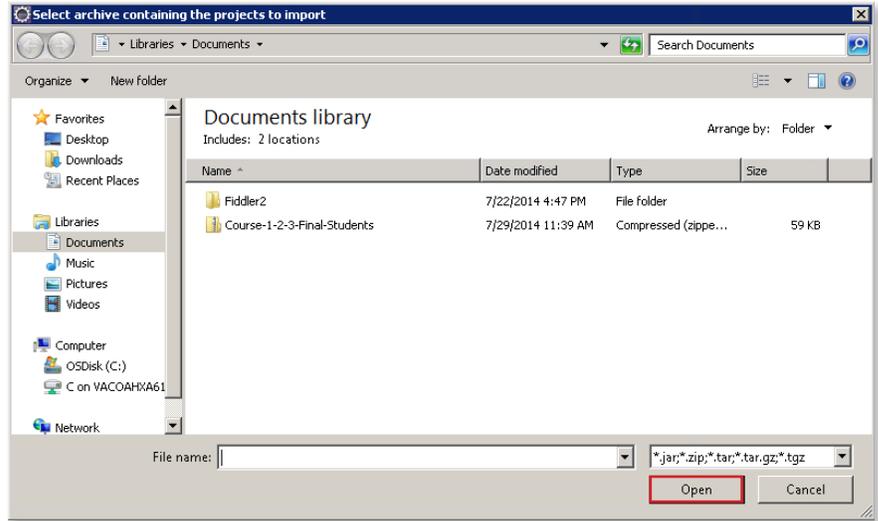
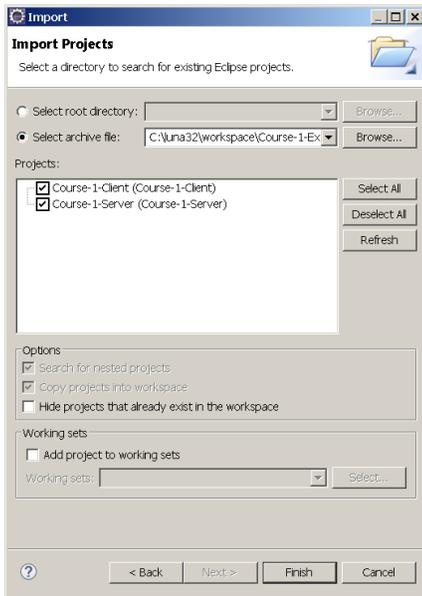
4. Select the Project to Import

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

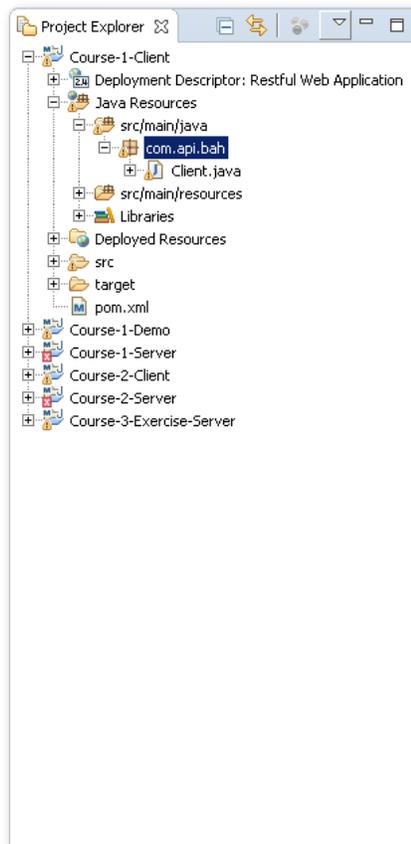


5. Locate on Your Computer Where the Project is Stored

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



This will show you your projects in the zip file.

