Exercise 6a

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 Gradle can be used.

Software Requirements
(see separate document for installation of these)

- Java Development Kit 8
- Gradle build system
- Jetty and Jersey
- Eclipse Luna and Buildship
- 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, both as a WebApp and embedded.

Create and test a new Gradle project

Gradle is a very powerful build tool. It can be seen as a more effective and easier alternative to the Maven build system. Maven is brilliant but incredibly painful! For those of you who normally use Maven, I have included a sample Maven file for comparison here:

We are going to use Gradle to build our simple RESTful service project.

We are starting with a basic Gradle build file that does the following:

- Downloads any required dependencies.
- Includes Jersey, Jetty, and the bridge between the two.
- Runs the Jetty server to test the generated WAR file.

Steps

1. First start a Unix Terminal window.
   Now create a directory to store your code in.

   ```
   mkdir -p ~/ex6/POResource
   ```

2. Change to that directory
   ```
   cd ~/ex6/POResource
   ```

3. Test that you have gradle properly installed. Execute
   ```
   gradle -v
   ```

   You should see something similar to this (dependent on your machine, JVM, etc)

4. Grab the prewritten Gradle config from Github and save into ~/ex6/POResource/build.gradle:

```
cd ~/ex6/POResource
curl -L  http://freo.me/ex6-gradle -o build.gradle
```

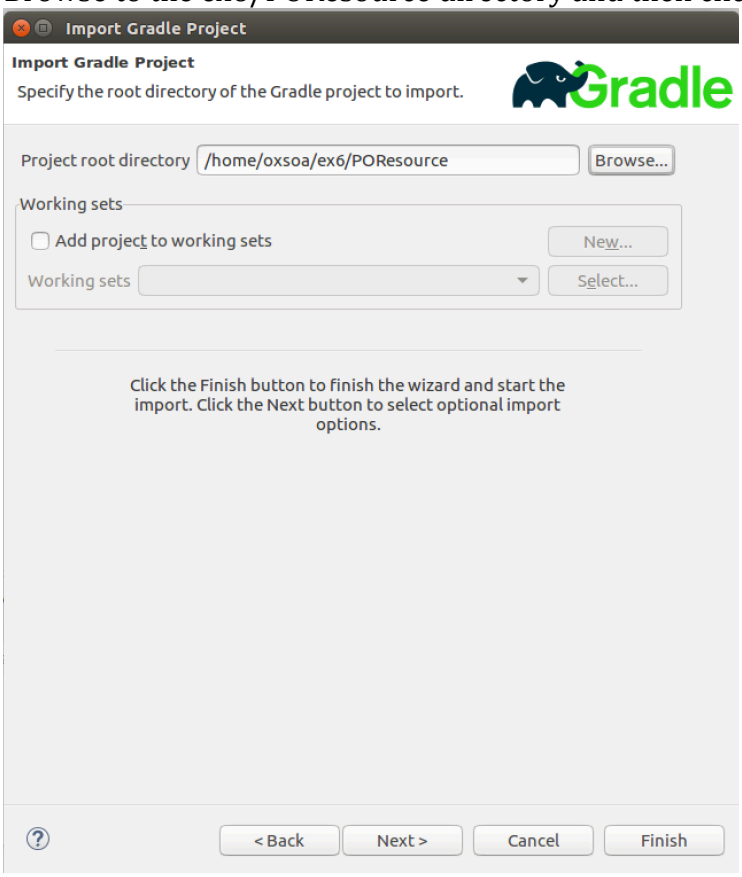
Please read the build file. It should be pretty self-explanatory.

5. From the **ex6/POResource** directory, make some directories for your source code.

```
mkdir -p src/main/java
mkdir -p src/test/java
mkdir -p src/main/webapp/WEB-INF
```

6. Now we can import the project into Eclipse. This will use the Gradle Buildship plugin for Eclipse which is already installed in Eclipse.

