**How To Develop**

**RESTFul Web Services (JAX-RS)**

**Using Jersey, Eclipse and Glassfish**

**From the Ground Up**

**Part 1**

**TRADOC Capabilities Manager -**

**Army Training Information System (TCM-ATIS)**

**System Interface Services (SIS)**

Contract: W911S0-12-C-0036

**4 February 2013**



**Prepared for:**

U.S. Army Training Support Center (ATSC)

Attn: TCM-ATIS

Building 3308

Fort Eustis, VA 23604-5166

**Submitted by:**



**ZCSolutions, LLC**

1600 Tysons Boulevard, Suite 1150

McLean, VA 22102



**Revision Record**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Description** | **Date** | **Submitter** |
| 1.0 | Initial draft | 4 Feb 2013 | B. Craig |
| 1.1 | Minor corrections and continue w/Task 14, Mis-used uri pattern | 8 Feb 2013 | B. Craig |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1 Purpose 5](#_Toc348080370)

[2 Tools Installation Setup and Configuration 5](#_Toc348080371)

[2.1 Glassfish Installation 5](#_Toc348080372)

[2.2 Create AS\_HOME User Environment Variable 6](#_Toc348080373)

[2.3 Upgrade Jersey 6](#_Toc348080374)

[2.4 Eclipse Installation 7](#_Toc348080375)

[2.5 Eclipse Configuration 7](#_Toc348080376)

[2.5.1 Configure Eclipse to use Glassfish 8](#_Toc348080377)

[2.5.2 Eclipse Plug-ins 10](#_Toc348080378)

[2.5.3 Create Jersey User Library 12](#_Toc348080379)

[3 Project Description 13](#_Toc348080380)

[3.1 Requirement 13](#_Toc348080381)

[3.2 Data and Data Storage 13](#_Toc348080382)

[3.2.1 Understanding the requirement 13](#_Toc348080383)

[3.2.2 White Board URI pattern 14](#_Toc348080384)

[3.3 Developing the Project 14](#_Toc348080385)

[3.3.1 Task 1 – Create StateResource Class 18](#_Toc348080386)

[3.3.2 Task 2 – Register Service in web.xml 20](#_Toc348080387)

[3.3.3 Task 3 – Test Run on Glassfish 21](#_Toc348080388)

[3.3.4 Task 4 – Create Database POJO Using JAXB Annotations 23](#_Toc348080389)

[3.3.5 Task 5 – Implement the GET method using Resource POJO 24](#_Toc348080390)

[3.3.6 Task 6 – Generate List of Resources 26](#_Toc348080391)

[3.3.7 Task 7 – GET a Specific Resource 26](#_Toc348080392)

[3.3.8 Task 8 – Mock Database 27](#_Toc348080393)

[3.3.9 Task 9 – Modify Get Methods to Query a Database 28](#_Toc348080394)

[3.3.10 A Quick Introduction to REST Client Test Tools 29](#_Toc348080395)

[3.3.11 Task 10 – Implement the PUT Method 32](#_Toc348080396)

[3.3.12 Task 11 – Implement the POST Method 34](#_Toc348080397)

[3.3.13 Task 12 - Implement the DELETE Method 36](#_Toc348080398)

[3.3.14 Task 13 – The OPTIONS HTTP Method 37](#_Toc348080399)

[3.3.15 Task 14 – Links to Detail Resources 40](#_Toc348080400)

[4 Principles of REST 40](#_Toc348080401)

[5 HTTP Methods 40](#_Toc348080402)

[6 JAX-RS Annotations 40](#_Toc348080403)

[7 HTTP Response Codes 42](#_Toc348080404)

[8 Mock Database Source Code 42](#_Toc348080405)

# Purpose

This document shows the necessary steps to develop a Java REST Web Service using Jersey, Eclipse and Glassfish V 2.1.1. You will learn how to install and configure Glassfish V 2.1.1, Eclipse Juno and Jersey 1.17 and develop a simple REST Web Service that implements the GET, PUT, POST, and DELETE HTTP methods.

# Tools Installation Setup and Configuration

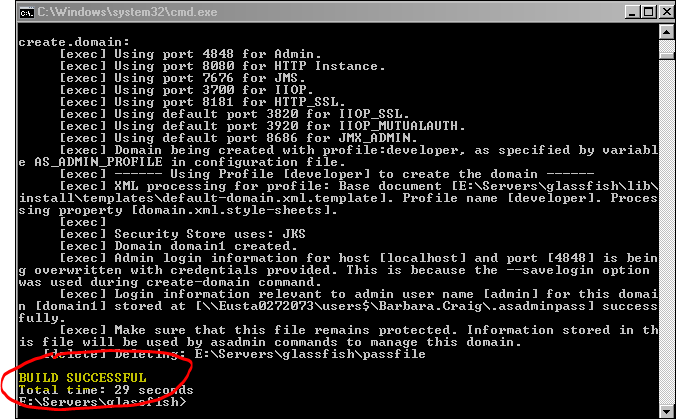
All installation files and any other demonstration files needed for this demonstration can be found in the **T:\DeveloperTools** directory. If you do not have a T drive mapping these files can be accessed [\\Eusta0272002\01ATISD\DeveloperTools](file:///\\Eusta0272002\01ATISD\DeveloperTools). If access to the network drives are is not available download [**Glassfish**](https://glassfish.dev.java.net/downloads/v2.1.1-final.html) [here](https://glassfish.dev.java.net/downloads/v2.1.1-final.html) and [**Eclipse**](http://www.eclipse.org/downloads) from [here](http://www.eclipse.org/downloads).

## Glassfish Installation

**Installation Prerequisite – JAVA JDK 1.6 and ANT must be installed and in your Environment PATH**

1. Download or copy the glassfish\_2\_1\_1.zip to your E:\Drive and unzip it to a folder such as **E:\Servers**.
2. Open a command prompt and navigate to the **E:\Servers\glassfish** directory or where unzipped
   1. Run the following command:

**ant -f setup.xml**

****

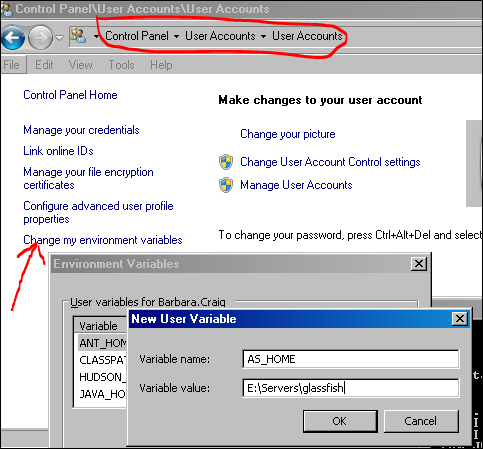
* 1. Be sure you have BUILD SUCCESSFUL displayed for a good installation.
  2. You may close this window

1. Glassfish is now installed.

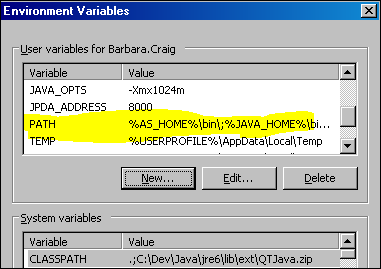
## Create AS\_HOME User Environment Variable

The ***AS\_HOME*** environment variable is the location of the Glassfish installation. The path to the /glassfish/bin directory must also be added to the PATH environment variable.

1. In Windows 7 Navigate to Control Panel 🡪 User Accounts 🡪 User Accounts and click 🡪 **Change My environment Variables.**
2. Under User Variables Click 🡪 ***New*** and enter **AS\_HOME** and the directory you installed glassfish in as below.



1. Scroll down to find the PATH environment variable click 🡪***Edit*** and add %AS\_HOME%\bin; then click 🡪Save.



## Upgrade Jersey

Glassfish V 2.1.1 comes with Jersey version 1.0.3.1 we need to upgrade to a later version that supports Declarative Security. The Upgrade files are located in the T:\DeveloperTools\Jersey directory.

Using File Explorer navigate to the <GLASSFISH\_HOME>\lib directory.

1. Create a backup directory as in <GLASSFISH\_HOME>\lib\backup
   1. Move the four following jar files to the backup directory

|  |
| --- |
| **Jars to Move to Backup** |
| jackson-asl-0.9.4.jar |
| jersey-bundle-1.0.3.1.jar |
| jettison-1.0.1.jar |
| jsr311-api-1.0.jar |

1. Copy the jersey-bundle-1.17.jar file from the T:\DeveloperTools\Jersey directory to the <GLASSFISH\_HOME>\lib directory. This new jar has jsr311, jersey, json, and jetty bundled in one jar.

|  |
| --- |
| **Copy to glassfish\lib** |
| jersey-bundle-1.17.jar |

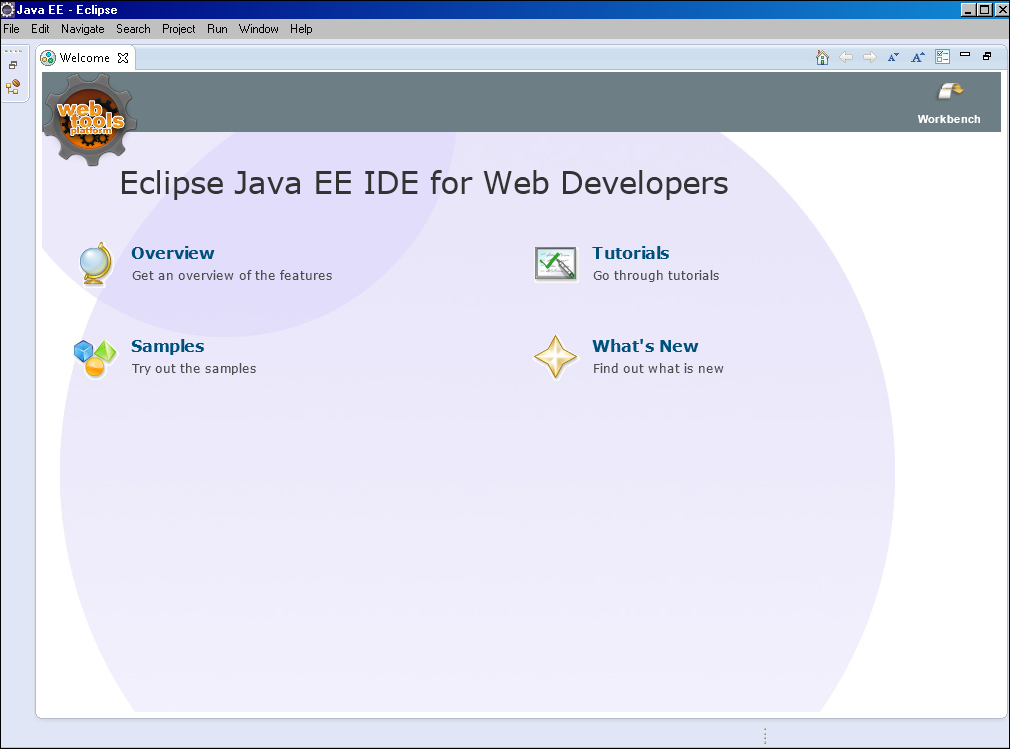
## Eclipse Installation

1. Download or copy the eclipse-jee-juno-SR1-win32.zip into your E:\Drive.
2. Unzip it to a folder such as E:\IDE.
3. Eclipse is now installed.

## Eclipse Configuration

Open eclipse - Navigate to the eclipse installation directory and double click 🡪 eclipse.exe. Change the workspace location. Such as E:\IDE\workspace and press OK

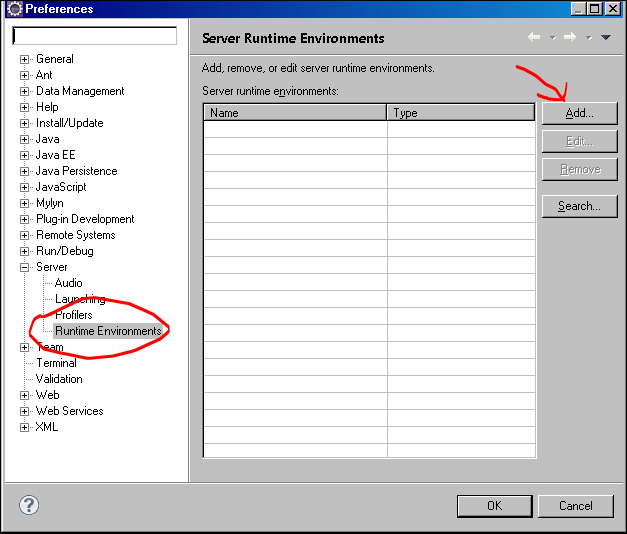
**\*\* NOTE \*\* DO NOT** use the VM workspace location [\\Eusta0272073\users$\YourName\workspace](file:///\\Eusta0272073\users$\YourName\workspace)



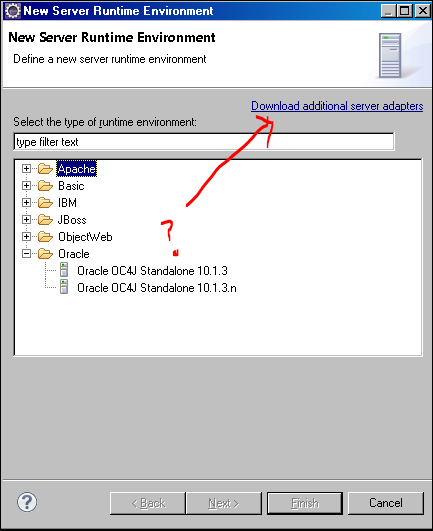
The Eclipse workbench should appear.