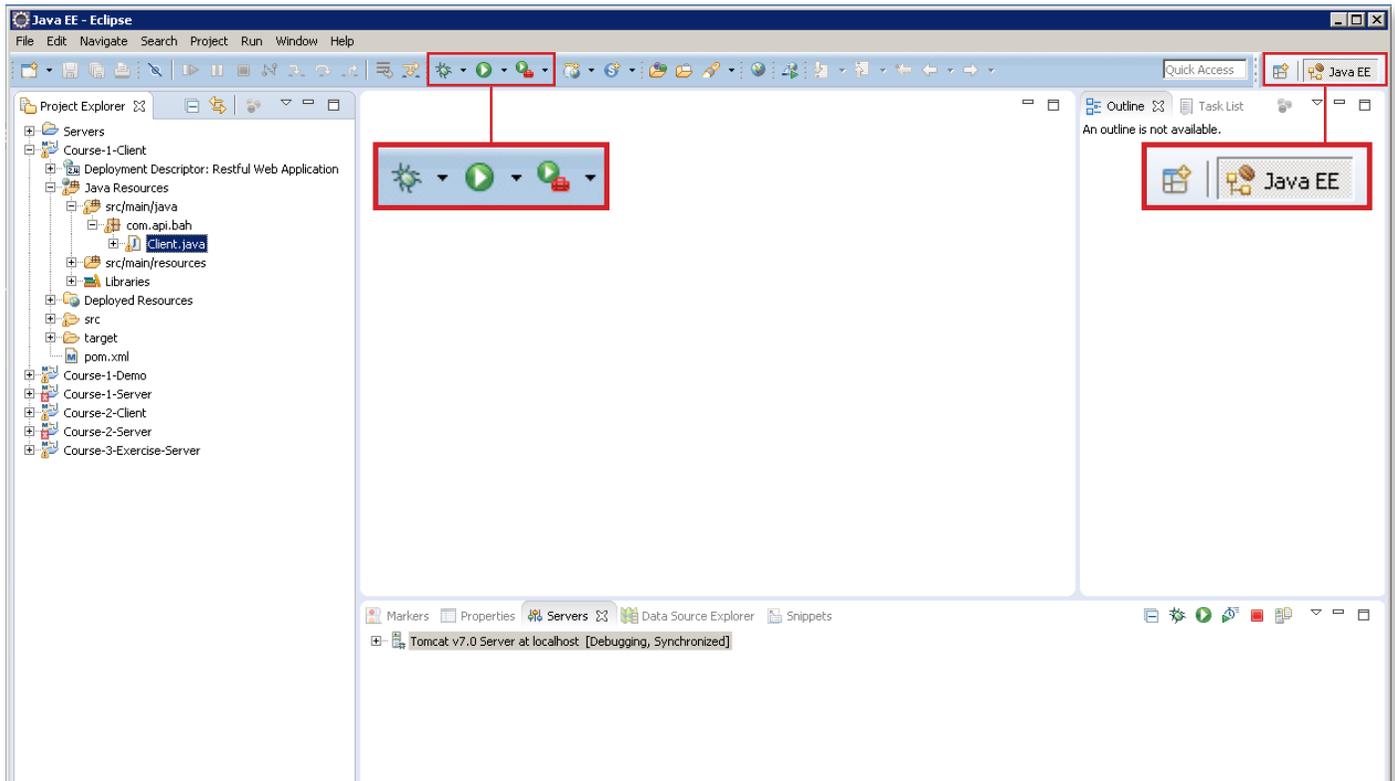


6. View the Imported Project in Project Explorer

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

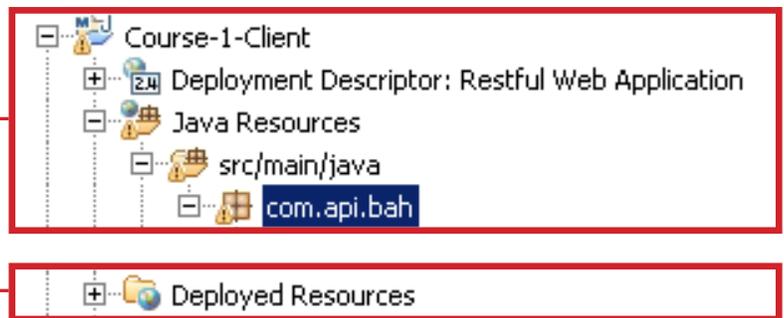
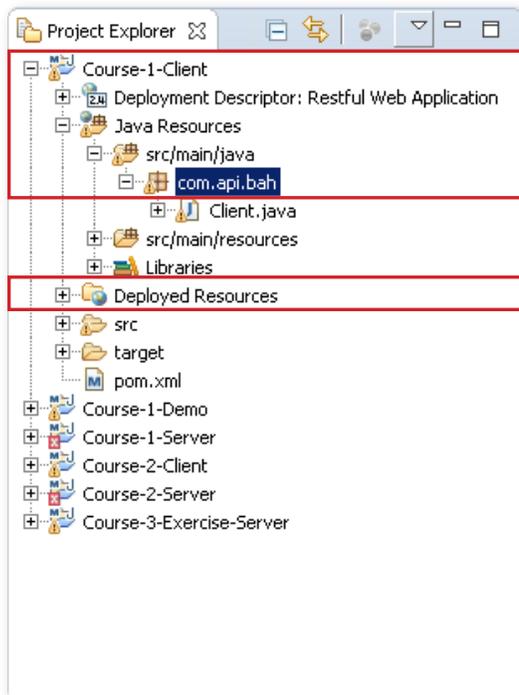
This button provides different views to assist you in completing specific steps while creating REST APIs. This button switches between different perspectives. The perspectives that will be most helpful in your assignments will be the Debug and Java.

