Exercise 6b

Evolving our Java REST service into something good

Prior Knowledge
Basic understanding HTTP verbs, REST architecture
Some Java coding skill
Exercise 6a

Objectives
Develop a good understanding of JAX RS assertions.
Understand the Richardson Maturity Model

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

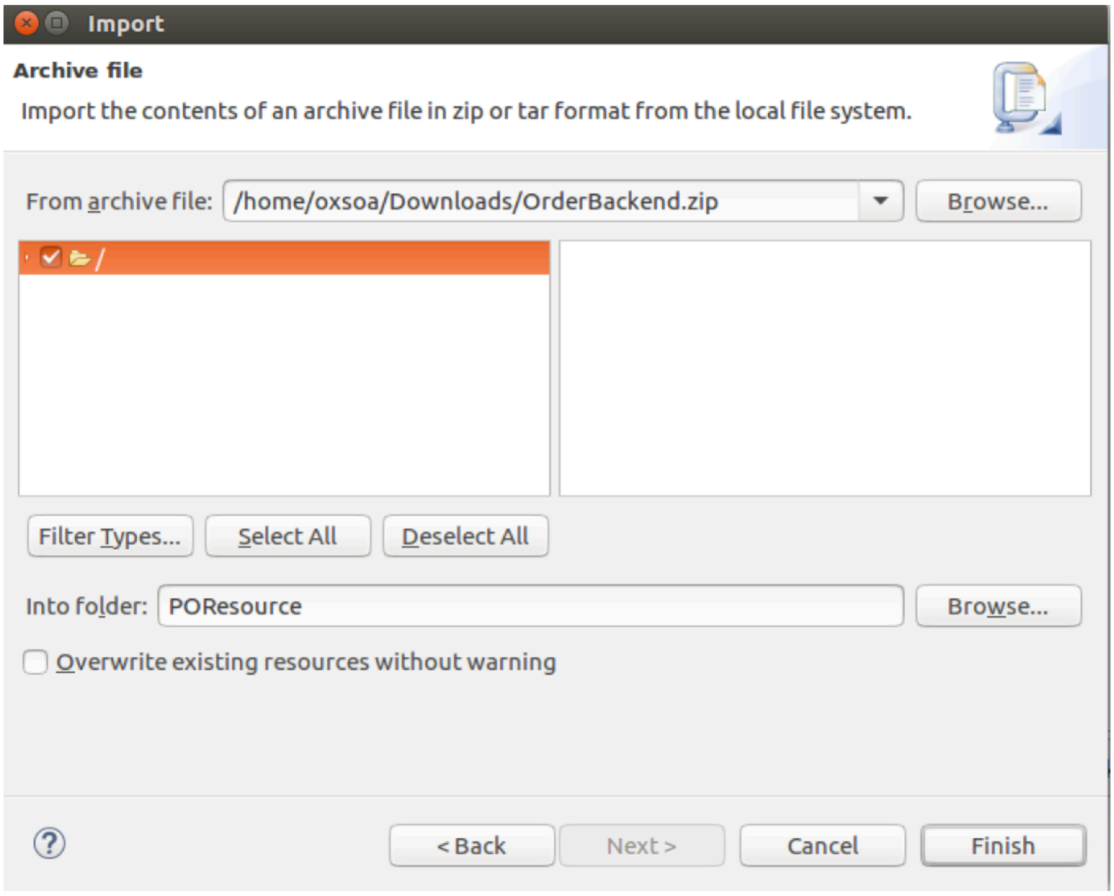
Overview
The service we developed in Exercise 6 was frankly pretty useless. It was designed to show a simple framework for building RESTful services in Java and to introduce the build system. Now we need to turn this into something useful.

Steps

1. Download the backend code:
   `curl -L http://freo.me/ex6-backend -o ~/Downloads/restbackend.zip`

2. Import some pre-written backend code:
   From project **POResource**
   Right-click, **Import.. General->Archive File->Next**

3. Browse to file **~/Downloads/restbackend.zip** and make sure the import destination is **POResource**:



Import

Archive file

Import the contents of an archive file in zip or tar format from the local file system.

From archive file: /home/oxsoa/Downloads/OrderBackend.zip

☑ /

Into folder: POResource

☐ Overwrite existing resources without warning

4. You will see some project errors. This is because the Eclipse project has not got the right dependencies defined.

5. In the existing POResource project, go to the build.gradle and add the following dependency (in the obvious place!):
   compile 'org.json:json:20160212'

6. Now right-click on the project **POResource** and choose **Gradle->Refresh Gradle Project**. This should now look clean.