### Configure Eclipse to use Glassfish

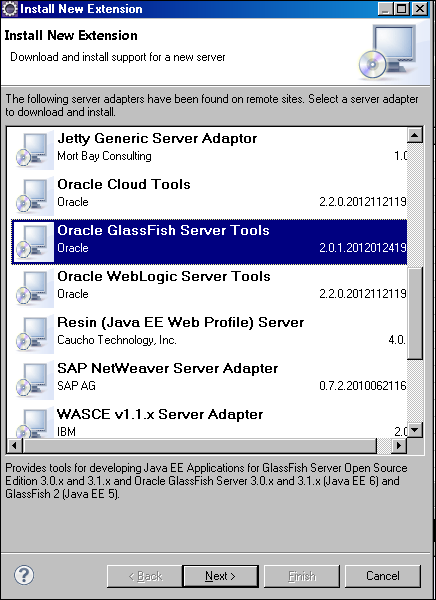
1. Using the Menu navigate to Window 🡪 Preferences
   1. In the preferences window expand the *Server* tree and select ‘*Runtime Environments’*. Then click🡪Add



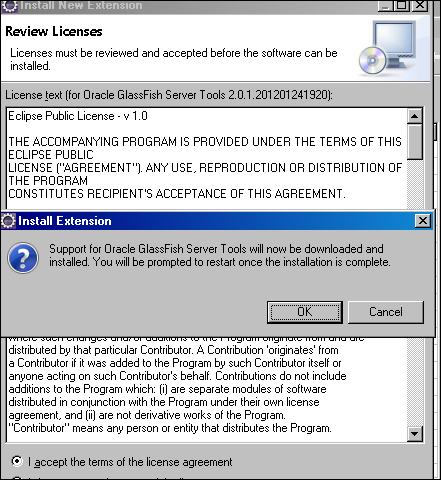
Don’t see Glassfish 2.1 Java EE? Click *🡪 Download additional server adapters.*



Scroll down to Oracle Glassfish Server Tools highlight and click 🡪 Next



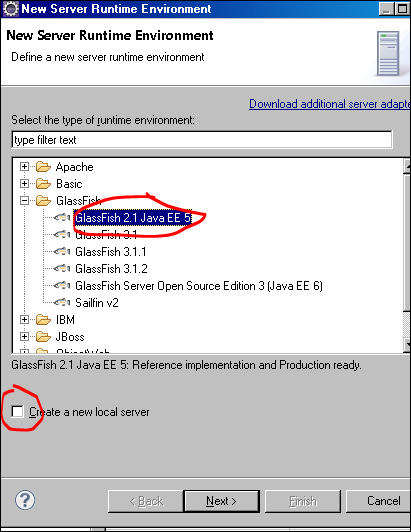
Accept the agreement license and click 🡪*Finish*. Then Click 🡪OK to install



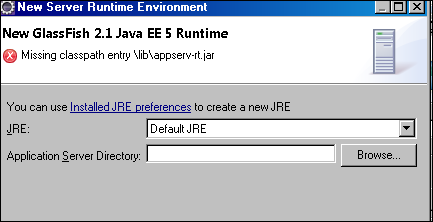
Wait for the installation to complete. Then Click 🡪Yes to Restart Eclipse.



1. Use the Menu navigate to Window 🡪 Preferences
   1. In the preferences window expand the *Server* tree and select ‘*Runtime Environments’*. Then click🡪Add
   2. Scroll down to Glassfish 2.1 Java EE5
   3. **DO NOT CHECK** Create a new local server. Click 🡪Next



* 1. Tell Eclipse where Glassfish is … Click 🡪Browse and find it, click 🡪OK and then click 🡪Finish.



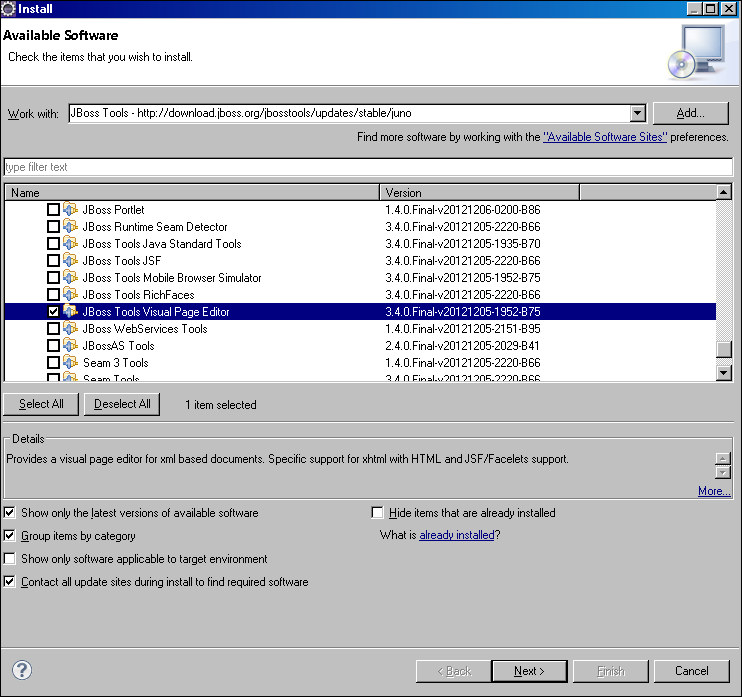
### Eclipse Plug-ins

JBoss has a decent Visual XML editor among many other Eclipse plug-ins. For now, let’s only install the Visual XML editor.

Navigate from the Menu to Help🡪Install New Software. In the Work with box enter the following:

JBoss Tools - <http://download.jboss.org/jbosstools/updates/stable/juno>

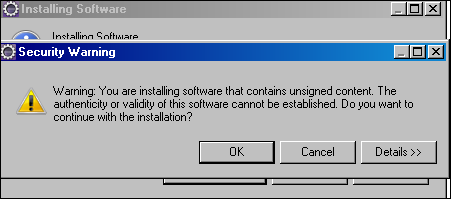
Expand the JBoss Web and Java EE Development tree. **WARNING - DO NOT INSTALL EVERYTHING.**

Scroll down to JBoss Tools Visual Page Editor and check the box. 

**WARNING** – Installing several features tends to time out. Install one thing at a time.

Click Next 🡪 Preview the installation. Click Next 🡪 again then click 🡪Finish. Accept the license agreement and click 🡪Finish again.

Click 🡪OK to finish the installation



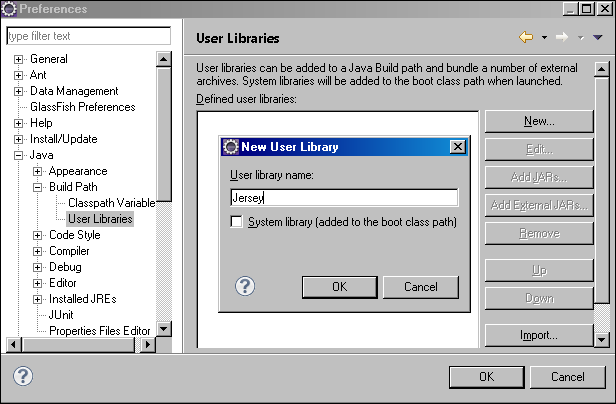
Then restart Eclipse



### Create Jersey User Library

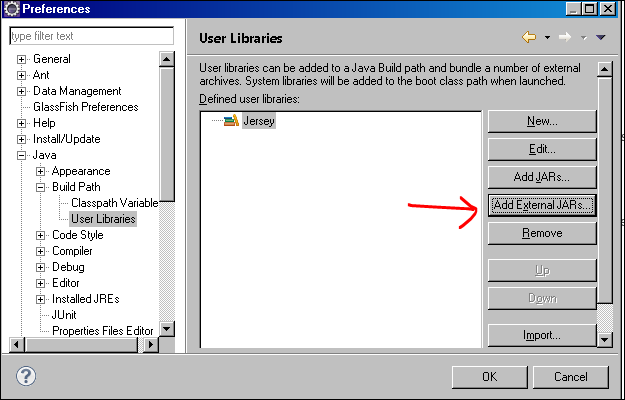
Creating a User Library for Jersey will allow you to reference the needed classes at compile time. The User Library is added to the Eclipse CLASSPATH and needs to be added to each new REST Web Service project. The library contents are already on Glassfish and do not need to be packaged in your deployment file.

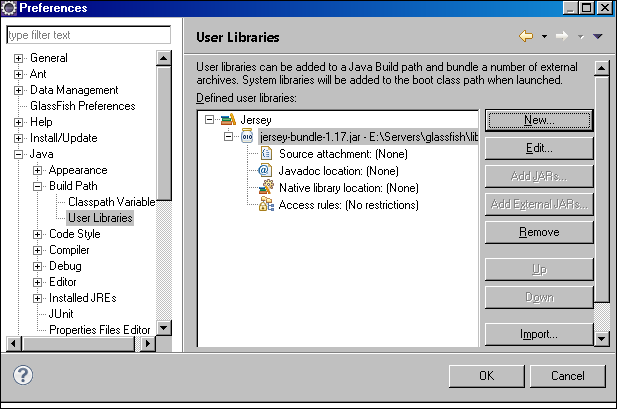
To create a User Library use the menu and navigate Window 🡪 Preferences. In the preferences pane expand the Java 🡪 Build Path tree and select ***User Libraries*** then click 🡪*New.*

Enter ‘Jersey’ and Click 🡪OK

Click 🡪Add External JARS

Navigate to the <GLASSFISH HOME>/lib directory



Locate the jersey-bundle-1.17.jar and click 🡪OK 

The Jersey User Library is now set up for inclusion in a REST Web Service Project. Click 🡪OK.

# Project Description

## Requirement

The requirement is to create a REST Web Service that creates, retrieves’, updates and deletes a US state resource.

1. Service must support XML output and JSON output formats.
2. Service must retrieve by State Code
3. Service must return a list of state resources.
4. Show both XML output and JSON output format within a Browser.

## Data and Data Storage

The developer will be provided a *Database.java* class that simulates the CRUD capabilities of a persistence layer. Data will only persist in memory at runtime. Redeployments and server restarts will not retain any data.

### Understanding the requirement

REST Web Services uses the HTTP *methods GET, PUT, POST, DELETE, HEAD, OPTIONS, and TRACE*as actions to perform on the service. This project requires the implementation of GET, PUT, POST, and the DELETE methods. The OPTIONS method is a method that is used to retrieve information about the service. The output from the OPTIONS method is provided by Jersey for free. This output is also known as the Web Application Description Language or **WADL.**

### White Board URI pattern

REST Services have what’s called a root path. The root path is the first path after the context path of the application for example if we wanted to access the **banana** resource from the **monkey-ws** service application the URL may look like this: <http://localhost:8080/monkey/api/v1/bananas/banana>

Where:

Server URL: [*http://localhost:8080*](http://localhost:8080)

Application: monkey-ws

Rest Root path: /api

Resource collection: /bananas

Resource: /banana

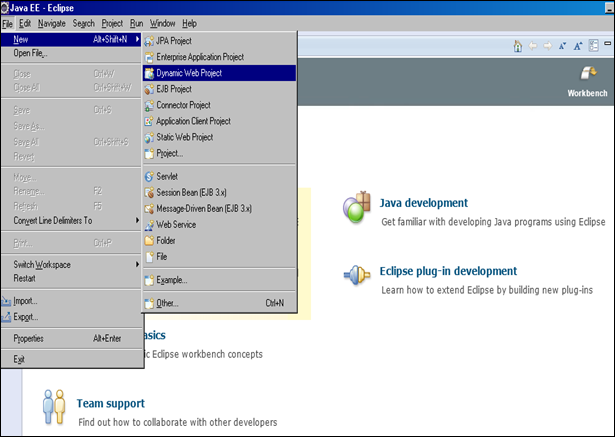
For this project we will use the following for our URI pattern

|  |  |  |
| --- | --- | --- |
| **HTTP METHOD** | **Path** | **Description** |
| GET | /api/states | Retrieve a list of States |
| GET | /api/states/{code} | Retrieve a State resource |
| POST | /api/states | Create a State resource |
| PUT | /api/states/{code} | Update a State resource |
| DELETE | /api/states/{code} | Delete a State resource |
| OPTIONS | /api | Retrieve a WADL |

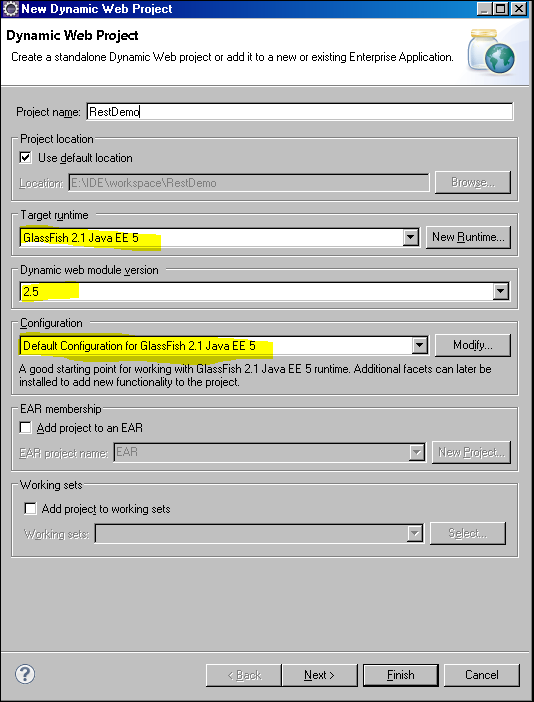
***\*\* NOTE:*** The notion of the braces **{}** as in ***{code}*** is called URI a Template. This is a programmatic convention that allows for variables to be embedded in the URL where values are injected at runtime.

## Developing the Project

Within Eclipse create a new project from the menu select File 🡪 New 🡪***Dynamic Web Project***.