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

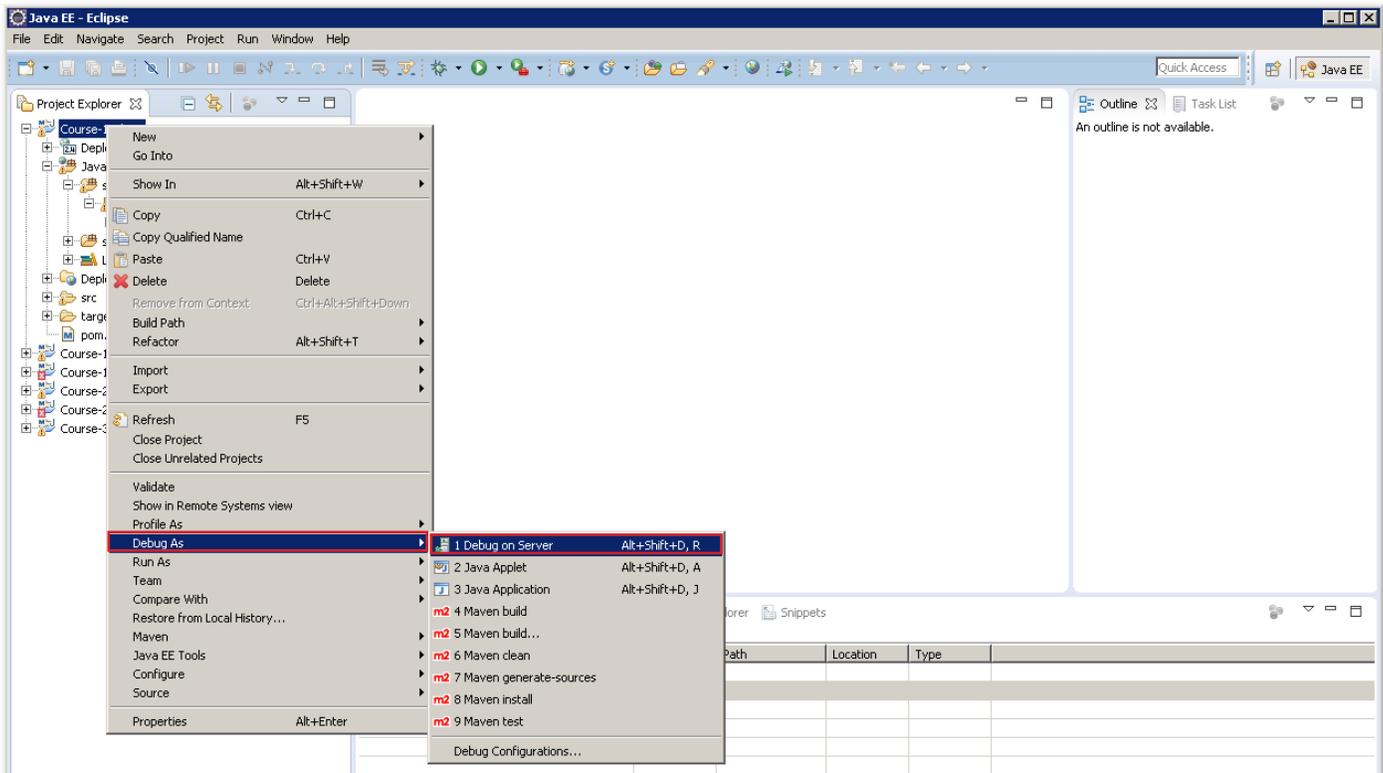
The **src** and **target** folders are in the same level of hierarchy as the **Java Resources** folder and shouldn't be edited or removed. The files in these folders were created to point to specific files necessary for running your program.

How to Test Your Exercises

To test your exercises, you will need to start by debugging the project. This function uses a compiler to put data in an executable package so that the computer will understand it once the application is executed. While you can use the debug button mentioned earlier, we're going to show you through the debugger in the **Project Explorer** pane. The steps below will show you how to test your exercises.

