**Exercise 6**

*Create an embedded Java REST Service using JAX-RS and Jetty*

**Prior Knowledge**

Basic understanding HTTP verbs, REST architecture

Some Java coding skill

**Objectives**

Understand what it takes to create REST services. Interact with a REST service using simple web clients in Chrome, on the command line.

See how Maven can be used.

**Software Requirements**

(see separate document for installation of these)

* Java Development Kit 8
* Apache Maven 3.3.9 or later
* Jetty and Jersey
* Eclipse
* curl
* Google Chrome/Chromium plus Chrome Advanced REST extension

**Overview**

There are many technologies for creating RESTful Web Services in Java. In order to create a simple approach, we are going to use the Java standard for creating REST services, which is called JAX-RS. The “official” Oracle implementation of JAX-RS is Jersey, although there are other implementations such as CXF which we used in Exercises 4 and 5.

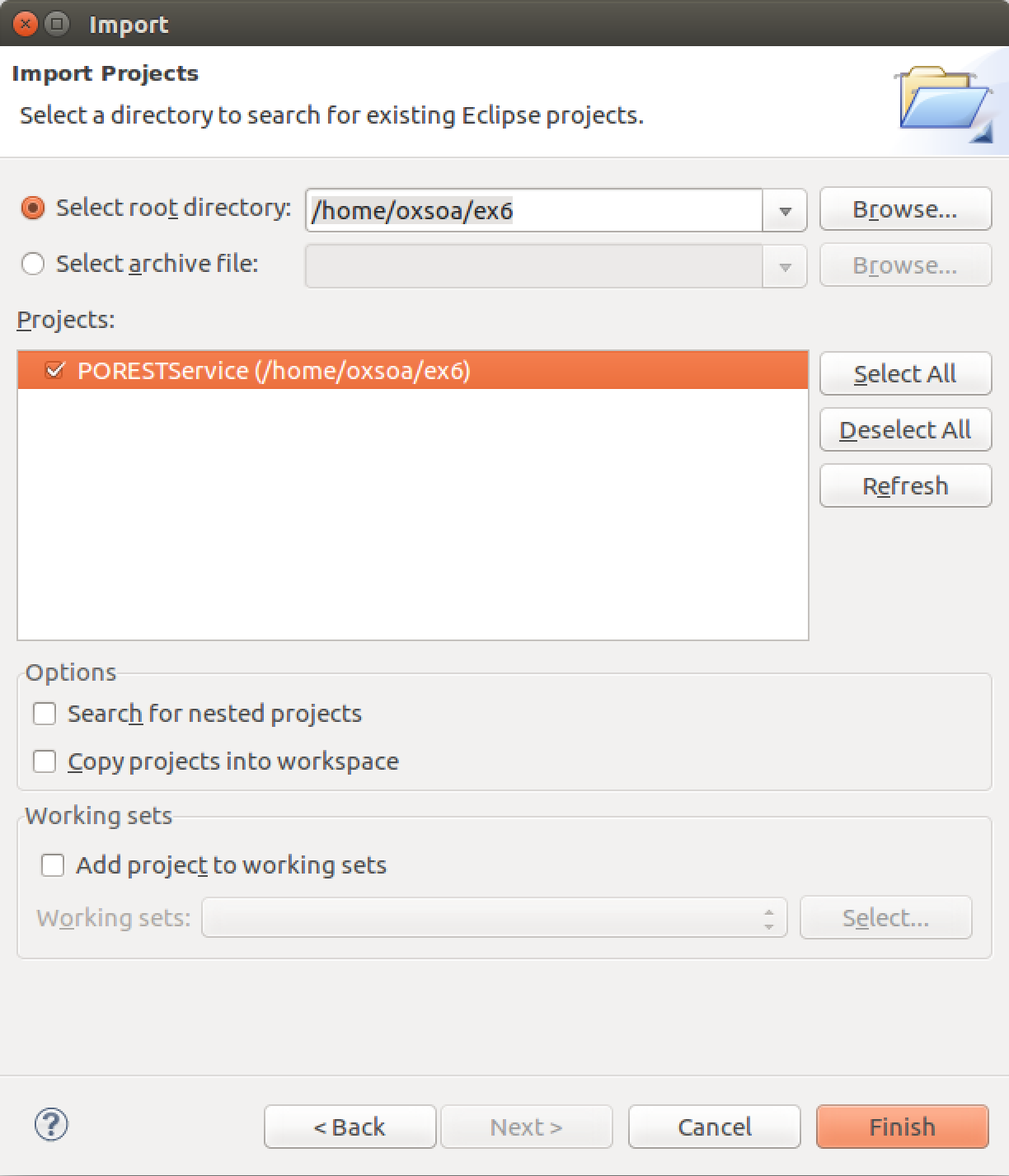
Jetty is a lightweight embeddable HTTP server that we will use to make these JAX-RS services available. We will also use a packaging model that creates “Uber JARs” that contain all the dependencies within them.