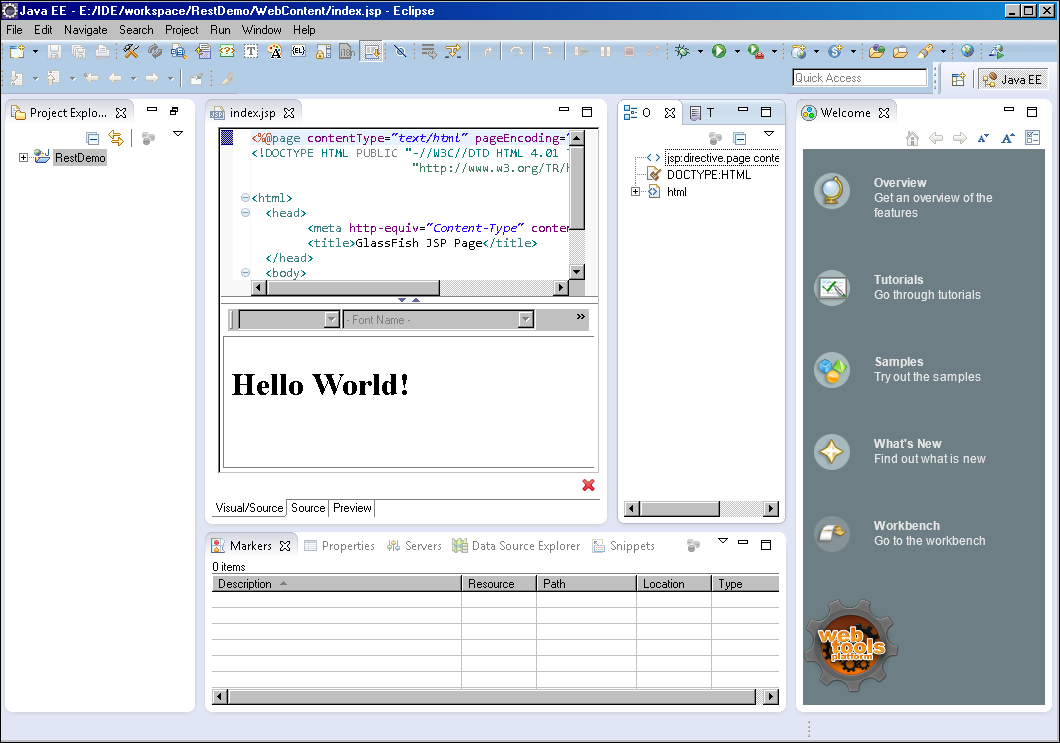


Name the project **RestDemo**

****

Ensure the target runtime, dynamic module version, and configuration is set for Glassfish V 2.1.1. If there is no target runtime displayed and no options to select from click the🡪 New Runtime button and follow the steps in the section *.*

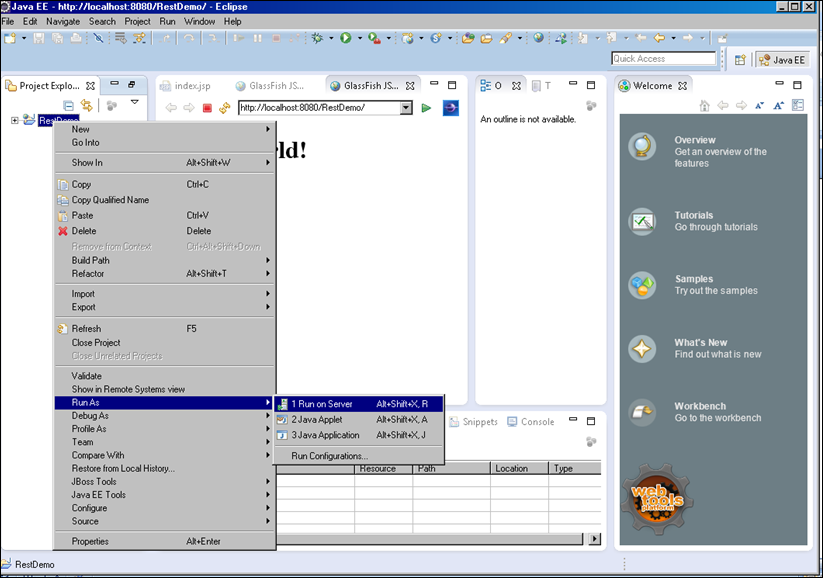
Click the 🡪Finish Button.



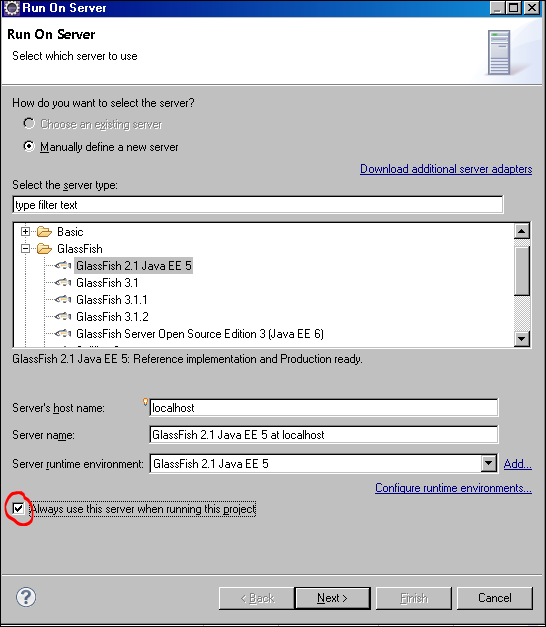
This completes the initial project creation.

The project wizard created a shell web application with an index.jsp page saying Hello World. Let’s attempt to run it on Glassfish before we make changes.

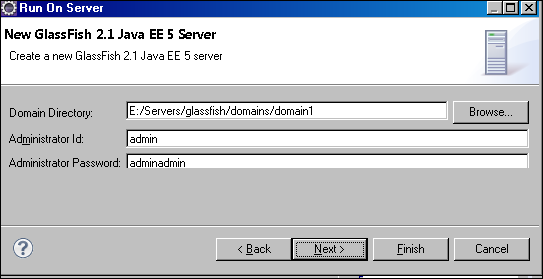
Right click on the 🡪project name in the left hand project explorer and select Run As 🡪 Run on Server.



Check the Always use this server when running this project and click Next 🡪

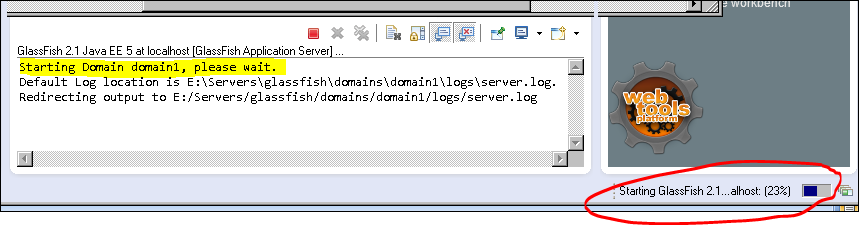


The first time the Run on Server is used for Glassfish the location and console administration configuration of credentials will appear. The default account name is ***admin*** with password of ***adminadmin*** as indicated below:

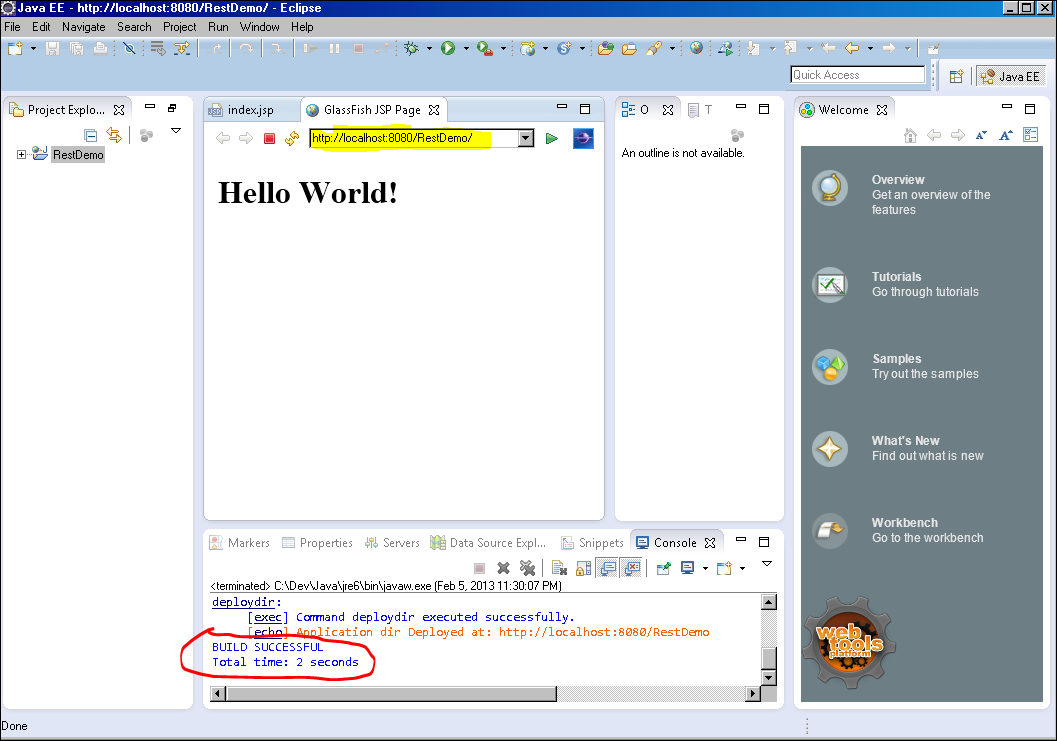


Click 🡪Finish

At the bottom of Eclipse you will see the Glassfish starting.



The application should build successfully and display the Hello World jsp page using the Eclipse internal browser as shown below.

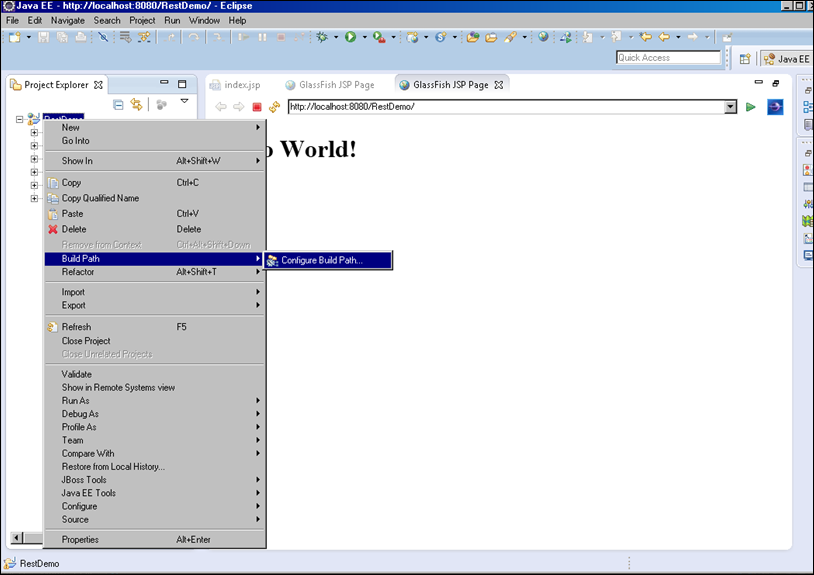


### Task 1 – Create StateResource Class

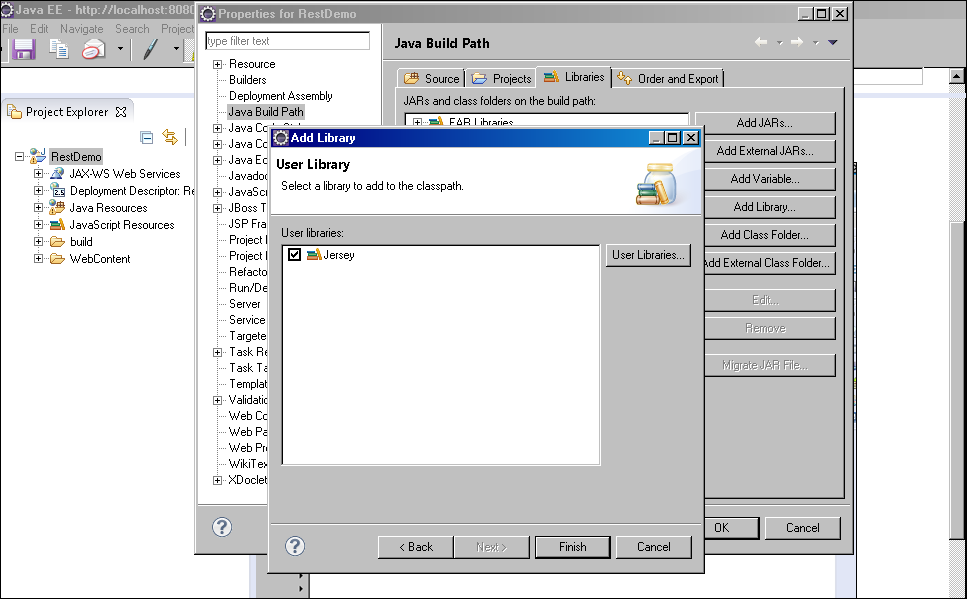
The StateResource class is where the REST Service will be implemented. Create a class named ***StateResource*** in the package name ***mil.army.train.enterprise.service.resources***. We will begin with a very simple example and publish it to Glassfish for a test run before tackling the details of the project requirement.

First step is to add the Jersey Library to our project.

Right click on the 🡪project name select ***BuildPath 🡪 Configure BuildPath***

******

Select the Library tab and click 🡪Add Library … select User Library then check Jersey click Finish



***Let’s add some code***

Expand the project tree navigate to java resources -🡪 src

Right click 🡪 src and select 🡪New package enter mil.army.train.enterprise.service.resources click finish.

Right click 🡪package select🡪 New Class enter State and click 🡪Finish.

JAX-RS uses many annotations that you need to get comfortable with using. Refer to the section for explanation of the JAX-RS annotations.

The first thing we need to do is add the **@Singleton** annotation so only one instance of the root resource is created per web application.

Next add the **@Path** annotation. We already determined what our root path for the service, “/api”, and the path to the states resource is “/states”. The root path of the service is defined in the web.xml which we will get to shortly.

The @PATH annotation can be added at the Class level and at the method level. Method level PATHS are relative to the Class level. We will add the “/states” path at the Class level as below.

@Singleton

@Path("/states")

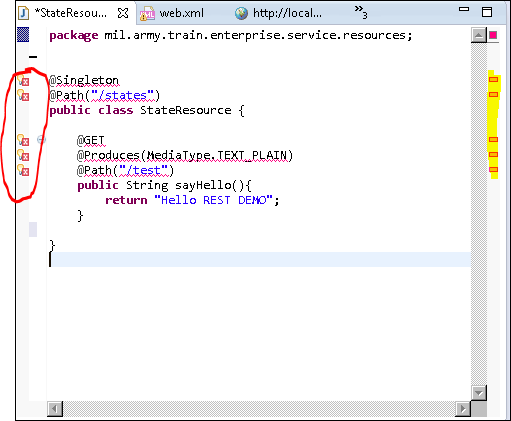
**public** **class** StateResource {

}

Let’s add a test GET method with a method level Path of /test that it returns a string.

We add a GET annotation.

Here is the class so far:

****

**Notice** the red ‘X’s. This means that Eclipse does not know where to find the Classes we are using such as Singleton, Path, Get and Produces. We need to import the appropriate package to our class.

First save your class then right click each red🡪 **x** icon and select 🡪 **Quick Fix** [HOT KEY <CTL> 1 ] and select import **Singleton** - **com.sun.jersey.spi.resource.** For **Media Type** import **javax.ws.rs.core**.

Here is the complete class.