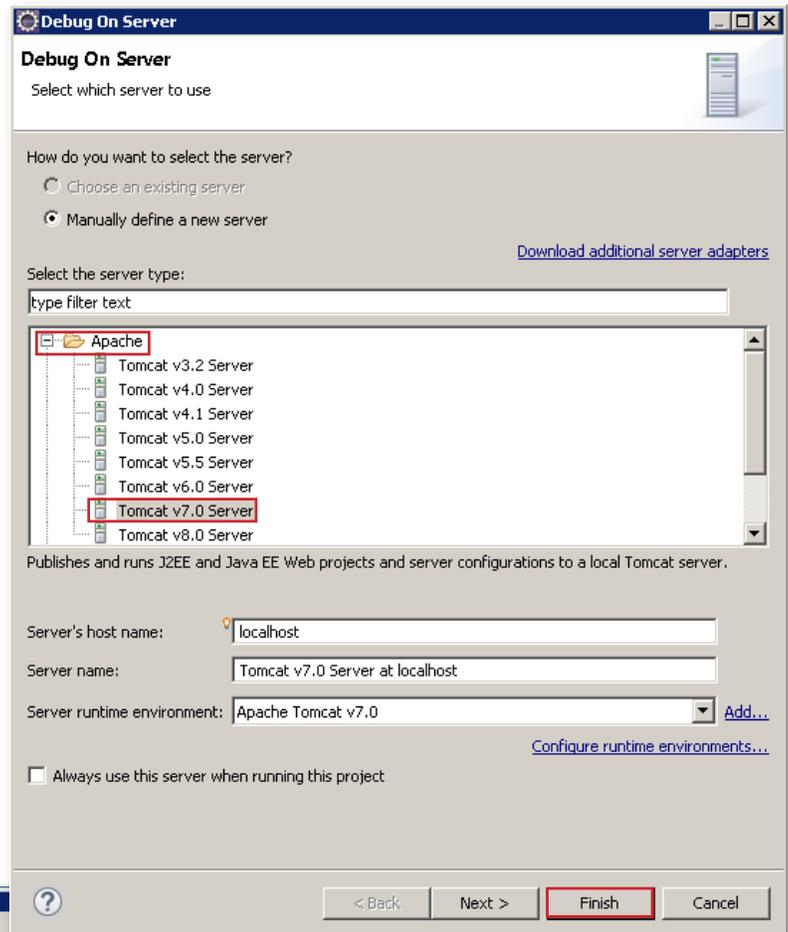
1. Right-click on the Top Folder of the project you're working on

When you right-click on the top folder a sub-menu will open. Navigate down to **Debug As** and select 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.



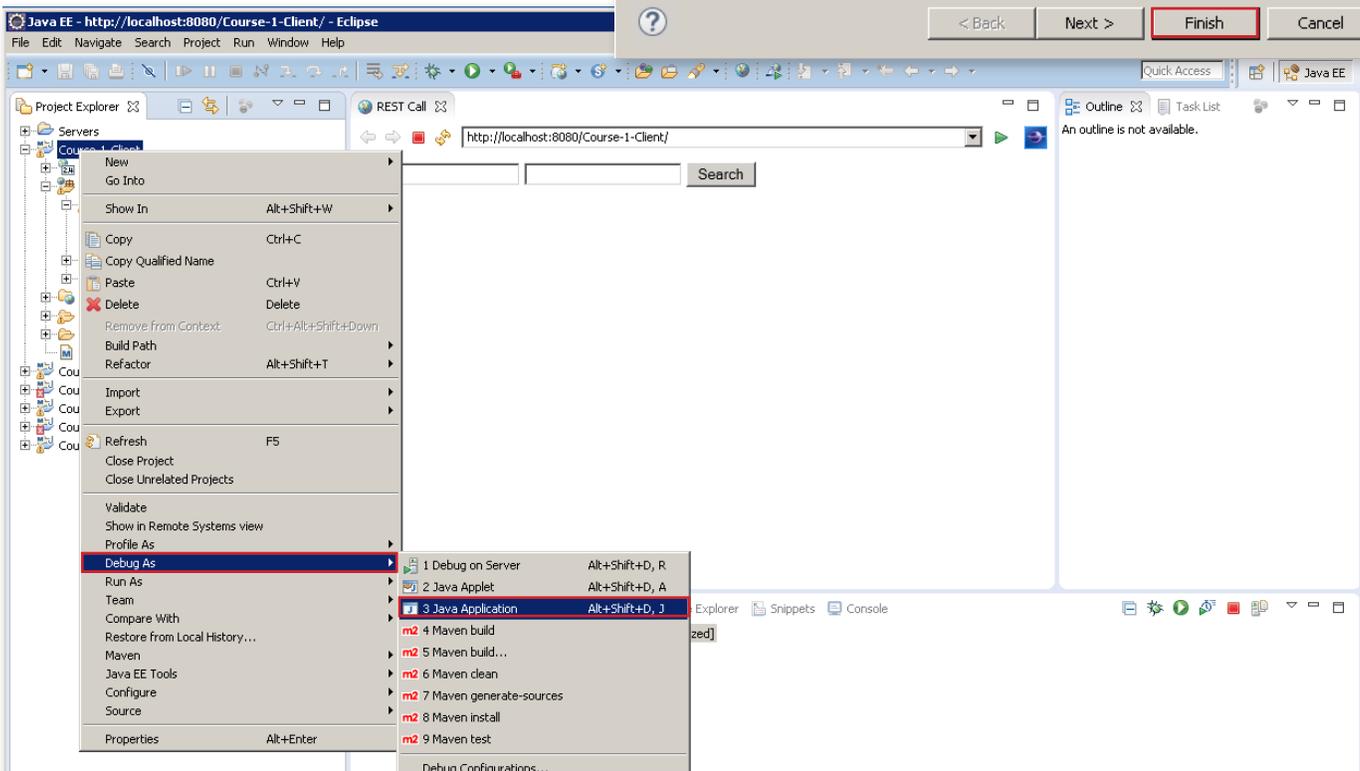
2. Debug On Server

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.