7. In Eclipse: **File->Import.. Gradle->Gradle Project**
Browse to the ex6/POResource directory and then click **Finish**

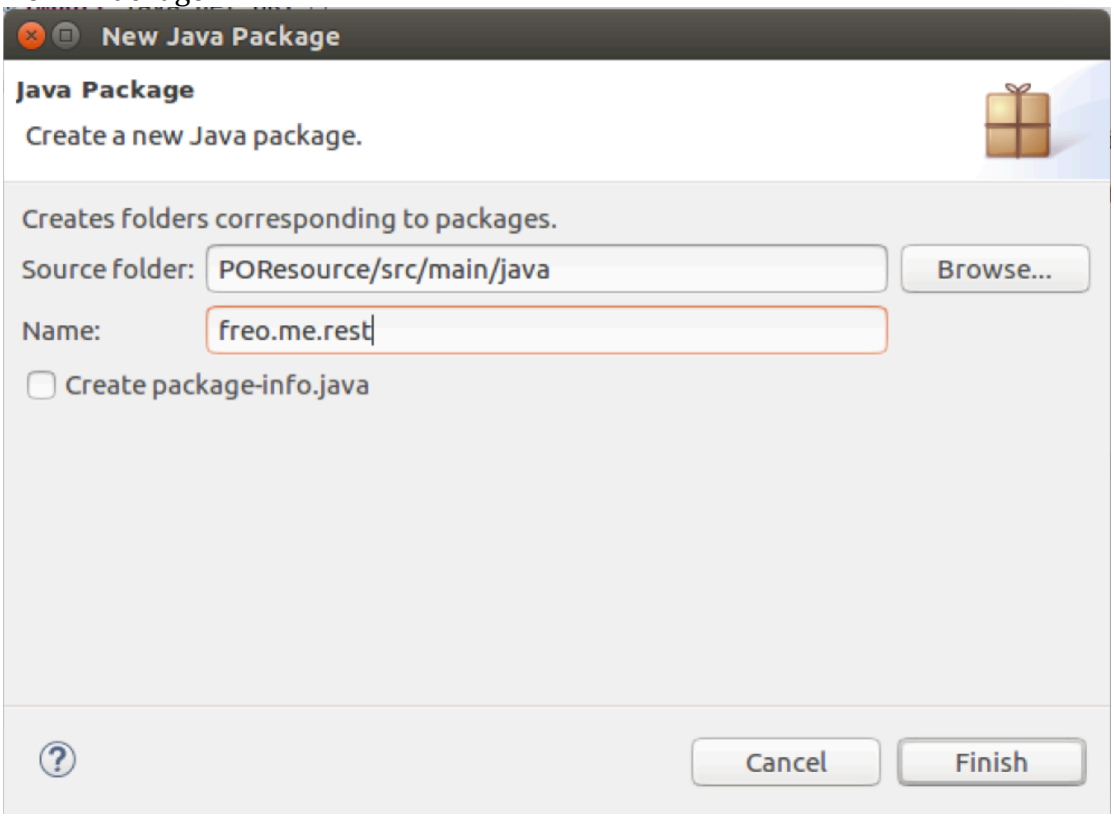


Note that unlike creating a project directly in Eclipse, where the files are stored in ~/workspace, the Eclipse project will work off of your files here. Just to be clear, if you delete the Eclipse project files, these files will disappear and vice-versa.