PART A. **Create a new Maven project**

Maven is a very powerful (and somewhat arcane) build tool. We are going to use Maven to build our simple RESTful service project.

Since Maven is quite so fiendish, we are going to start with a usable Maven config that does the following:

* Downloads any required dependencies
* Includes Jersey, Jetty, and the bridge between the two.
* Packages the result up as an “uber-jar” that can be run standalone

1. First start a Unix Terminal window.  
   Now create a directory to store your code in.  
     
   mkdir ~/ex6
2. Change to that directory  
   cd ~/ex6
3. Test that you have maven properly installed. Execute  
   mvn –v  
     
   You should see something similar to this (dependent on your machine, JVM, etc)  
     
   Apache Maven 3.3.9  
   Maven home: /usr/share/maven  
   Java version: 1.8.0\_91, vendor: Oracle Corporation  
   Java home: /usr/lib/jvm/java-8-openjdk-amd64/jre  
   Default locale: en\_GB, platform encoding: UTF-8  
   OS name: "linux", version: "4.4.0-22-generic", arch: "amd64", family: "unix"
4. Grab the prewritten Maven config (pom.xml) from Github:  
   TODO add correct URL
5. From the ex6 directory, make a directory for your source code.   
   mkdir –p src/main/java
6. Let Maven create you an eclipse project definition:  
     
   mvn eclipse:eclipse
7. Now we can import the project into Eclipse.  
     
   In Eclipse: **File->Import.. General->Existing Projects Into Workspace**Browse to the ex6 directory and then click **Finish**  
     
     
   *Note that unlike creating a project directly in Eclipse, where the files are stored in ~/workspace, the Eclipse project will work off of your files here. Just to be clear, if you delete the Eclipse project files, these files will disappear and vice-versa.*
8. We now need to create two classes to implement our first Java service.  
   The first class is called freo.me.rest.Main and this is the class that will configure the system and start up the Jetty server. This class is almost all “boilerplate” just like the pom.xml. In other words you can take this and the pom.xml and re-use them for other projects with very minimal changes.
9. First create a new package in Eclipse under src/main/java:  
   freo.me.rest
10. Create a Main.java class in that package, and copy and paste the code from here:  
    TODO add URL
11. The Main class has the following code:   
    URI baseUri = UriBuilder.fromUri("http://localhost/").port(8000).build();  
    *This is fairly self-explanatory. You can define the URL on which the server will listen.*ResourceConfig config = new ResourceConfig(POResource.class);  
    *This is where we identify that the class POResource is the JAX-RS Resource (aka Service) that we want to expose.*Server server = JettyHttpContainerFactory.createServer(baseUri, config);  
    *Here is where we create the Jetty Server object.*

try {

server.start();

*This initiates the startup of the server.*

server.join();  
 *Wait until the server finishes initiation*

} finally {

server.destroy();

}

1. You will have noticed that we referenced a class *POResource.class* which is so far absent. Let’s create it. This is where we finally get to use the JAX-RS specification to define an HTTP-based Resource.
2. Here is the code listing for POResource.java. You should create the class in the same package as Main. Type this in or cut and paste from here:  
   TODO URL

package freo.me.rest;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.core.MediaType;

@Path("purchase")

public class POResource {

@GET

@Produces(MediaType.TEXT\_PLAIN)

public String get() {

return "Hello!";