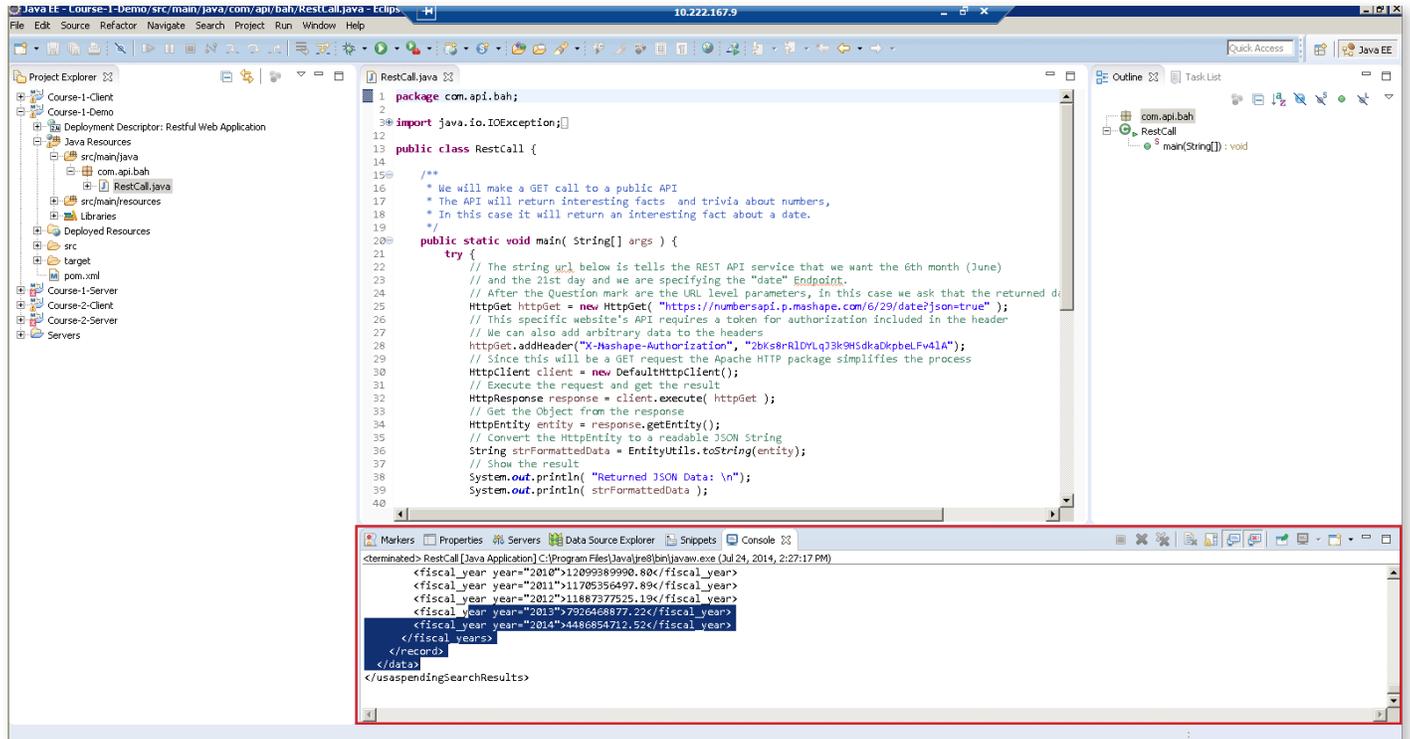
3. Debug As Java Application

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 2: REST Exercise, 2

Exercise: Creating a REST API with Standard Actions

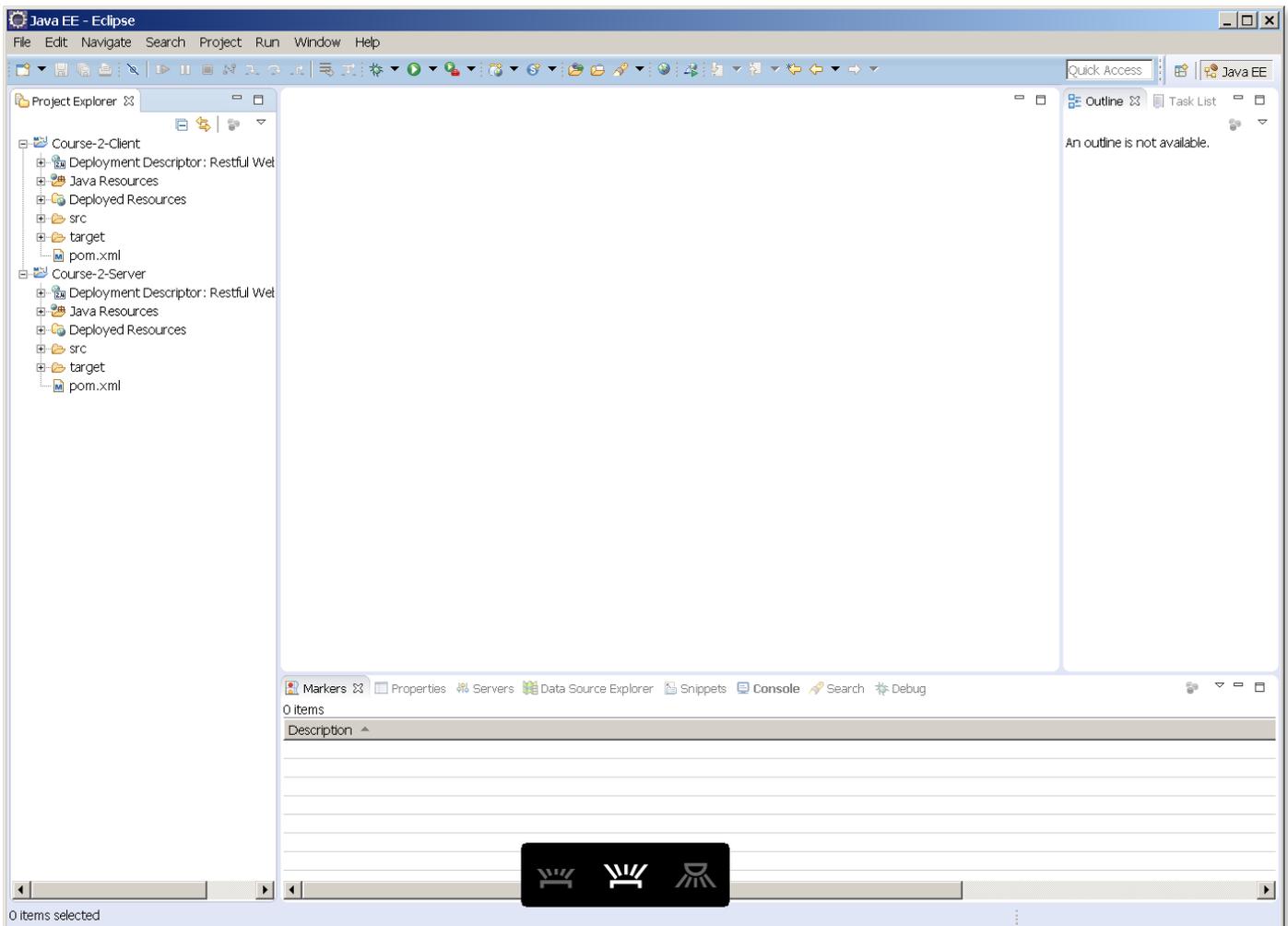
In this second exercise, we're going to ask you to create a more complex API with actions. Good APIs use logical actions. There can be dozens of actions. In this exercise, you will use the four standard actions, GET, PUT, POST, and DELETE. (You may notice that the code in this exercise is in a similar format as in Course One. You should be familiar with this code. However this time, you'll need to complete the code.)

Remember, this API has a list of keys that are mapped to values. Every key is unique and consists of a string. Every value is a corresponding string. Values don't need to be unique. Like Exercise One, we'll be using the Jersey Library to implement the JAX-RS interfaces. To begin this exercise, you will need to import the following existing REST API project: Course-2.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.