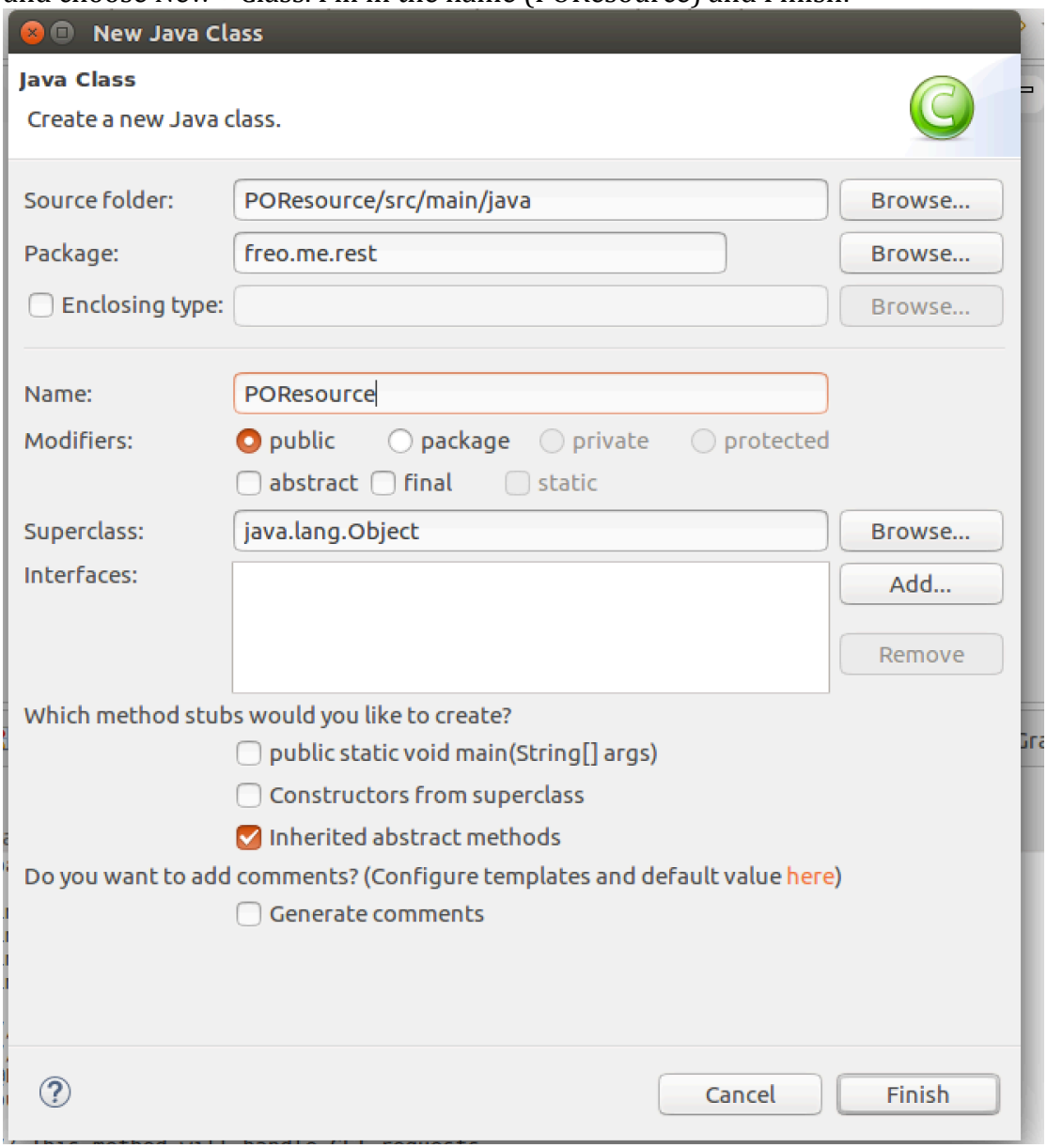
8. We now need to create a class and a deployment descriptor to implement our first Java service.