**package** mil.army.train.enterprise.service.resources;

**import** javax.ws.rs.GET;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.Produces;

**import** javax.ws.rs.core.MediaType;

**import** com.sun.jersey.spi.resource.Singleton;

@Singleton

@Path("/states")

**public** **class** StateResource {

@GET

@Produces(MediaType.*TEXT\_PLAIN*)

@Path("/test")

**public** String sayHello(){

**return** "Hello REST DEMO";

}

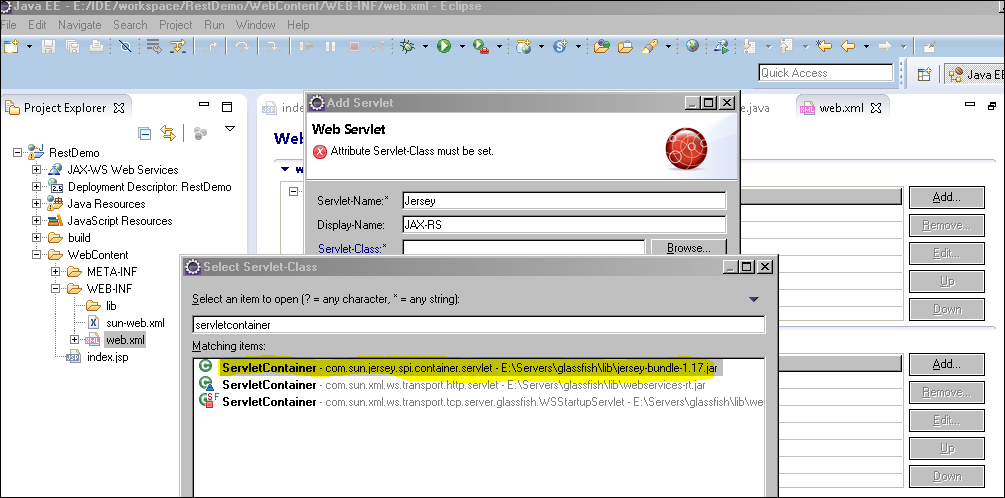
}

Let’s run it and see if it works. You can try … but you will not see anything. You will get an HTTP 404.

We have one more configuration to make. We need to register our class with Jersey and we do this in the web-xml file. We have completed **Task 1 – Create StateResource Class**.

### Task 2 – Register Service in web.xml

Expand the RestDemo project tree navigate to WebContent 🡪 WEB-INF 🡪web.xml. Double click on 🡪web.xml file. If you do not get a visual editor and would like it right click on the🡪 web.xml file navigate to ***Open With*** and select JBoss Tools XML Editor. In The Web XML Editor select **Servlets** then click the Add button. Name the Servlet Jersey, Display name JAX-RS and for the servlet class click the Browse button and search for **ServletContainer**. Select the ***com.sun.jersey.spi.container.servlet.ServletContainer*** class.



Click 🡪Finish.

Set Load-on –Startup to 1

Click the 🡪Add Button under ***InitParams*** and enter the following values:

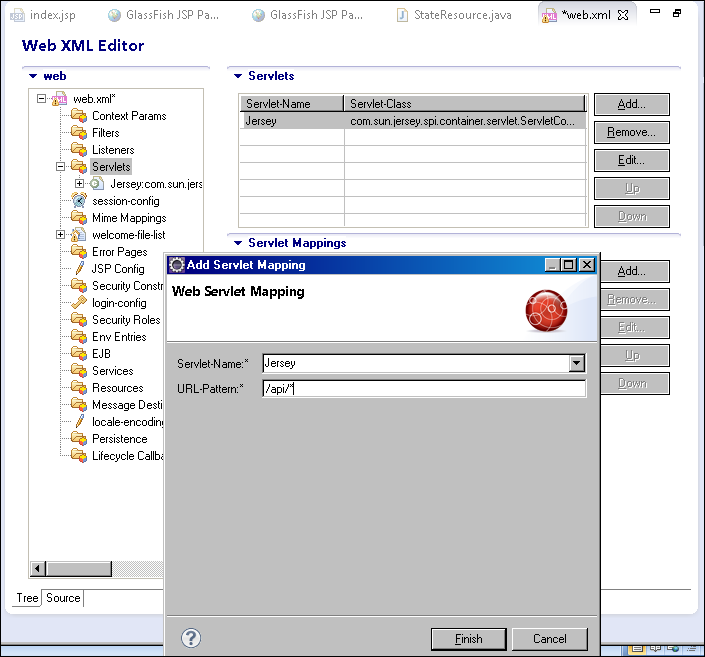
Param-Name: **com.sun.jersey.config.property.packages**

Param-Value: **mil.army.train.enterprise.service.resources**

Note: the Param-Value is the name of your packages where your REST Service implementation resides.

Click back on the 🡪**Servlets** Tree in the Web XML Editor under Servlet Mappings click 🡪Add

Select 🡫 Jersey in the dropdown and add the root path to the service /api/\*

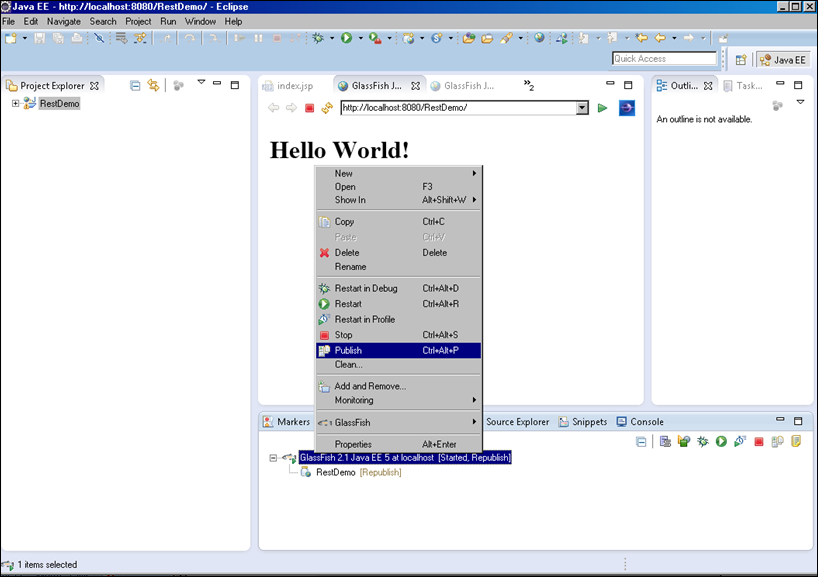
Click 🡪Finish save the project.

We are ready to run it on Glassfish. We have completed .

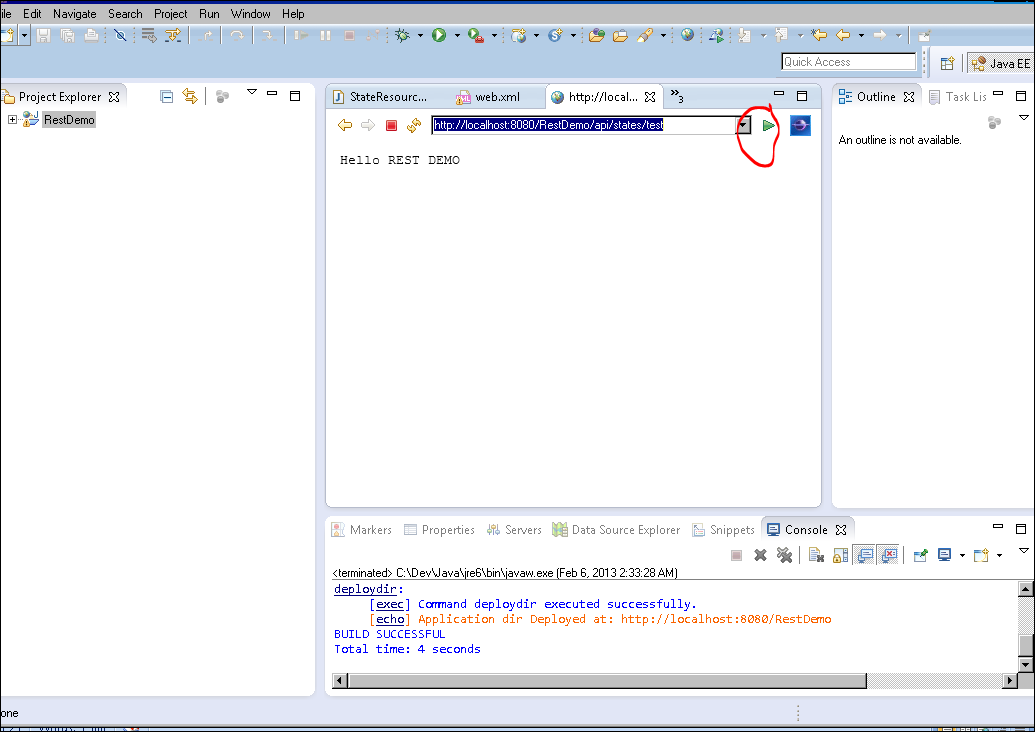
### Task 3 – Test Run on Glassfish

Navigate to the Server tab at the bottom of Eclipse. Don’t see it you can double click any of the source files or press **<CTL> m**

Right click 🡪 Glassfish Server and select 🡪 Publish



Change the URL to <http://localhost:8080/RestDemo/api/states/test> and click the 🡪Green arrow.



Yeah!! You have successful created and ran a simple REST Web Service on Glassfish. Let’s add on to it. We have completed

### Task 4 – Create Database POJO Using JAXB Annotations

Create Database POJO for the source of the State Resource Class and add the necessary JAXB annotations for XML serialization/de-serialization of the object. Create the class named ***State*** in the package **name mil.army.train.enterprise.service.model**

Expand the project tree navigate to java resources 🡪 src

Right click 🡪 src and select🡪 New package enter mil.army.train.enterprise.service.model click finish.

Right click 🡪 package select 🡪New Class enter State and click finish.

In the State Class source code

Add implements Serializable

Add 4 instance variables:

**private** **int** id;

**private** String code;

**private** String name;

**private** Date modified;

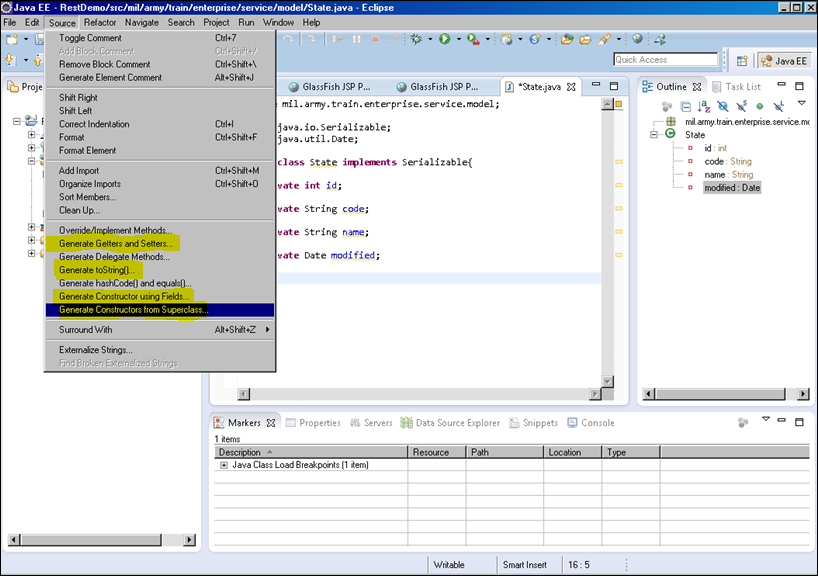
Add No Argument public Constructor [Menu Source 🡪 Generate Constructor …]

Add public Constructor with all arguments. [OR hot key <ALT> s]

Add getters and setters for all variables.

Add a toString () method consisting of all instance variables.

Eclipse has code generator for Constructors and getters/setters etc.



Add JAXB annotations

Above the class add the JAXB annotations

@XmlRootElement(name="state")

@XmlAccessorType(XmlAccessType.*FIELD*)

@XmlType(propOrder={"id","code","name","modified"})

**public** **class** State **implements** Serializable{

. . . . .

}

Above each instance variable add the @XmlElement(name=”variable name”} as in the following:

@XmlElement(name="id")

**private** **int** id;

@XmlElement(name="code")

**private** String code;

@XmlElement(name="name")

**private** String name;

@XmlElement(name="modified")

**private** Date modified;

. . .

. . .

. . .

\*\*\* Import all unknown XML classes from the **javax.xml.bind.annotation** package.

Save the State class.

That completes .

### Task 5 – Implement the GET method using Resource POJO

Let’s implement a GET method that uses the State resource class we just created. In the ***StateResource*** class we want to now Produce XML output instead of String. We also need to add a Path sub-resource to the **state** resource since our root path is at the collection level. Our return type will now be a representation of our State class.

Add a GET method int the **StateResource** class as below

@GET

@Produces(MediaType.*APPLICATION\_XML*)

@Path("/state")

**public** State getState() {

State state = **new** State(1,"VA","virginia",**new** Date());

**return** state;

}

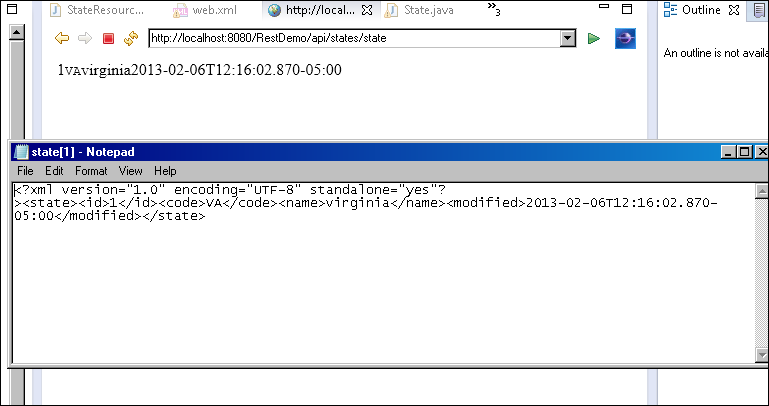
Import the State class from the **mil.army.train.enterprise.service.model** package

Import the Date class from the **java.util** package.

Let’s publish the project to Glassfish. Navigate to the Server Tab Select Glassfish Press **<CTL >< ALT> p**

Select the tab <http://localhost> … within Eclipse and change the URL to <http://localhost:8080/RestDemo/api/states/state> and click 🡪 the Green arrow. To see the XML tags right click on the window and view source.

The output should look something like:

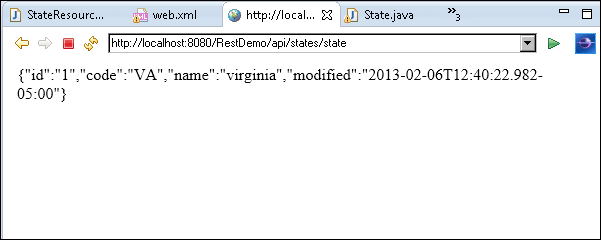


Let’s change our GET method Produces to **MediaType.APPLICATION\_JSON**

@Produces(MediaType.*APPLICATION\_JSON*)

Let’s publish the project to Glassfish again and see the new output.

This time it rendered in our internal browser as raw JSON text.



**Note:** To list multiple values in the Produces Annotation the entire contents be surrounded by braces and each value separated by a comma.

@Produces({MediaType.*APPLICATION\_JSON*, MediaType.*APPLICATION\_XML*})

Jersey will render the output to the client in the order listed in the Produces annotation. Only if the client has specified a particular Accept header will the service honor that Media Type. If we publish again this time we will see the XML representation. The internal browser is controlling the Accept headers. We will show this later using a different client test tool. For now this completes .