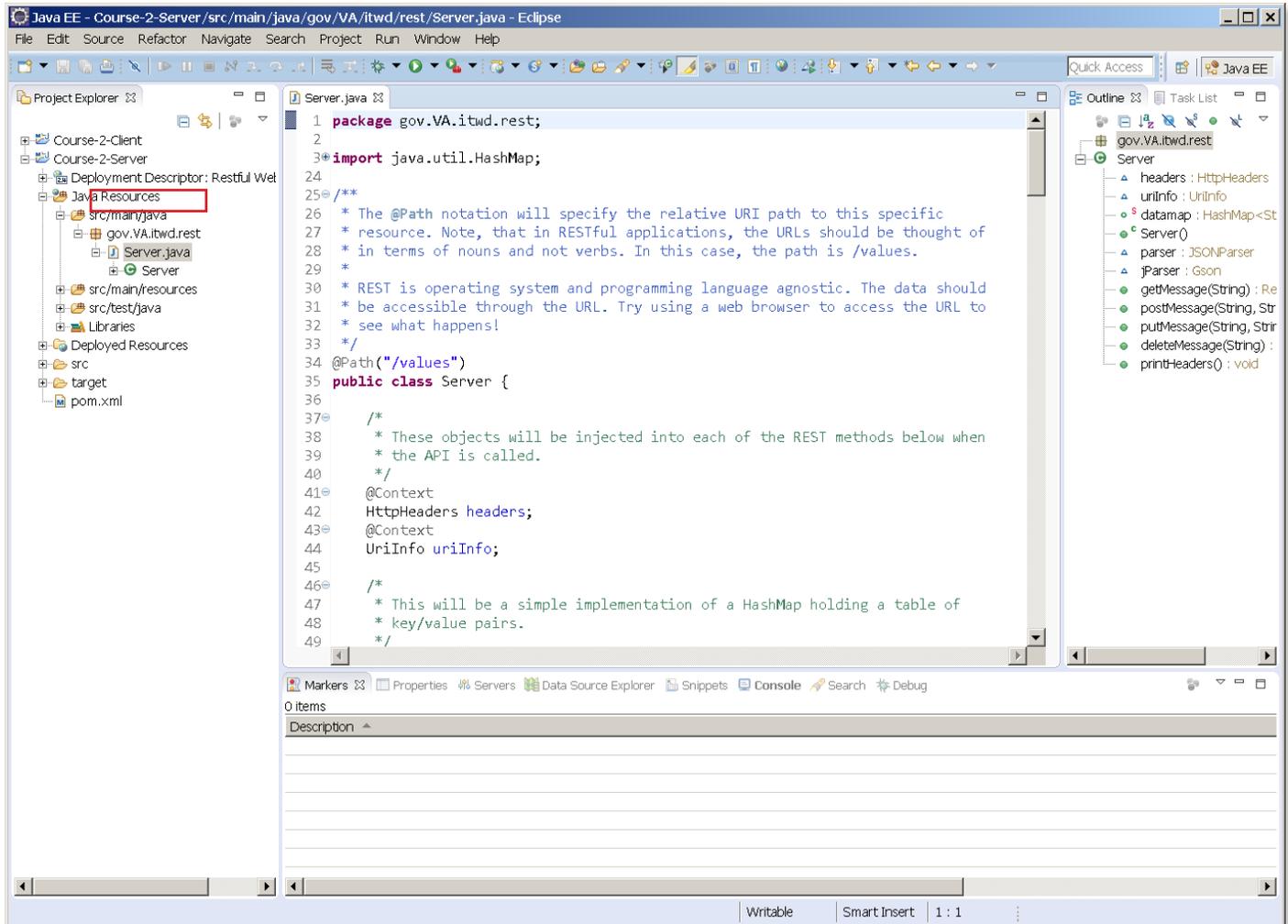
1. Import the Course 2 Files into the Package Explorer

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



2. Expand the Java Resources Folder to the Server.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.rest** file. After expanding the **gov.VA.itwd.rest** file, locate the **Server.java** file. Your screen should look similar to the screen below. Double-click the **Server.java** file to open the exercise on your screen. Your screen will open and code will be displayed as in image 3.



3. "Get" Standardized Action

Scroll down until you see the block notes about **@ GET**. Locate the line comment about adding an **@Path** and **@Produces**. Your exercise for the GET standardized action starts here. Follow the line comments and add the necessary Java when asked.

Hint: It might only be necessary to uncomment a couple of lines of code!

```
Server.java
72  /*
73  * Add the @GET to let the server know this is the method to be called when
74  * a GET request is issued.
75  *
76  * Add @PATH("/{id}") to inject the path id.
77  *
78  * Add @Produces("application/json") to identify the output type.
79  *
80  * Notice in the method parameters, the @PathParam tag is used to inject the
81  * value of the id defined by the path.
82  */
83  // @GET
84  // @Path("/{id}")
85  // @Produces("application/json")
86  public Response getMessage(@PathParam("id") String id) {
87
88      // Print out the HTTP Request.
89      printHeaders();
90
91      String value = null;
92
93      /*
94      * Get the value with the given id. If the id was not found, let the
95      * requester know by returning a 404 not found HTTP response and the
96      * client side should handle properly displaying the message.
97      */
98
99      // Server Response Status in more detail in Course 3. If the
100     // id is found then the request was successful, hence we will send
101     // HTTP 200 "OK" message by default unless the id is not found.
```

```
Server.java
128  /*
129  * Add @POST to let the server know this is the method to respond to POST
130  * requests.
131  *
132  *
133  * Add @Consumes({ MediaType.APPLICATION_FORM_URLENCODED }) to identify how
134  * the data is encoded.
135  *
136  * Notice the @FormParam tag in the parameters is used to inject the id from
137  * the request header.
138  */
139  // @POST
140  // @Consumes({ MediaType.APPLICATION_FORM_URLENCODED })
141  public Response postMessage(@FormParam("id") String id,
142                             @FormParam("name") String name) {
143
144      // Print out the HTTP request.
145      printHeaders();
146
147      /*
148      * In future work, you will want to check the incoming parameters to
149      * verify that they are safe before attempting to use them.
150      *
151      * If the id is already in the HashMap, modify the associated value.
152      * Otherwise, let the requester know the resource was not found. We will
153      * send back OK/200 by default, unless the item is not found.
154      */
155      ResponseBuilder response = Response.status(Status.OK);
156      if (dataman.containsKey(id)) {
```