9. First create a new package in Eclipse under src/main/java:
   `freo.me.rest`

   The easiest way is to right-click on src/main/java and then choose New->Package



10. Create a POResource.java class in that package. Right-click on the Package and choose New->Class. Fill in the name (POResource) and Finish:



11. Here is the code listing for POResource.java. This version has some simple explanations of each part.
    Type this in or cut and paste from here:
    http://freo.me/ex6-java

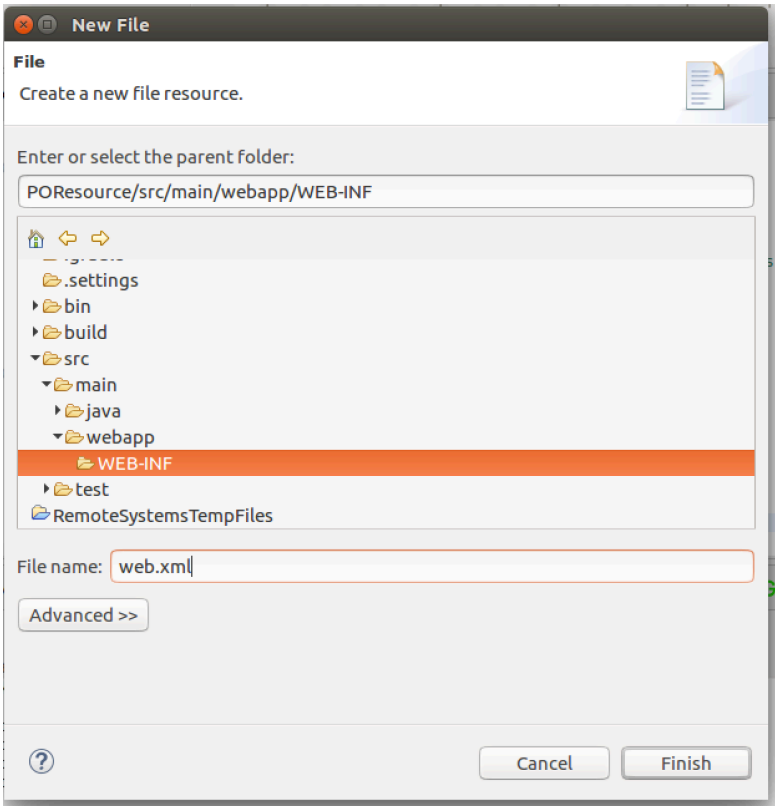
```
package freo.me.rest;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

// this will be the HTTP URL sub path from the Jetty server's URI where this
// resource/service will be available
@Path("purchase")
public class POResource {

// This method will handle GET requests
    @GET
// Specify the resulting content type
    @Produces(MediaType.TEXT_PLAIN)
    public String get() {
        return "Hello!";
    }
}
```

12. We also need to tell our Java runtime about this class. There are multiple ways to do this.
(documented here:
https://jersey.java.net/documentation/latest/deployment.html)

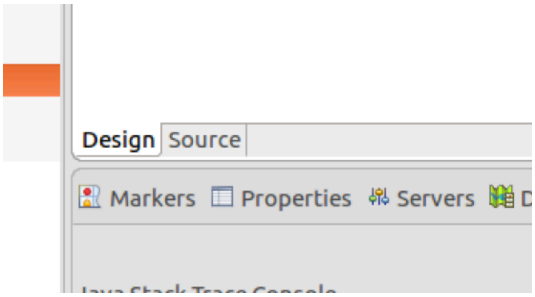
    For this stage of our exercises we are going to create a WebApp and a corresponding WAR file deployment.

    This requires a web.xml deployment descriptor. You can create that by:

    Navigate to **src/main/webapp/WEB-INF**
    Right-click **New->File**
    File name **web.xml**