}

}

1. Mvn clean package in Eclipse
2. Mvn exec:java in Eclipse
3. Run as a java –jar externally
4. Test

This will build **and test** the sample code. Its pretty cool. It actually starts an embedded Tomcat to run the service and call unit tests against it.

f. Assuming your build worked just fine, you now have a WAR file (Web Application aRchive) that you can deploy in Tomcat. Check that there is a file:

~/oxsoa/HelloWorld/target/HelloWorld-1.0.war

g. Tomcat is already installed on your VM, in the directory

**~/servers/tomcat/**

h. Install your webapp (do the following all on one command line!)

cp ~/oxsoa/HelloWorld/target/HelloWorld-1.0.war ~/servers/tomcat/webapps

i. Start Tomcat so it runs on the command line (so you can see the logs)

From the tomcat directory:

cd ~/servers/tomcat

bin/catalina.sh run

j. Try your REST service

Browse

<http://localhost:8080/HelloWorld-1.0/hello/echo/paul>

You should see “paul”.

You can also try this as a command line:

curl <http://localhost:8080/HelloWorld-1.0/hello/echo/paul> -v

You should see something like:

ox-soa@oxsoa-2014:~$ curl http://localhost:8080/HelloWorld-1.0/hello/echo/paul -v

\* Hostname was NOT found in DNS cache

\* Trying 127.0.0.1...

\* Connected to localhost (127.0.0.1) port 8080 (#0)

> GET /OrderService-1.0/hello/echo/paul HTTP/1.1

> User-Agent: curl/7.35.0

> Host: localhost:8080

> Accept: \*/\*

>

< HTTP/1.1 200 OK

\* Server Apache-Coyote/1.1 is not blacklisted

< Server: Apache-Coyote/1.1

< Date: Sun, 23 Nov 2014 21:02:37 GMT

< Content-Type: text/plain

< Content-Length: 4

<

\* Connection #0 to host localhost left intact

paulox-soa@oxsoa-2014:~$

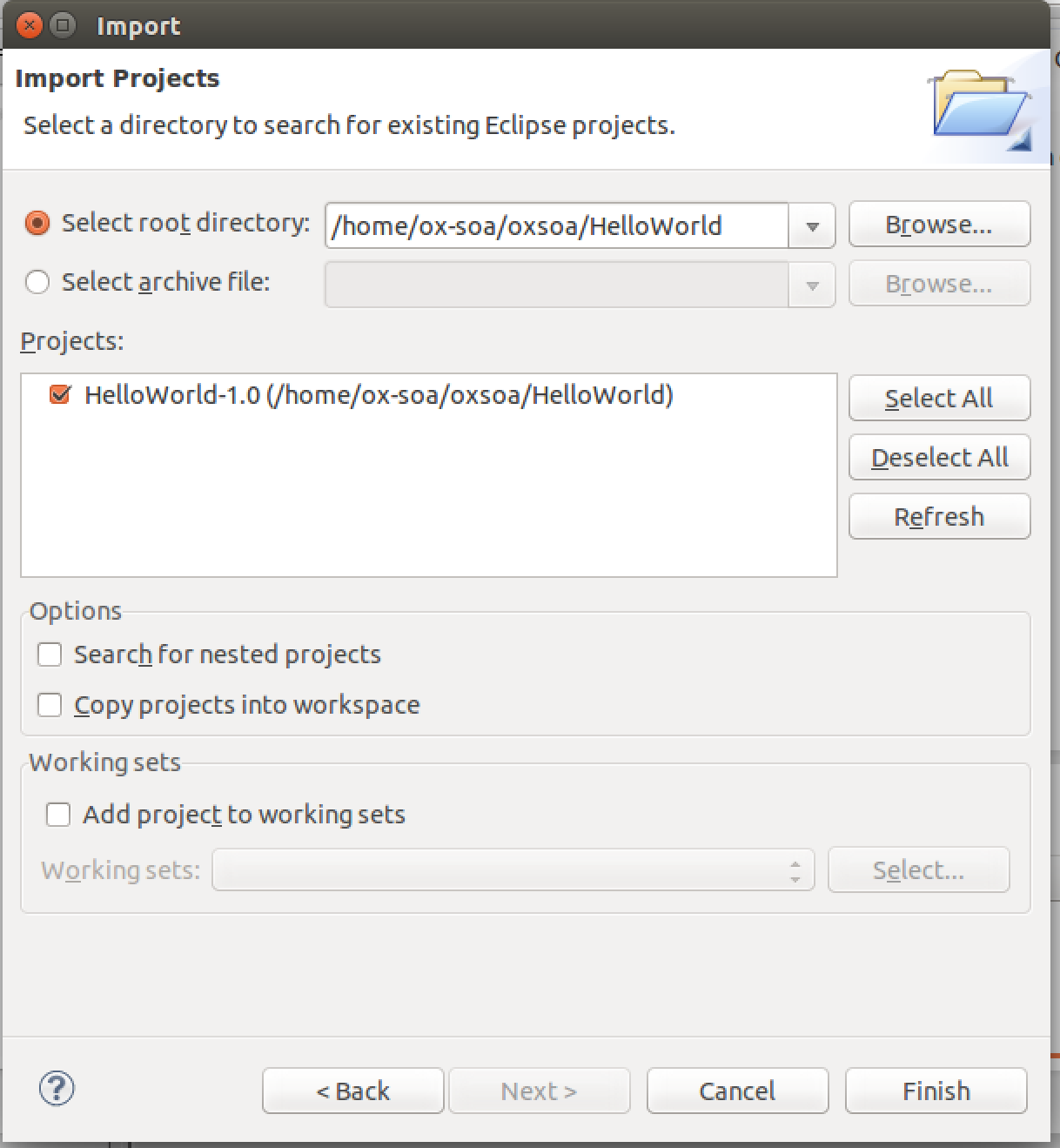
f. You can also build the Eclipse project for this too:  
In the ~/ox-soa/HelloWorld directory type