4. "POST" Standardized Action

Scroll down until you see the block notes about **@ POST**. Locate the line comment about production code. Your exercise for the POST standardized action starts here. Follow the line comments and add the necessary Java when asked.

5. "PUT" Standardized Action

Scroll down until you see the block notes about the **@ PUT**. Locate the line comment about a path. Your exercise for the PUT standardized action starts here. Follow the line comments and add the necessary Java when asked.

```
Server.java
172  /*
173  * Add the @PUT to identify the method as before.
174  *
175  * Add @Path("/{id}/{name}") to identify the path.
176  *
177  * Notice the @PathParam tag is used to inject the id and name in the
178  * parameters.
179  *
180  * Add @Consumes({ MediaType.APPLICATION_FORM_URLENCODED }) to identify how
181  * the data is encoded.
182  */
183  // @PUT
184  // @Path("/{id}/{name}")
185  // @Consumes({ MediaType.APPLICATION_FORM_URLENCODED })
186  public Response putMessage(@PathParam("id") String id,
187                            @PathParam("name") String name) {
188
189      // Print out the HTTP request.
190      printHeaders();
191
192      /*
193      * Respond to the request, with OK/200 status to tell the user the
194      * request was successful by default.
195      */
196      ResponseBuilder response = Response.status(Status.OK);
197
198      /*
199      * Add the Name and Title in the REST request to the HashMap. Most
200      * often, you will want to check whether these are valid entries here
```

```
Server.java
218 /**
219  * The @DELETE request will ask the server to delete an existing entry. The
220  * server is in charge of checking whether the entry exists and if it can be
221  * deleted.
222  *
223  * @return
224  */
225
226 /**
227  * Add the @DELETE tag to identify the method.
228  *
229  * Notice how @Path and @PathParam are used here as previously.
230  */
231 // @DELETE
232 // @Path("/{id}")
233 public Response deleteMessage(@PathParam("id") String id) {
234
235     // Print out the HTTP request.
236     printHeaders();
237
238     /*
239     * Setting response to OK/200 by default. Subsequent logic will set it
240     * depending on the status of the request.
241     */
242     ResponseBuilder response = Response.status(Status.OK);
243
244     // If the id exists, remove it from the HashMap.
245     if (datamap.containsKey(id)) {
246         datamap.remove(id);
```

6. "DELETE" Standardized Action

Scroll down until you see the block notes about the **@ DELETE**. Locate the line comment about **Path {id}**; your exercise the for **DELETE** standardized action starts here. Follow the line comments and add the necessary Java when asked.

7. Debug and Test Your REST API

Follow the steps to 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), then run the client portion with Debug As Java Application. Feel free to experiment with the client to try out different scenarios. We'll talk more about this and debugging in general in the upcoming course, Part 3.

Tip: Try accessing your API via a web browser, such as Firefox—do you get the same results?

```
Server.java
218 /**
219  * The @DELETE request will ask the server to delete an existing entry. The
220  * server is in charge of checking whether the entry exists and if it can be
221  * deleted.
222  *
223  * @return
224  */
225
226 /**
227  * Add the @DELETE tag to identify the method.
228  *
229  * Notice how @Path and @PathParam are used here as previously.
230  */
231 // @DELETE
232 // @Path("/{id}")
233 public Response deleteMessage(@PathParam("id") String id) {
234
235     // Print out the HTTP request.
236     printHeaders();
237
238     /*
239     * Setting response to OK/200 by default. Subsequent logic will set it
240     * depending on the status of the request.
241     */
242     ResponseBuilder response = Response.status(Status.OK);
243
244     // If the id exists, remove it from the HashMap.
245     if (datamap.containsKey(id)) {
246         datamap.remove(id);
```

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.