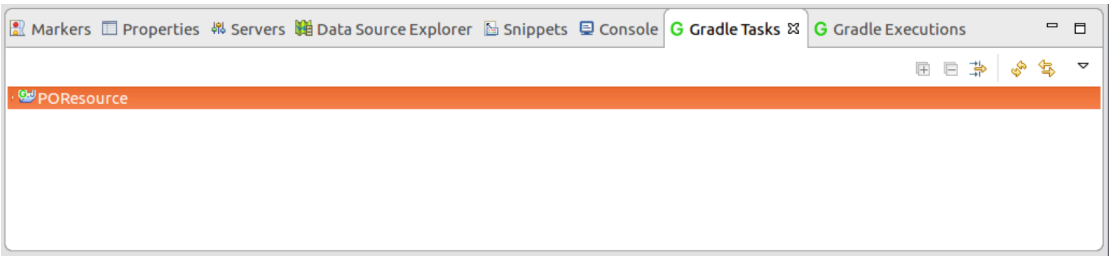
13. Switch to the Source view of this file:



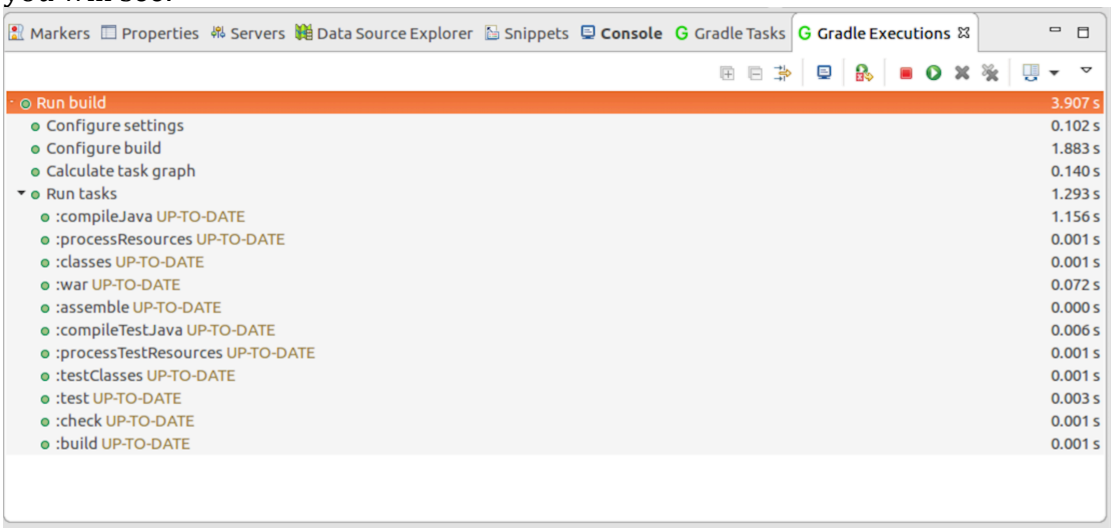
14. Cut and paste (from here: http://freo.me/ex6-webxml) the required xml. Here is a copy as well:

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://java.sun.com/xml/ns/javaee"
	xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
	http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
	id="WebApp_ID" version="3.0">