mvn eclipse:clean eclipse:eclipse

This creates a project file that you can import into Eclipse with the right classpath, settings, etc.

Now import the project. To do this, in Eclipse:

**File -> Import -> General/Existing Projects Into Workspace->[Choose the directory where HelloWorld service is]->Finish**



Now you should have the project installed in your Eclipse and be able to edit and build it. **Take a look at the sample hello service.**

Now we are ready to build our own RESTful Service.

**Step 2. Creating the OrderService**

Rather than spending a lot of time writing Java, which is not the main point of this exercise, you should focus on the REST and HTTP aspects of this. I have ready written a set of Java classes that:

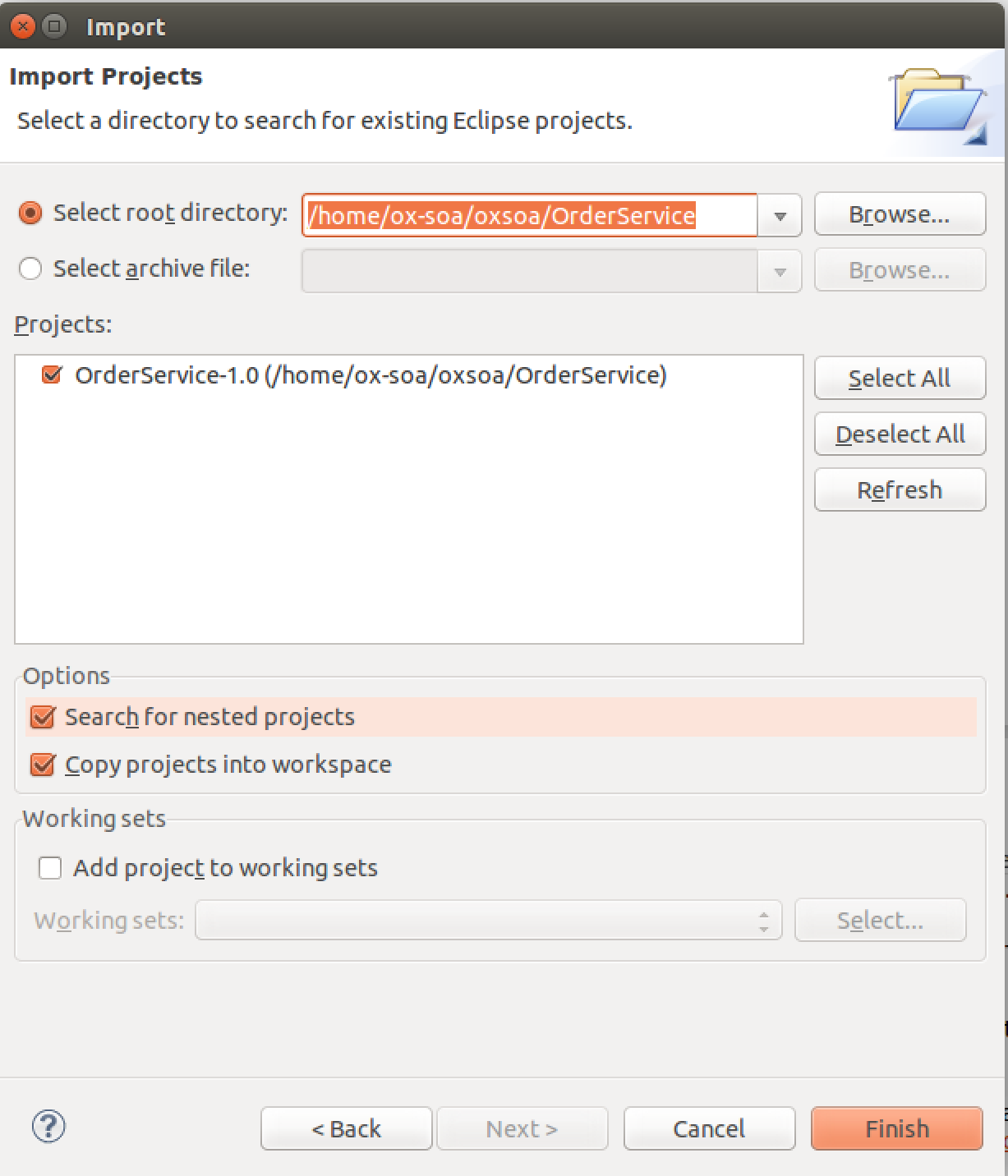
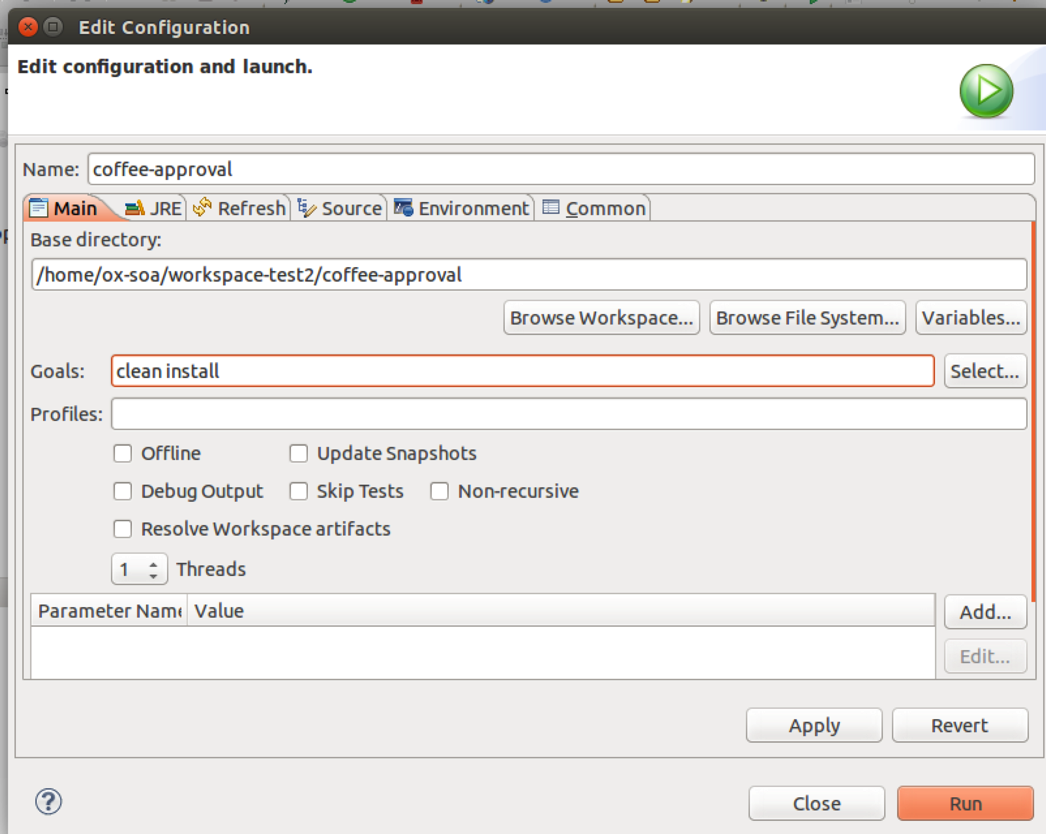
1. Implement a simple “Order Scenario”
2. Serialize and Deserialize as JSON