### Task 6 – Generate List of Resources

To generate a list of the Path value is not set this means it will take the value from the Class level which is “/states” this is the URI path we want to expose for getting a list of States.

Here is the implemented code for generating a list

@GET

@Produces({MediaType.*APPLICATION\_XML*, MediaType.*APPLICATION\_JSON*})

**public** List<State> getStates(){

List<State> states = **new** ArrayList<State>();

State state1 = **new** State(1,"VA","virginia",**new** Date());

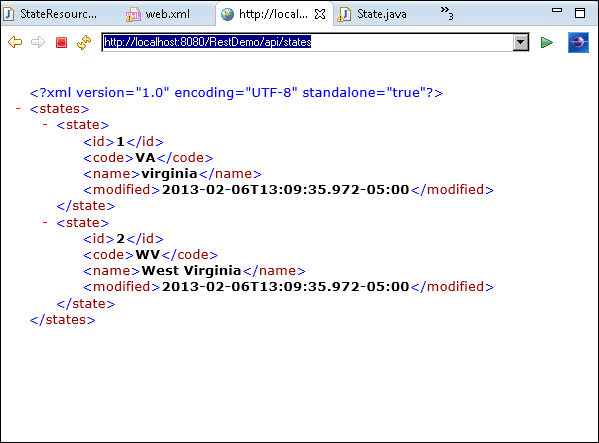
State state2 = **new** State(2,"WV","West Virginia",**new** Date());

states.add(state1);

states.add(state2);

**return** states;

}

The URL for the list of states is [http://localhost:8080/RestDemo/api/states](http://localhost:8080/RestDemo/api/states)

This completes .

### Task 7 – GET a Specific Resource

Recall from section where we noted URI Templates for handling variables at runtime. Here we will implement a GET method that will receive a Path Parameter from the client and just display it back for now. We will implement more details later.

In the StateResource class create another method GET getState() but this time to receive a parameter and then just display what parameter was passed in. We use a new annotation PathParam. We also have to add the URI template variable so Jersey can inject the value at runtime.

@GET

@Produces(MediaType.*APPLICATION\_XML*)

@Path("/{code}")

**public** State getState(@PathParam("code") String code) {

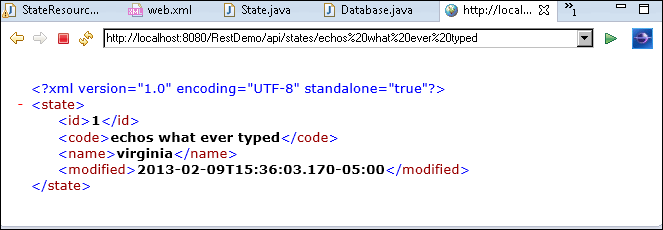
State state = **new** State(1,code,"virginia",**new** Date());

**return** state;

}

Let’s publish to Glassfish and see what we get. The URL to test is

[http://localhost:8080/RestDemo/api/states/](http://localhost:8080/api/states/state/)🡨Here we type any value to pass in to the service



That completes .

### Task 8 – Mock Database

Before we go further implementing the PUT, POST, and DELETE methods let’s set up a Mock Database. This mock database is nothing more than a HashMap that stores objects in a list for at runtime. There are methods similar to Hibernate/JPA. This Database Class has already been implemented for you.

Add a new class in the **mil.army.train.enterprise.service.model** package. Name it Database then copy the contents of the source code included in the section 🡪. It is also available T:\DeveloperTools\RestDemoArtifacts\Database.java. Place the contents in the new class just created.

The database class has 6 methods you can call.

|  |  |
| --- | --- |
| **Method** | **Description** |
| findAll(Class) | Query and retrieve all records in Class |
| findBy(Class,column) | Query and retrieve record in Class by column name |
| findById(Class, pk id) | Query and retrieve record in Class by primary key |
| remove(Class, pk id) | Delete record in Class by primary key |
| save(entity) | Save object in store errors on duplicates |
| saveOrUpdate(entity) | Save or update object in store. Creates or updates if already exists |

Back in the State Resource Class at the instance level we need bring in an instance of the Database class.

Here’s how:

@Singleton

@Path("/states")

public class StateResource {

private Database database = Database.getInstance();

@GET

@Produces(MediaType.*TEXT\_PLAIN*)

@Path("/test")

**public** String sayHello(){

**return** "Hello REST DEMO";

}

. . .

}

Be sure to **import** mil.army.train.enterprise.service.model.Database

This completes .

### Task 9 – Modify Get Methods to Query a Database

Modify the **public** List<State> getStates() method to retrieve all states stored in the database using the findAll() method.

Here’s how:

@GET

@Produces({MediaType.*APPLICATION\_XML*, MediaType.*APPLICATION\_JSON*})

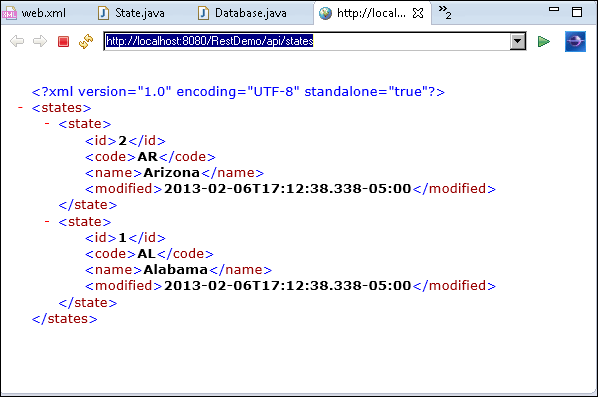
**public** List<State> getStates(){

List<State> states = database.findAll(State.**class**);

**return** states;

}

Let’s publish this in Glassfish and see what we get.



There are two records already in the database. Now let’s change the GET method that is using the PathParam variable to retrieve the value from the Mock Database using the Database findBy() method instead of just echo what is passed into the service. The findBy() method takes the class name representing the database table, the column name to use and the value for the column to search for.

Here is how:

@GET

@Produces(MediaType.*APPLICATION\_XML*)

@Path("/{code}")

**public** State getState(@PathParam("code") String code) {

State state = (State) database.findBy(State.**class**, "code", code);

**return** state;

}

Publish to Glassfish and test it out. Remember there are only two records currently in the database. Try other values. And see what you get.

Hmm testing values that are not in the database seem to either do nothing or render the same page. What is Jersey sending back? I expected to get a 404 page not found. How can I return page not found? We’ll look at both those questions in the next task.

This concludes .

### A Quick Introduction to REST Client Test Tools

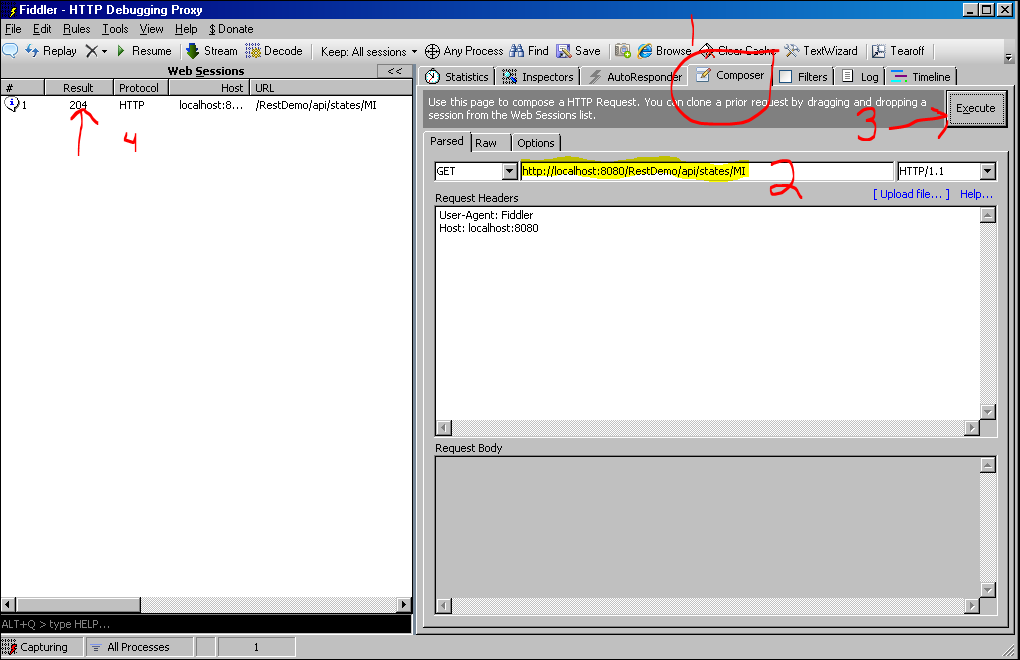
Using the internal IDE browser has its limitations, in fact, Internet Explorer and Fire Fox have limitations also. These browsers can only support issuing GET requests on the URL line. So before we get into PUT, POST, and DELETE examples, let’s get familiar with some readily available REST Client Test Tools. These tools will allow us to issue PUT, POST, and DELETE method calls as well as see what happened in the previous task.

Open Internet Explorer. From the menu navigate to 🡪Tools and select 🡪Fiddler. A popup box will open asking “A Web Site wants to open web content using this program on your computer. “ Click 🡪 Allow.

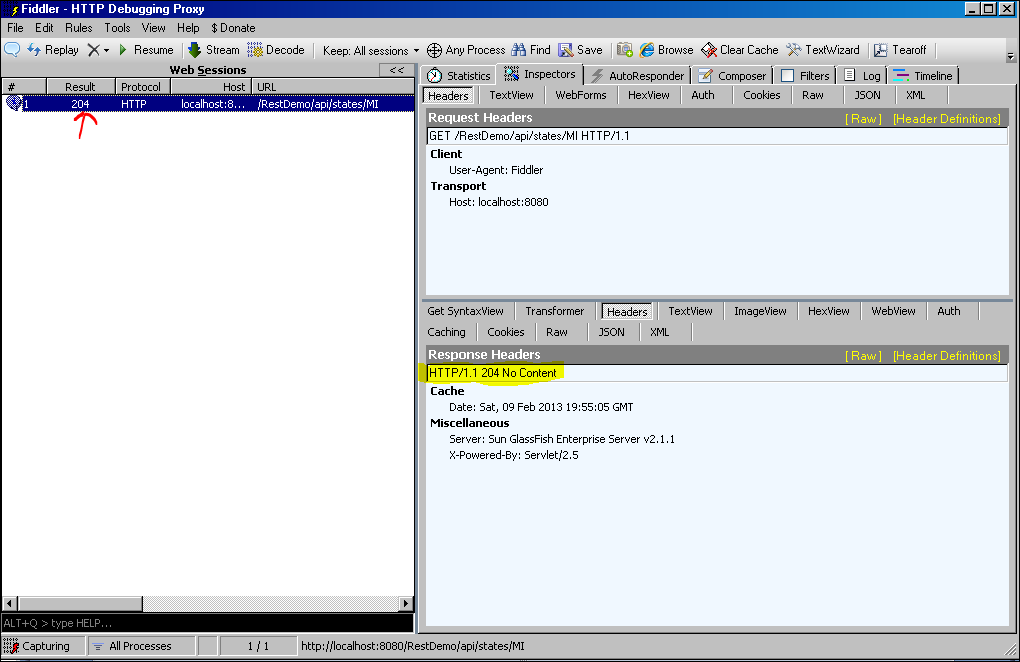
Click 🡪***Composer*** tab

Enter the URL <http://localhost:8080/RestDemo/api/states/MI>

Click 🡪 Execute



The response will appear in the left window. Double Click the Result line.

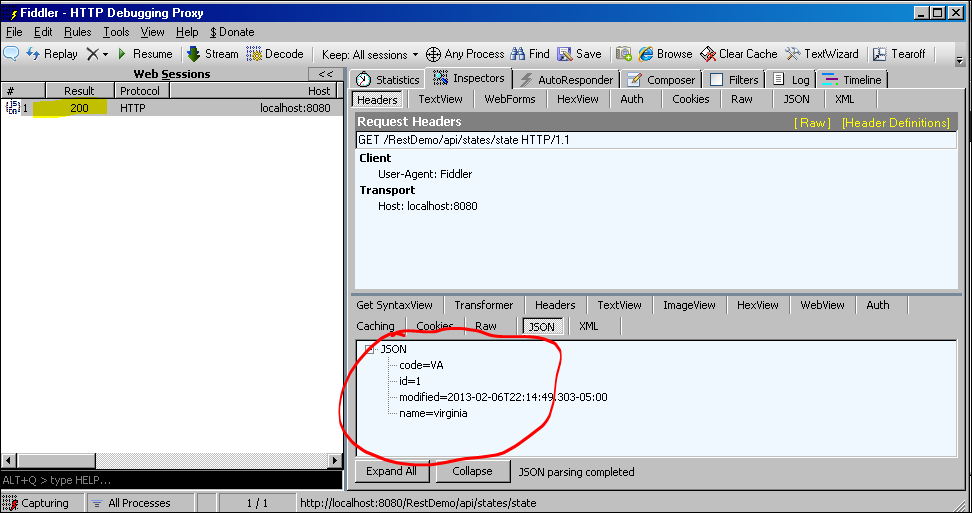


The result response information will be populated in the right hand window. The response from the URL call to <http://localhost:8080/RestDemo/api/states/MI> we knew should be not found in the database came back as an HTTP 204 – No Content. Ok so that answers the first question “What did Jersey send in the response”. There are a couple of ways to change what Jersey is sending back. We could throw an exception or use a “Response” object to better control what Jersey returns. They will be discussed the next task.

As you can see there are several tabs. We are most interested in the **Compose** tab for requests and the Headers, JSON, XML, TextView and WebView tabs for the responses.

Currently in our service there is one method that has a Produces MediaType.APPLICATION\_JSON if we ran that request in Internet Explorer it wants to save a file to disk. Here in Fiddler we can see what the request is returning without saving it to disk.

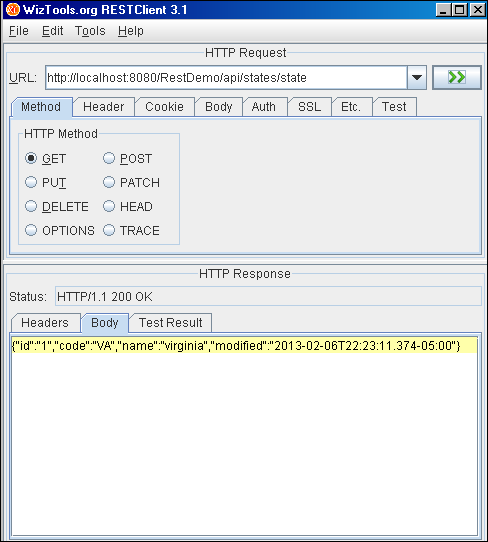
In Fiddler click 🡪 **Compose** tab and enter the URL <http://localhost:8080/RestDemo/api/states/state>



Here we see that the Response code is 200 and the response data is in JSON format.