	<display-name>Jersey JAXRS</display-name>
	<servlet>
		<servlet-name>JerseyServlet</servlet-name>
		<servlet-
		class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
		<init-param>
			<param-name>jersey.config.server.provider.packages</param-
			name>
			<param-value>freo.me.rest</param-value>
		</init-param>
		<load-on-startup>1</load-on-startup>
	</servlet>
	<servlet-mapping>
		<servlet-name>JerseyServlet</servlet-name>
		<url-pattern>/*</url-pattern>
	</servlet-mapping>
</web-app>
```

15. We are now ready to build this. If you look at the bottom pane, you should see a tab called Gradle Tasks. Click on it.



16. Now expand the POResource object and find **build/clean.** Click on it. Then **build/build**. The pane should change to Gradle Executions where you will see:



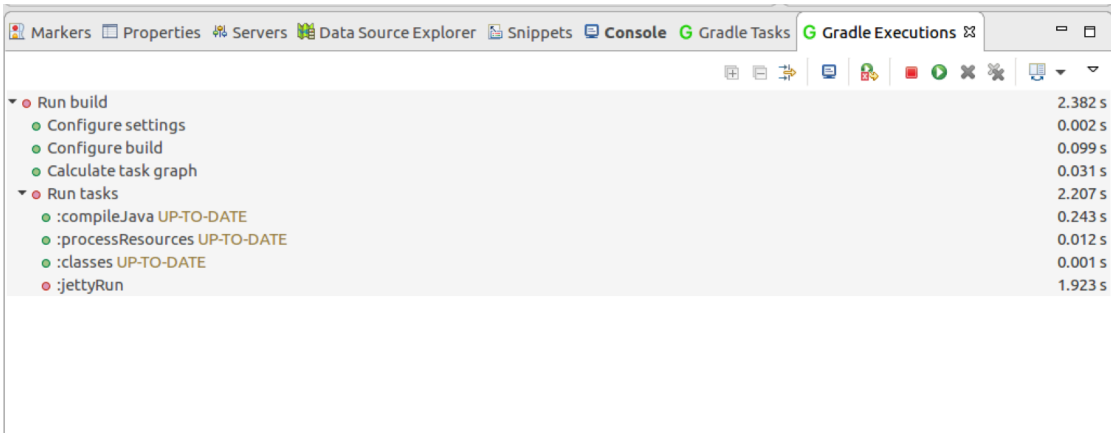
Hint: since this Eclipse project is directly linked to your source folder (~/ex6/POResource) you can also build from the command line by typing