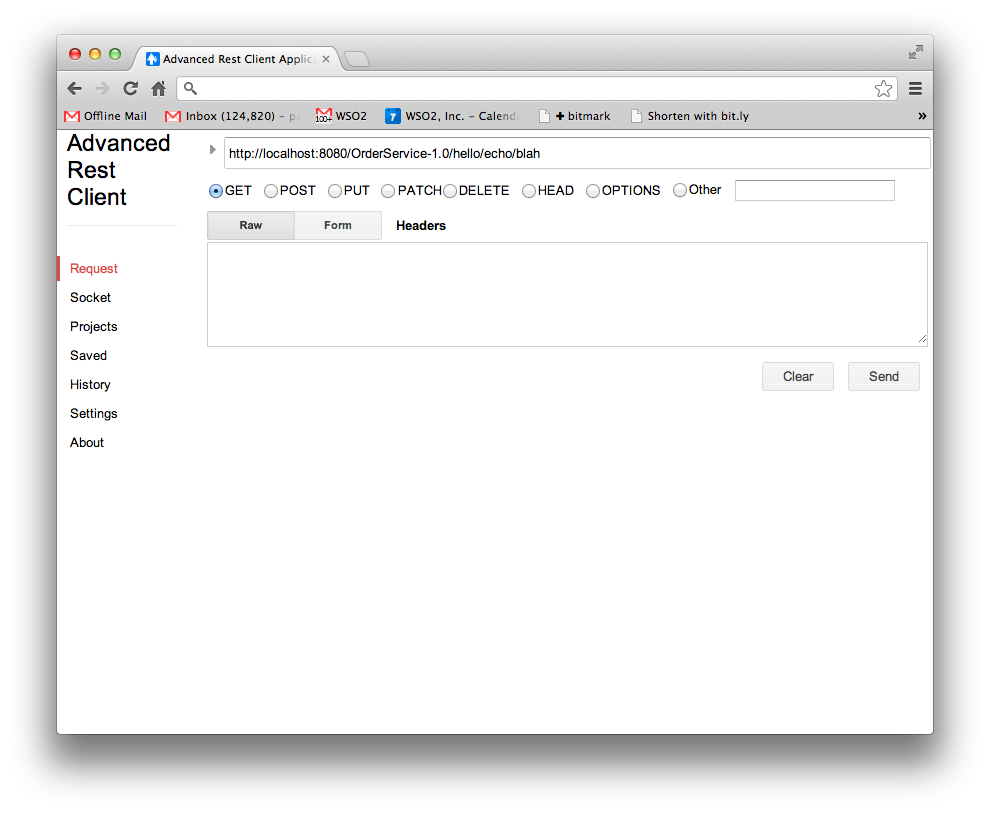
I have also written a test case that validates a service interface.

Your aim is to create a Java Service that utilizes the existing code, and is correctly annotated using JAX-RS annotations so that it meets the test case.