Let’s look at the Google RESTClient tool. Copy the files from the T:\DeveloperTools\RestClientTools\CLI-Google to your local machine. The only requirement is JAVA\_HOME must be set. Double Click on the restClient.bat file.

Enter the URL <http://localhost:8080/RestDemo/api/states/state> click the green >>



These REST Client Test Tools will come in handy testing the PUT, POST, and DELETE methods shortly. SoapUI is another tool you could use but we will not cover it here.

That completes .

### Task 10 – Implement the PUT Method

In the StateResource class let’s implement the PUT method. The HTTP PUT method can mean both Create and Update depending on who should be in control of the ‘KEY’ data. In our case for this demonstration our PUT will mean Update only. We already know how to use the @***PathParam*** receive an input value from the client. This will be the record to look up in the database to change. So, how do we tell the client how to change the value of the other fields not on the URI path?

The service method will use a @***Consumes*** annotation to tell the client what is acceptable format for passing data to the method. This service will use the common “application/x-www-form-urlencoded” media type that HTML forms uses to POST data. There is nothing to produce so we will return void.

There is another annotation we will use for each resource field that can be modified. In this service only the ‘name’ field can be modified. This annotation is the @***FormParam*** and is placed in the method signature.

Let’s code the service

Here’s how:

@PUT

@Consumes(MediaType.*APPLICATION\_FORM\_URLENCODED*)

@Path("/{code}")

**public** **void** updateState(@PathParam("code") String code, @FormParam("name") String name) {

State state = (State) database.findBy(State.**class**, "code", code);

**if** (state != **null**){

state.setName(name);

database.saveOrUpdate(state);

}

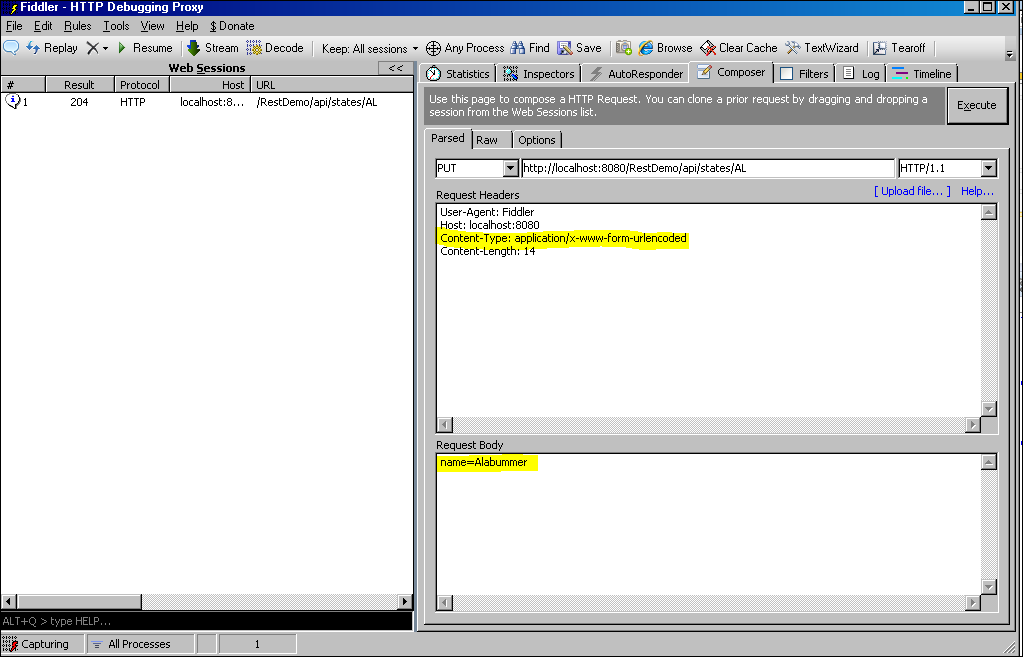
}

Let’s publish to Glassfish and test in Fiddler.

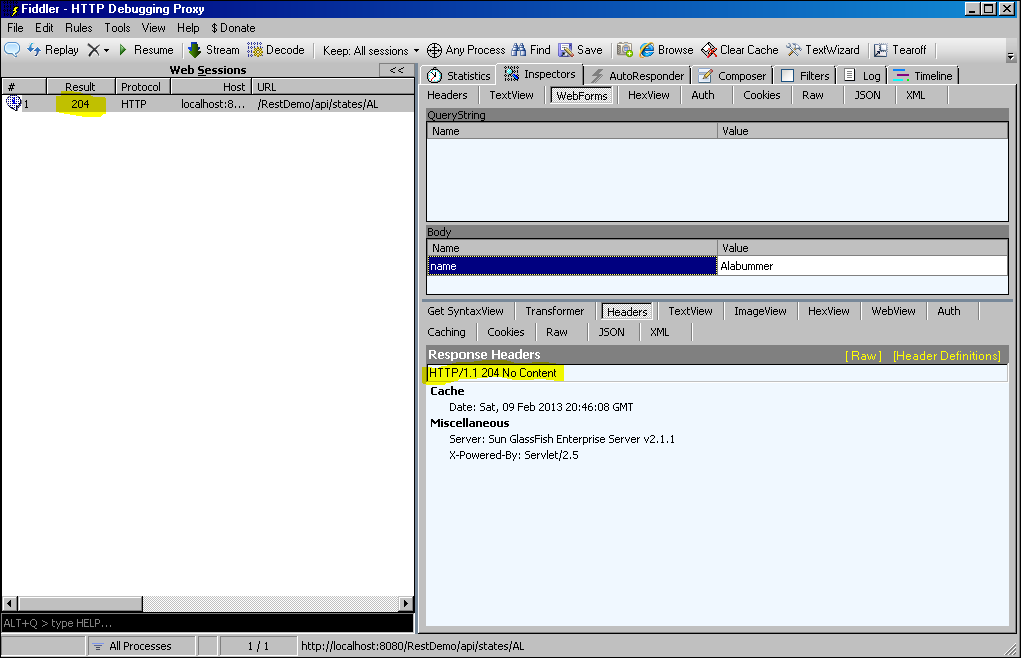
In fiddler we set the request header ***Content-Type: application/x-www-form-urlencoded*** and in the request body the form variable name ‘name’ = value we want to change.

The resource we want to change is at <http://localhost:8080/RestDemo/api/states/AL>

We are changing the name = Alabummer

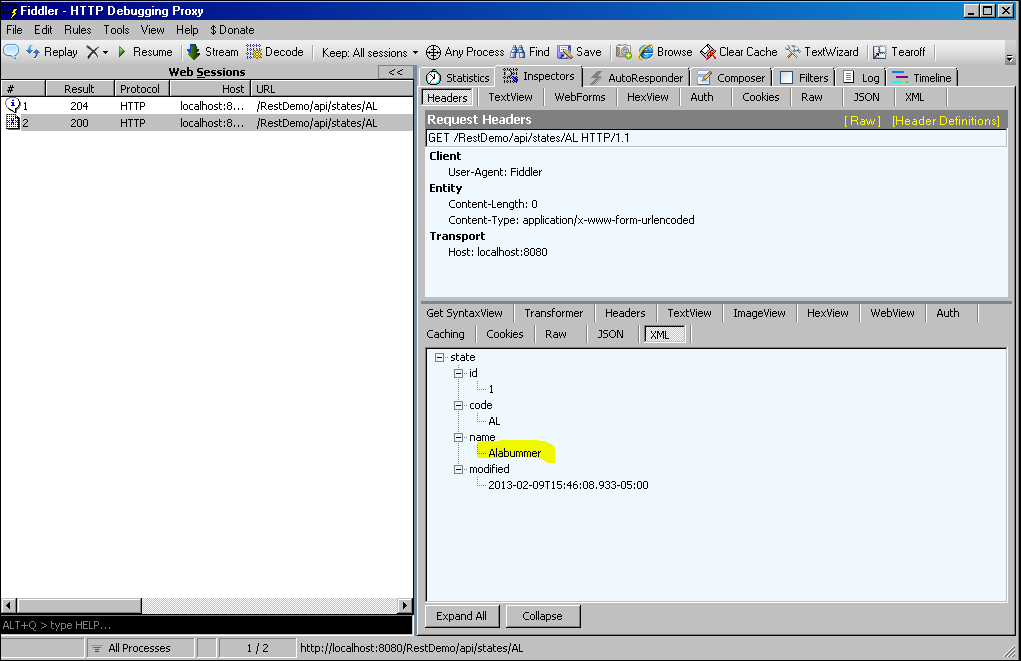


The response was successful with a HTTP 204 – No Content Status



Executing a GET request on the same URL <http://localhost:8080/RestDemo/api/states/state/AL>

We can see the change has been made.



When issuing an HTTP PUT request on a resource that does not exist Jersey returns an HTTP 204 – No Content response and also when the record was successfully updated. This could be confusing to a client.

This is the same problem as when no records found in database. Getting HTTP 204 – No Content for when data is successfully changed and when there are no records.

One way to solve this is to thow an exception and return a HTTP 404 – Not Found. The ideal way is to use the Response object which allows control of headers, media types, content, and response codes. We will not cover the use of the Response object in this session.

Here’s how throw an exception to return HTTP 404 – Not Found:

@PUT

@Consumes(MediaType.*APPLICATION\_FORM\_URLENCODED*)

@Path("/{code}")

**public** **void** updateState(@PathParam("code") String code, @FormParam("name") String name) {

State state = (State) database.findBy(State.**class**, "code", code);

**if** (state != **null**){

state.setName(name);

database.saveOrUpdate(state);

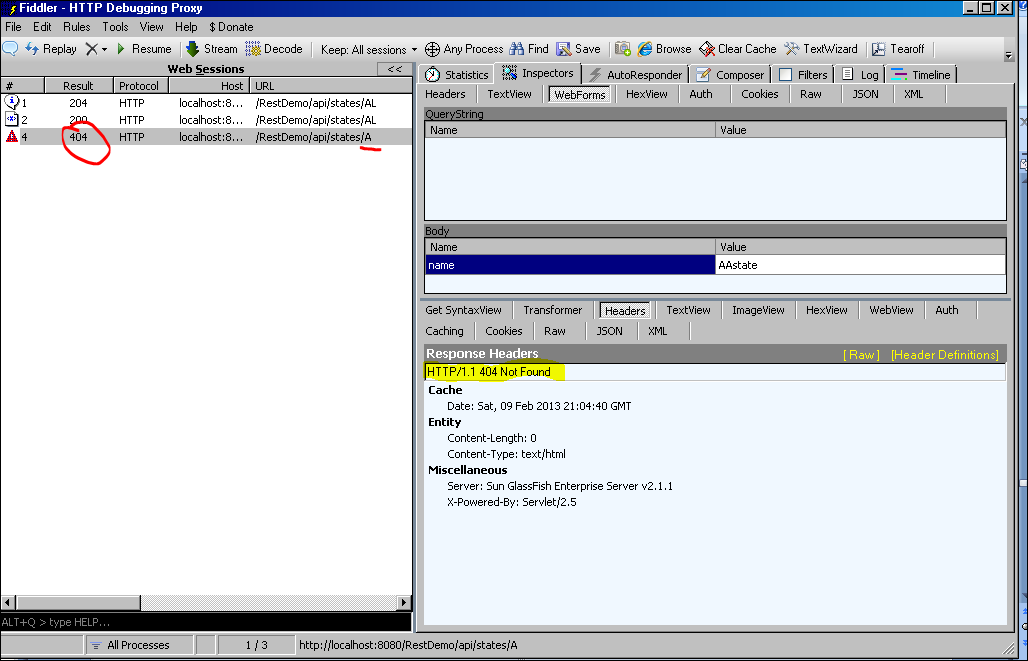
}**else**{

**throw** **new** WebApplicationException(Status.*NOT\_FOUND*);

}

}

Republish to Glassfish. Verify you are now getting HTTP 404 – Not Found when sending a PUT request to a resource that does not exist.



That completes .

### Task 11 – Implement the POST Method

The POST method is used to create a resource. Implementations of the POST method often react like a ‘Create or Update’ depending on whether the Service or Client is allowed to generate the unique identifying ‘KEY’ for the resource or not. For the purpose of this session, the POST method will only allow for the creation of a resource which the service creates the unique key. The service should return an HTTP 409 – Conflict if the resource attempting to create already exists.

If all the resource data cannot be supplied on the URI then the client must send the data in such a format the service can consume. The service will consume the media type ‘appliacation/x-www-form-urlencoded’ and produces nothing but a response. Only the FormParam annotation will be used to retrieve the client supplied data necessary to attempt to create the resource. Recall the white board session we stated the URI path for creating resources will be <http://localhost:8080/RestDemo/api/states> so the implementation of the method will not have a Path annotation. It will use the class level path.