```
gradle clean build
```

17. We can now run this by choosing the Gradle Task:
web application/jettyRunWar

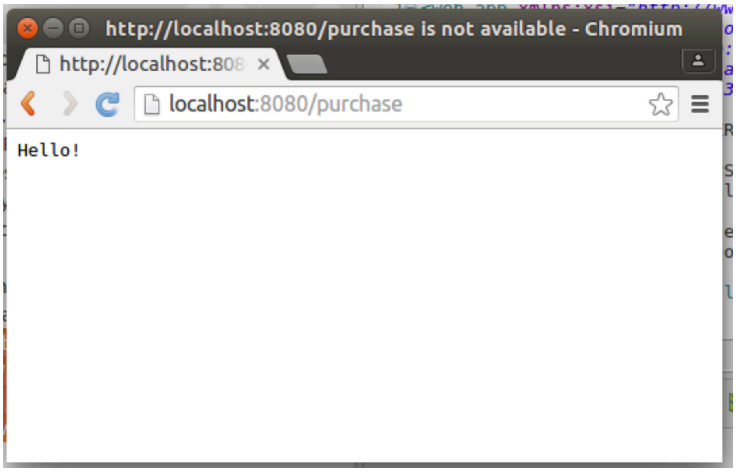
Or from the command line: `gradle jettyRunWar`

Because this is continuous, the Gradle Eclipse plugin will mark this as still running (red dot) rather than successfully completed (green dot)



18. Test this by going to the browser and browsing:
http://localhost:8080/purchase

You should see:

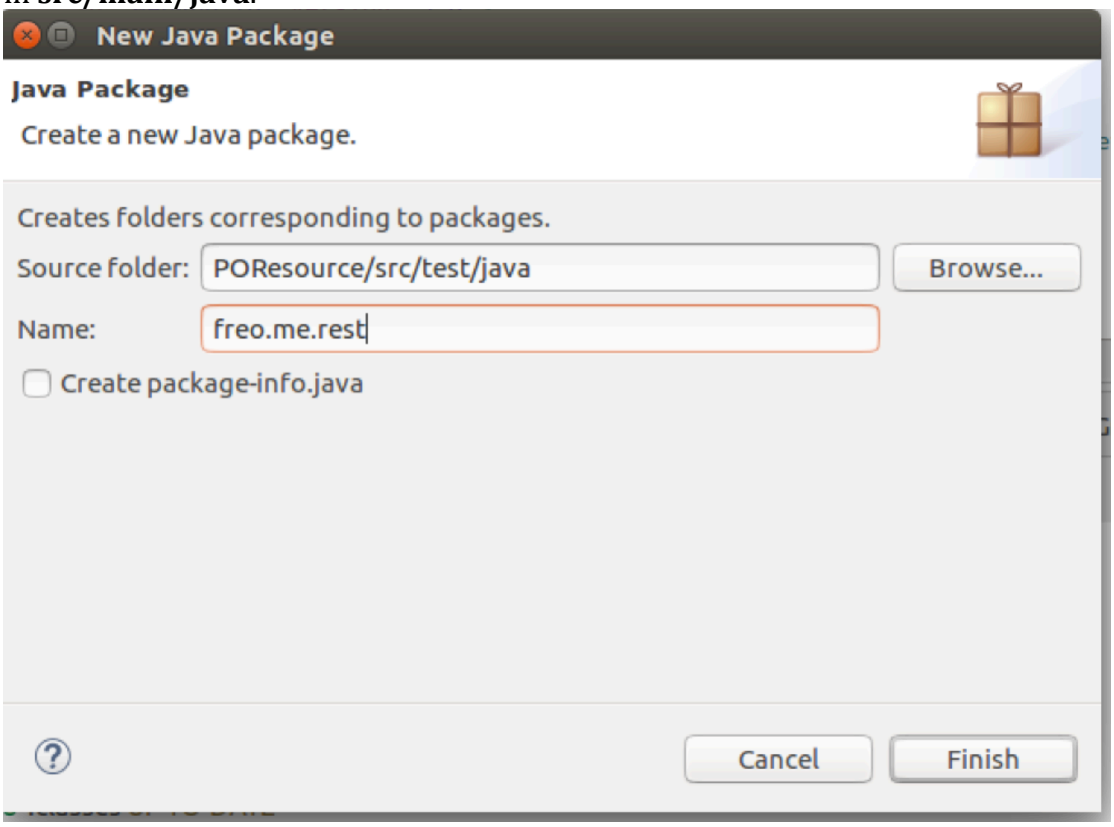


19. Stop jetty by using the gradle Task **jettyStop**

20. **Automated test**
    We would like to test this with an automated test. While I prefer the syntax and style of Frisby.js, there is poor integration between Frisby (see Exercise 1) and Gradle, so we will stick to JUnit (Java) testing instead.

    However, if you had a continuous integration server running, you could potentially use that to run Gradle and Frisby side-by-side.
    It is certainly possible to use Frisby within Gradle too, but the project that supports it has been discontinued, so you'd probably have to use Gradle's support for calling shell scripts.

21. First create a package in **src/test/java** that matches the one you created in **src/main/java**.



22. Create a new class **POResourceTest** in that package.
23. Use the following test code (from http://freo.me/ex6-test)

```
package freo.me.rest;
import org.junit.Test;
import static org.junit.Assert.*;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class POResourceTest {
    @Test public void testGet() {

      Client client = ClientBuilder.newClient();
      WebTarget target =
       client.target("http://localhost:8080").path("purchase");
      Response response =
         target.request(MediaType.TEXT_PLAIN).get();
      assertEquals(response.getStatus(),
      Response.Status.OK.getStatusCode());
      assertEquals(response.readEntity(String.class),"Hellooo!");
    }
}
```

This code is fairly simple and self-explanatory. It combines JUnit assertions together with JAX-RS client code to call HTTP and test the return code and value of the response.

24. Try this test by doing another gradle clean / gradle build

25. Why is the test failing?

26. Fix the test (or the code) so that the build succeeds.

27. That is the end of this exercise. Congrats.

28. Recap:

    We have created a very simple skeleton project (build.gradle, web.xml, directory structure) together with the simplest possible JAX-RS class (POResource.java). We have tested this manually and using automated tests. This is a great foundation for the next steps. The next steps are to create a proper service / resource that does something useful, and to test it, which is coming up in the next exercise.