Here is a rough set of documentation that explains the service interface.

|  |  |  |  |
| --- | --- | --- | --- |
| Method | URI template | Description | Supported encoding |
| GET | /orders | Get a list of href links to available orders  If no orders are on the system, return an empty list. | Produces application/json |
| GET | /orders/{id} | Get back a representation of order with identifier id.  If no such order is yet in the system, returns HTTP Not Found  If the order previously existed but has been deleted, returns HTTP Gone | Produces  application/json |
| POST | /orders | Passes a representation of the order and create a new entry in the order database.  On success returns HTTP 201 Created and an HTTP Location header containing the URI of the resulting order | Consumes  application/json |
| PUT | /orders/{id} | Updates an existing order  On success return HTTP 200 OK  If no such order is yet in the system, returns HTTP 404 Not Found  If the order previously existed but has been deleted, returns HTTP 410 Gone | Consumes  application/json |
| DELETE | /orders/{id} | Marks an order as deleted  Returns HTTP 200 OK on success  If no such order is yet in the system, returns HTTP Not Found  If the order previously existed but has been deleted, returns HTTP Gone | No body content |

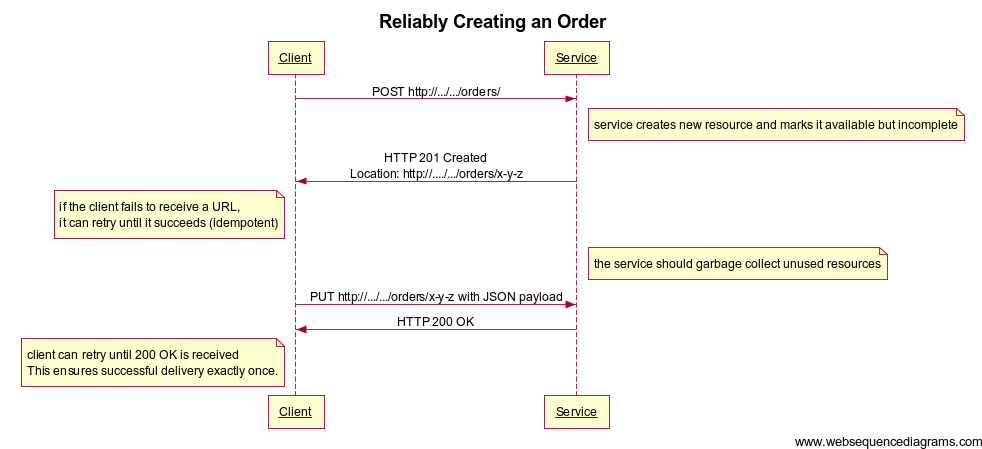
1. Unzip the code:  
   cd ~/oxsoa/   
   unzip ~/Downloads/OrderService-project-code.zip  
     
   PS the file is also here: <https://github.com/pzfreo/ox-soa/blob/master/lab-exercises/code/OrderService-project-code.zip?raw=true>
2. This is an Eclipse project.   
   Import it into the Eclipse Workspace as you did before:  
   
3. Build the project using **mvn clean install** *Hint: you can run* mvn clean install *inside Eclipse.*Right click on the pom.xml file in your Eclipse session, **Run As -> Maven Build.**   
     
   The first time you do this, enter the **Goals** as **clean install**. Future times it will remember this. If this just runs without asking you for Goals, then the project is already set up correctly so don’t worry.   
     
   The build will **FAIL**. That is on purpose. Your task is to sort it out!  
     
   ****
4. Look at the following class in your workspace  
   me.freo.rest.OrderService
5. Now you can incrementally add the correct methods and annotations for get/post/put/delete until the test case is met.

1. ONCE YOU GET YOUR BUILD WORKING:  
     
   Re-install the webapp into Tomcat. (It should hot deploy if you’ve left Tomcat running)
2. Now you can test it in Chromium, using the Advanced REST Test Client.  
   Start Chromium and get up a blank/empty page (Command-N or Ctrl-N)
3. Click on the Apps link at the top of the page:  
     
   You should now see the following icon:  
    
4. Click on that. You should see a screen like this:  
   By now you should understand the REST patterns and URLs well enough to be able to test our your app. It should be available at:  
   <http://localhost:8080/OrderService-1.0/orders/>
5. Also try interacting with it using curl commands. See if you can automate posting an order.

**EXTENSION**

The model we have built does not support reliability. It is impossible to know if you have successfully created a new order. To fix this, we need to implement a different pattern. In this pattern, the POST is empty, and returns a location, and only when the POST has successfully returned a Location and the client has the location, do we PUT an order into the existing location.

Here is a sequence diagram showing this model.



Implement this model by changing the code.