Here’s how:

@POST

@Consumes(MediaType.*APPLICATION\_FORM\_URLENCODED*)

**public** **void** createState(@FormParam("code") String code, @FormParam("name") String name) {

State state = **new** State();

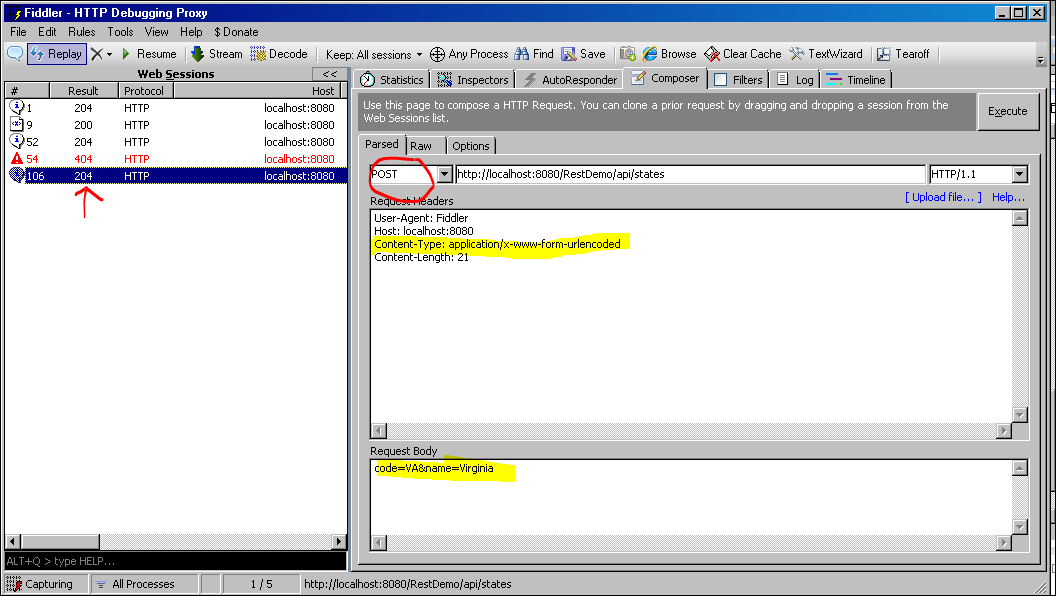
state.setCode(code);

state.setName(name);

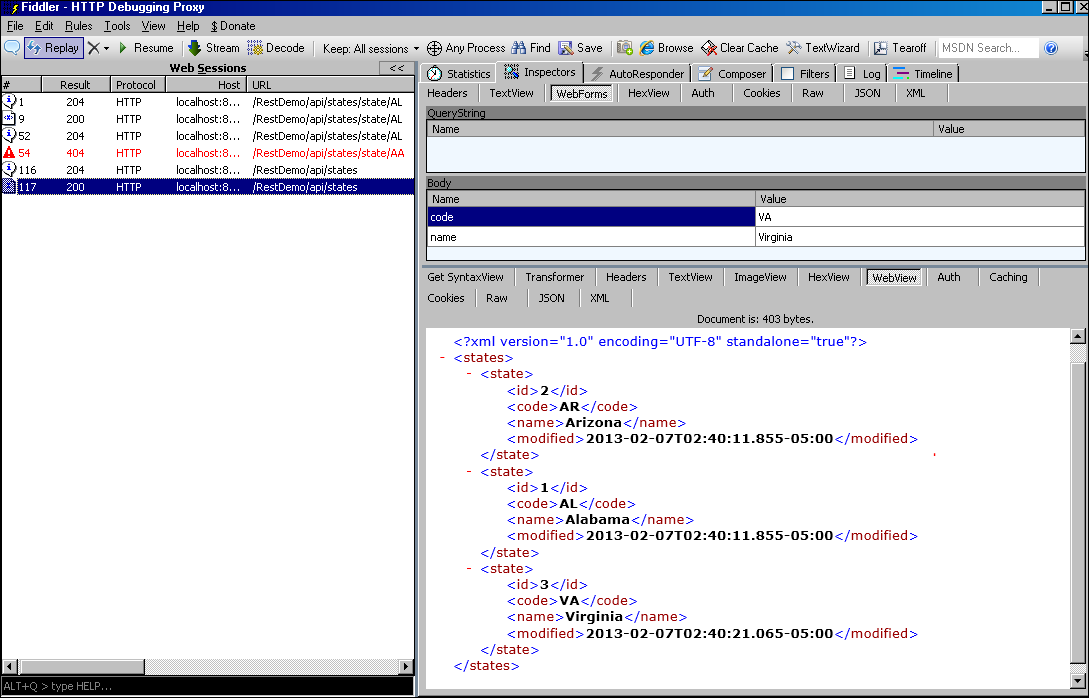
database.save(state);

}

Publish to Glassfish and use Fiddler to POST the request.



We see the same response creating the same resource as with creating a duplicate however executing a GET /states list request reveals one being created.



We can modify the service code to throw an exception returning HTTP 409 – Conflict when the record attempting to create already exists.

Here’s how:

@POST

@Consumes(MediaType.*APPLICATION\_FORM\_URLENCODED*)

**public** **void** createState(@FormParam("code") String code, @FormParam("name") String name) {

State state = **new** State();

state.setCode(code);

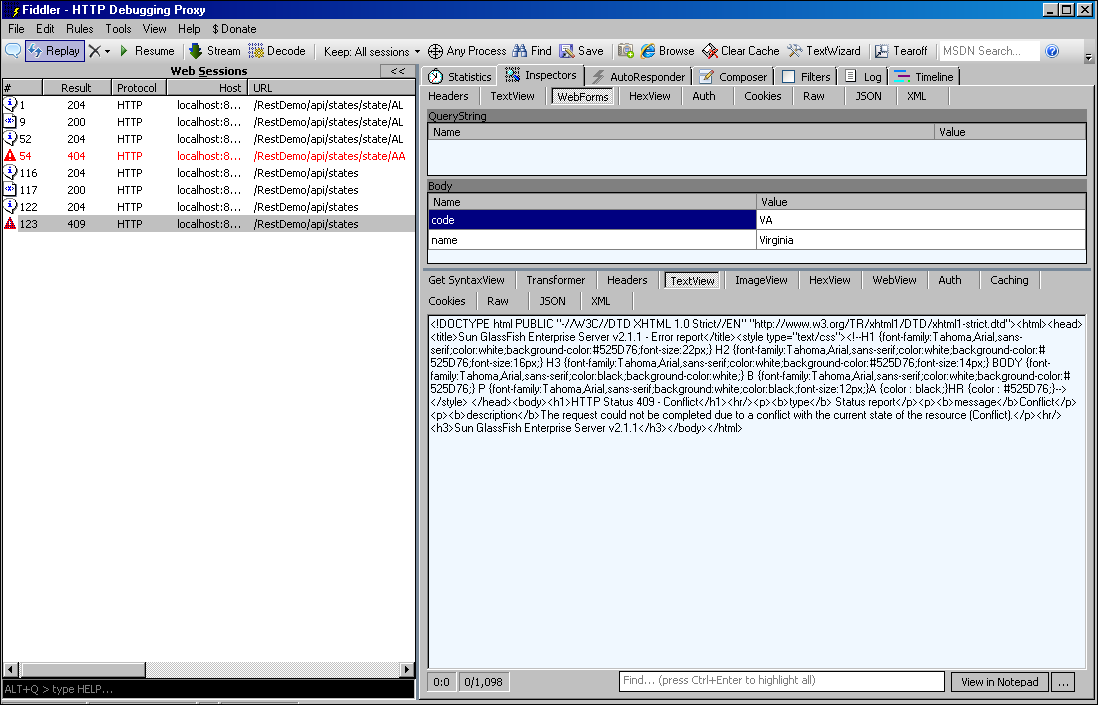
state.setName(name);

**if** (database.save(state) < 0){

**throw** **new** WebApplicationException(Status.*CONFLICT*);

}

Publish and test in Fiddler.



That completes .

### Task 12 - Implement the DELETE Method

The DELETE method is used to delete a resource. It is similar to the GET /states/{state} method using the PathParam however the HTTP method is DELETE versus GET. Recall to the project requirement that to delete we must supply the ‘id’ not the ‘code’. For this implementation know we want to return an HTTP 404 – Not Found if there is no data. Let’s throw an exception if the call to the database returns null.

Here’s how

@DELETE

@Path("/{id}")

**public** **void** deleteState(@PathParam("id") **int** id){

State state = database.findById(State.**class**, id);

**if** (state != **null**){

database.remove(State.**class**, id);

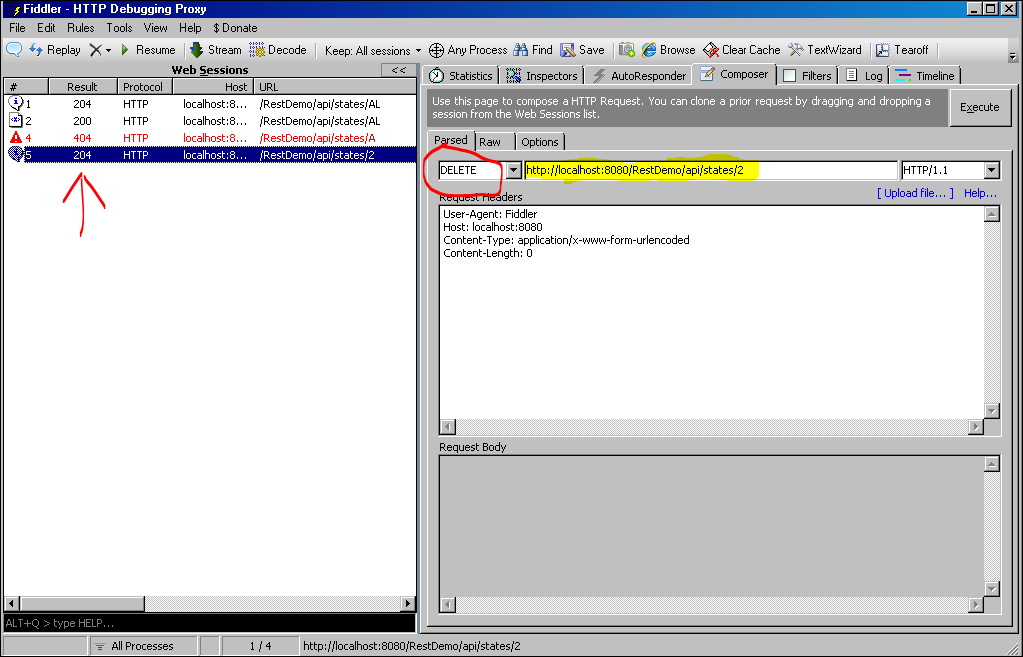
}**else**{

**throw** **new** WebApplicationException(Status.*NOT\_FOUND*);

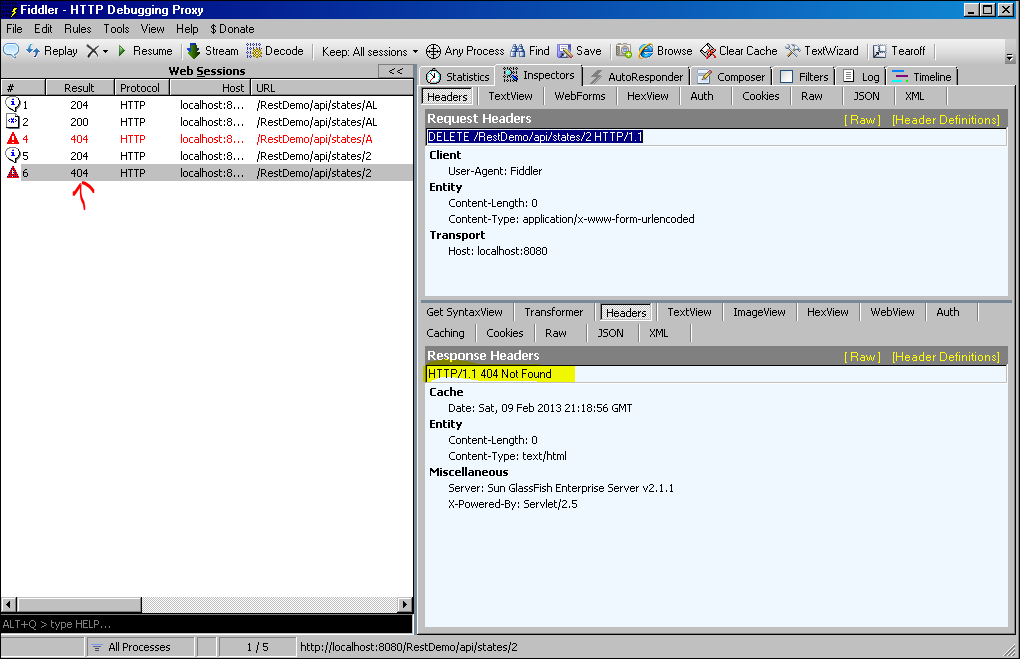
}

}

Republish and test with Fiddler reveals a HTTP Status – 204 No Content



Issuing the DELETE request again returns HTTP 404 – Not Found as expected.

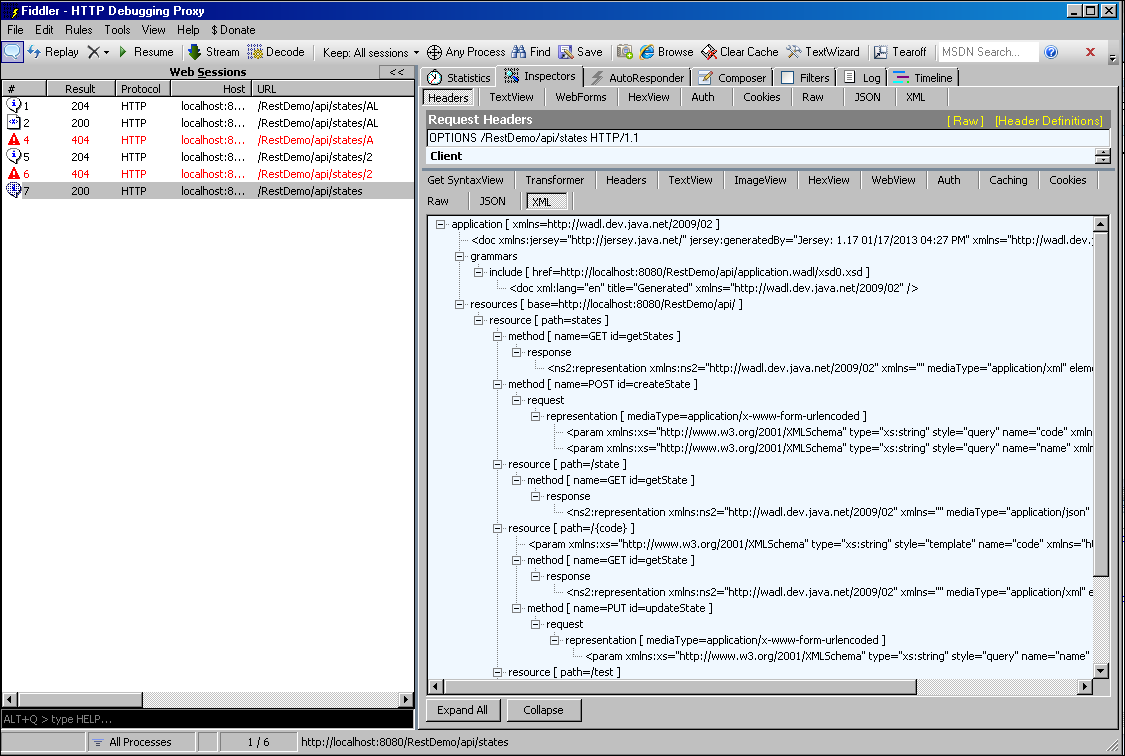


That completes **.**

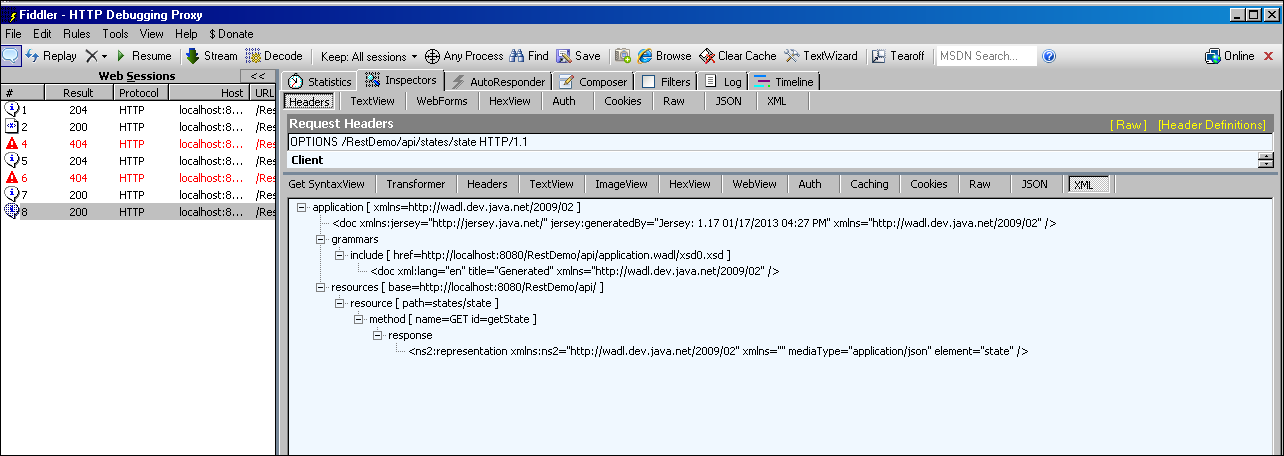
### Task 13 – The OPTIONS HTTP Method

The OPTIONS method provides information about the request and response options a client is allowed to do against a resource, what request media types are acceptable, what possible representations are available, URI template constructs, and any query parameters offered. Jersey provides this information for free. It is called a WADL document – Web Application Description Language. It can be accessed on the URI starting at the root path level and can be accessed at sub-resource levels as well reflecting only the sub-resource information. For example using Fiddler, change the HTTP method to OPTIONS and use the base URI path to the service <http://localhost:8080/RestDemo/api/states>

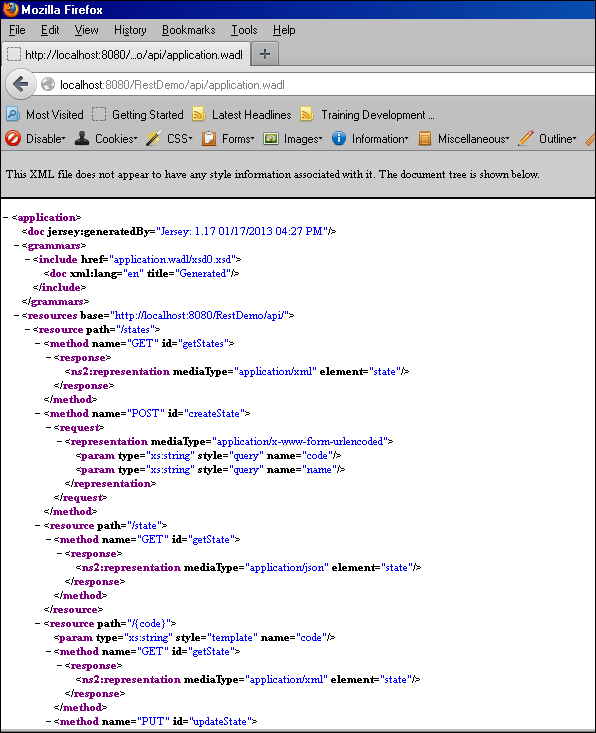
This is the result:



Request the URI path <http://localhost:8080/RestDemo/api/states/state> and you see the resource information is filtered for that resource:



Access to the WADL is available to browsers that support the “**application/vnd.sun.wadl+xml**”. Unfortunately Microsoft Internet Explorer out of the box does not. However Firefox does. Let’s see what Firefox does. With the browser using a GET method the URL access to the WADL using the GET method (or browser) tack on “/***application.wadl***” to the root path of your service as in path http://localhost:8080/RestDemo/api/application.wadl



When using Microsoft Internet Explorer you force to save it to a file first. You can still get it but is less convienient.

That completes **.**

### Task 14 – Links to Detail Resources

When returning collections of resources and rendering all sub-resource details within that list may be fine if the details of those sub-resources are few in detail and the number of resources are also few. We want to limit the metadata in the list of resources returned to the client but provide a way for the client to get those details. The way we accomplish this is to provide a link in the metadata of the collection to the detail resource.

Let’s see how we could do that with this project. In the getStates() method we are retuning List<State> object. Using the List<State> as a return object is no longer suitable for this concept. First off we are returning all the metadata that describes the State resource. And second, even though it has only a few elements, just imagine if our resource has 15 elements and there are 3000 of them in the collection. That list would be huge. So let’s cut the number of elements returned in the collection to say just code and title and then provide a link to the detailed item. You may also think about pagination of some sort. For now let’s just modify the project to do generate links to details.

We begin by creating a new resource class named States in the **mil.army.train.enterprise.service.model** package.

Let’s add an instance variable that contains a list of the State objects as in

protected List<State> states;

Add public constructor

Add getter/setter

Next is to add JAXB Annotations. The root will be named “states” and we want to reference the already existing “State” JAXB element instead of creating a new element.

Let’s see how we mark up the States class for XML representation of a list of State objects.

@XmlRootElement(name="states")

@XmlAccessorType(XmlAccessType.*FIELD*)

**public** **class** States {

@XmlElementRef

**protected** List<State> states;

**public** States() {

**super**();

}

. . .

We have not added any links yet however lets modify our getStates() method to use the States object instead of returning List<State>. Note: The expected result returning States object as with returning the List<State> object should be the same for the moment.

This is how

… Sorry ran out of time! Will try to go over in class.

# Principles of REST

RESTful web services are implemented using HTTP and the principles of REST. A RESTful web service a collection of resources defined by:

1. Addressability - The base URI
2. Uniform Interface - A set of supported operations defined by HTTP Methods
3. Statelessness – Requests are self contained and independent
4. Representations - The internet media type consumed and produced by the web service
5. HATEOAS – Hypertext as the Engine of Application State -The API must be hypertext driven.

# HTTP Methods

|  |  |
| --- | --- |
| **HTTP Method** | **Description** |
| GET | The HTTP GET method is used to retrieve a representation of a resource. |
| PUT | The HTTP PUT method is used to create a new resource or modify an existing resource.  **NOTE:** To determine the appropriate usage between PUT and POST, the client uses PUT when it is in charge of deciding what the new URI resource should be when it is created and the client uses POST when the server is in charge of assigning a new URI to the resource being created. |
| POST | The POST method is used create a new resource. Typically the client only needs to know the parent URI resource in which the server creates the resource under the parent. The Server response is usually HTTP Status 201 (“Created”) with the location header containing the URI of the newly created resource. |
| DELETE | HTTP DELETE is used to delete an existing resource. |
| HEAD | The HTTP HEAD method is used to retrieve a metadata representation of the resource to include the HTTP headers. It is identical to the GET method without a message body in the response or the representation itself. |
| OPTIONS | The HTTP OPTIONS method provides information about the request and response options a client is allowed to do to a resource and the resource representations that are available. |
| TRACE | HTTP request method used for debugging which echo's back input back to the user. |

# JAX-RS Annotations

|  |  |  |
| --- | --- | --- |
| **Reference:** [**http://jsr311.java.net/nonav/releases/1.1/spec/spec3.html#x3-67000A**](http://jsr311.java.net/nonav/releases/1.1/spec/spec3.html#x3-67000A) | | |
| **Annotation** | **Target** | **Description** |
| Consumes | Type or method | Specifies a list of media types that can be consumed. |
| Produces | Type or method | Specifies a list of media types that can be produced. |
| GET | Method | Specifies that the annotated method handles HTTP GET requests. |
| POST | Method | Specifies that the annotated method handles HTTP POST requests. |
| PUT | Method | Specifies that the annotated method handles HTTP PUT requests. |
| DELETE | Method | Specifies that the annotated method handles HTTP DELETE requests. |
| HEAD | Method | Specifies that the annotated method handles HTTP HEAD requests. Note that HEAD may be automatically handled, see section 3.3.5. |
| ApplicationPath | Type | Specifies the resource-wide application path that forms the base URI of all root resource classes. |
| Path | Type or method | Specifies a relative path for a resource. When used on a class this annotation identifies that class as a root resource. When used on a method this annotation identifies a sub-resource method or locator. |
| PathParam | Parameter, field or method | Specifies that the value of a method parameter, class field, or bean property is to be extracted from the request URI path. The value of the annotation identifies the name of a URI template parameter. |
| QueryParam | Parameter, field or method | Specifies that the value of a method parameter, class field, or bean property is to be extracted from a URI query parameter. The value of the annotation identifies the name of a query parameter. |
| FormParam | Parameter, field or method | Specifies that the value of a method parameter is to be extracted from a form parameter in a request entity body. The value of the annotation identifies the name of a form parameter. Note that whilst the annotation target allows use on fields and methods, the specification only requires support for use on resource method parameters. |
| MatrixParam | Parameter, field or method | Specifies that the value of a method parameter, class field, or bean property is to be extracted from a URI matrix parameter. The value of the annotation identifies the name of a matrix parameter. |
| CookieParam | Parameter, field or method | Specifies that the value of a method parameter, class field, or bean property is to be extracted from a HTTP cookie. The value of the annotation identifies the name of a the cookie. |
| HeaderParam | Parameter, field or method | Specifies that the value of a method parameter, class field, or bean property is to be extracted from a HTTP header. The value of the annotation identifies the name of a HTTP header. |
| Encoded | Type, constructor, method, field or parameter | Disables automatic URI decoding for path, query, form and matrix parameters. |
| DefaultValue | Parameter, field or method | Specifies a default value for a field, property or method parameter annotated with @QueryParam, @MatrixParam, @CookieParam, @FormParam or @HeaderParam. The specified value will be used if the corresponding query or matrix parameter is not present in the request URI, if the corresponding form parameter is not in the request entity body, or if the corresponding HTTP header is not included in the request. |
| Context | Field, method or parameter | Identifies an injection target for one of the types listed in section 5.2 or the applicable section of chapter 6. |
| HttpMethod | Annotation | Specifies the HTTP method for a request method designator annotation. |
| Provider | Type | Specifies that the annotated class implements a JAX-RS extension interface. |

# HTTP Response Codes

The following table lists the intended HTTP Response Codes

List is not inclusive of all possible response Codes

|  |  |  |
| --- | --- | --- |
| **HTTP Status Code** | **Description** | **Reason** |
| 200 | OK | Request Succeeded |
| 201 | Created | Successful creation of resource |
| 202 | Accepted | The request has been accepted for processing, but the processing has not been completed. |
| 204 | No Content | The server has fulfilled the request but does not need to return an entity-body, and might want to return updated meta-information. |
| 301 | Moved Permanently | The requested resource has been assigned a new permanent URI and any future references to this resource SHOULD use one of the returned URIs. |
| 303 | See Other | The response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource. |
| 400 | Bad Request | The request could not be understood by the server due to malformed syntax. |
| 401 | Unauthorized | The request requires user authentication. If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials. |
| 403 | Forbidden | The server understood the request, but is refusing to fulfill it. |
| 405 | Method Not Allowed | HTTP Method not supported for requested URI |
| 406 | Not Acceptable | HTTP Accept header requested media type not available |
| 409 | Conflict | The request could not be completed due to a conflict with the current state of the resource. |

# Mock Database Source Code

Create a New Class in the mil.army.train.enterprise.service.model named *Database*.

Select the entire contents of the table below and replace the contents of the Database class.

Go Back to🡪

|  |
| --- |
| package mil.army.train.enterprise.service.model;  import java.util.ArrayList;  import java.util.Collection;  import java.util.Date;  import java.util.Iterator;  import java.util.List;  import java.util.Map;  import java.util.concurrent.ConcurrentHashMap;  import java.util.concurrent.atomic.AtomicInteger;  /\*\*  \* This is a crude attempt at a Database  \* Tables are nothing more than a ConcurrentHashMap  \*  \* @author Barbara.Craig  \*  \*/  public class Database {  private static Database INSTANCE = null;  private static Map<Integer,State> stateTable = new ConcurrentHashMap<Integer, State>();  private static AtomicInteger idStateSeq = new AtomicInteger();  //TODO add a user table  //private static Map<Integer,State> userTable = new ConcurrentHashMap<Integer, User>();    private Database(){  //Singleton  }    public static Database getInstance(){  if (INSTANCE == null){  INSTANCE = new Database();    loadStateData();  }  return INSTANCE;  }    /\*\*  \* findById() - Find record by Id  \* @param entity  \* @param id  \* @return  \*/  @SuppressWarnings("unchecked")  public <T> T findById(Class<T> entity, int id){  //State Table  if (entity.isAssignableFrom(State.class)){  return (T) stateTable.get(id);  }  return null;  }    @SuppressWarnings("unchecked")  public <T> List<T> findAll(Class<T> entity){  List<T> list = new ArrayList<T>();  Collection<T> table = null;    //State Table  if (entity.isAssignableFrom(State.class)){  table = (Collection<T>) stateTable.values();  }      if (table != null){  Iterator<T> iter = table.iterator();  while (iter.hasNext()){  list.add((T) iter.next());  }  }  return list;    }  public <T> Object findBy(Class<T> entity, String field, String value) {  T record = null;  List<T> data = findAll(entity);  if (field == null || value == null){  throw new RuntimeException("findBy() - Insufficient values - incoming parameters are null or invalid");  }      //filter by field  if (entity.isAssignableFrom(State.class)){  for( T aState : data){  if (field.equalsIgnoreCase("code") && ((State) aState).getCode().equalsIgnoreCase(value)) {  record = aState;  }  }  }    return record;  }    public int save(Object entity){  int id = -1;  if (entity instanceof State){  State exists = (State) findBy(State.class, "code", ((State) entity).getCode());  if (exists == null){  id = idStateSeq.incrementAndGet();  ((State) entity).setId(id);  ((State) entity).setModified(new Date());  stateTable.put(id, (State) entity);  }  }  return id;  }    public int saveOrUpdate(Object entity){  int id = -1;  if (entity instanceof State){  State exists = (State) findBy(State.class, "code", ((State) entity).getCode());  if (exists == null){  id = idStateSeq.incrementAndGet();  }else{  id = exists.getId();  }  ((State) entity).setId(id);  ((State) entity).setModified(new Date());  stateTable.put(id, (State) entity);  }  return id;  }  public <T> boolean remove(Class<T> entity, int id) {  boolean success = false;  if (entity.isAssignableFrom(State.class)){  State did = stateTable.remove(id);  if (did!=null && did.getId() == id){  success = true;  }  }  return success;  }    private static void loadStateData() {  State state = new State(idStateSeq.incrementAndGet(),"AL", "Alabama", new Date());  stateTable.put(idStateSeq.get(), state);  state = new State(idStateSeq.incrementAndGet(),"AR", "Arizona", new Date());  stateTable.put(idStateSeq.get(), state);    }  } |