7. Take a look at the new code. There are three main things to note:
    a. I have chosen to use my own serialization into and out of JSON. Jersey, JAXRS and Java have many ways of serializing into and out of JSON, XML and other formats. I recommend you look up Moxy and Jackson as two approaches. However, this approach is perfectly simple.

    b. As a result of this, my "bean" depends on org.json.*.
    This is a trade-off.

    c. There is a class OrderInMemory which implements 5 main methods (in pseudo-code)
        i. orderid = createOrder(JSON)
        ii. updated = updateOrder(orderid, JSON)
        iii. JSON_array = getOrders()
        iv. JSON = getOrder(orderid) throws NotFoundException
        v. deleted = deleteOrder(orderid)

8. This class is designed to be used as a singleton. This is not a serious implementation model but it saves us needing a real backend database at this stage.

    You can instantiate this class in POResource with the following code.

```
OrderInMemory orderSingleton = null;
 public POResource() {
     orderSingleton = OrderInMemory.getInstance();
}
```

9. Add this code into your **POResource** java class.

10. You will need to take a look at the OrderInMemory class to understand it and then use its methods to enhance POResource.java, and therefore take the implementation through the 4 levels of the RMM.

11. *POX level (Level 0)*

    Let's first support creating an Order.
    In the Richardson Maturity Model, the first stage is to allow POX (Plain Old XML) or in our case, Plain Old JSON. The simplest example of this will be that we post a JSON to the endpoint, and we get back a JSON with some kind of response in it.

12. I have created a Test case for this. The test cases should have loaded alongside the other files you installed but they are currently all commented out. I'd like you to:

a. Uncomment out the Level0Test.java test case.

   Hint: Ctlr-A, Ctrl-/ will comment / uncomment the whole file.

b. Review it to understand the behaviour that you are required to implement.

c. Create a method inside POResource that handles POST calls and correctly passes the test case.

d. You might want to review the JAX-RS presentation once again.

e. When reviewing the tests, if you need to better understand the JAX-RS client model, there is excellent documentation under Jersey: https://jersey.java.net/documentation/latest/client.html

13. You should have all you need to now code the updated POST logic and pass Level0Test.

Level 1
Resource level – improving POST to create a resource

There are two key problems with the simple POST code that you have just created.

Firstly, it uses return codes built into the JSON response, rather than HTTP level return codes.

Secondly, it does not create a new Resource to manage the order. The ideal behavior is that the POST would create a new resource (e.g. http://localhost:8080/purchase/0123-456-789) which was unique to this order. The return code should be 201 Created, and the POST body would usually contain the server's representation of the resource.

14. Uncomment and review Level1Test.java.

15. Comment out Level0Test.java as this is now superseded.

    Implement the functionality required to meet Level1Test.java

    Hint:
    The following code demonstrates how to create and return a location in JAX RS neatly. There are a bunch of less beautiful ways.

    ```
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response
      createOrder(String input, @Context UriInfo uriInfo)
    {

      // logic to create Order

      UriBuilder builder = uriInfo.getAbsolutePathBuilder();
      builder.path(orderId);
      return Response.created(builder.build()).
        entity(/*some content*/).build();

    }
    ```

    Hint #2:
    You will need to handle the error case as well.

    Hint #3:
    If you are having trouble deciding if the error is happening in the client or server, you could employ MITMDUMP. I highly recommend this in any case to visualise and understand the flows.

16. Having successfully created a new resource, we now need to be able to interact with it. This means that we need to support some more HTTP verbs, specifically GET, PUT, DELETE. This is level 2 of the Maturity Model.

HTTP Verbs
(Get / Put / Delete)

17. We now need to enable the correct tests for this level.
    a. Comment out the POResourceTest (which in now out of date).

    b. Uncomment Level2Test.java

    c. You can keep Level1Test.java running – the POST logic should survive the improvements all the way to Level 3.

    d. You are aiming to support the following logic:

POST (already implemented)	/	Passes a representation of the order and create a new entry in the order database. On success: HTTP 201 Created Location header - URI of the new Resource The server's representation of the resource is returned Returns HTTP 400 Bad Request if bad JSON request sent	Consumes application/json Produces application/json
GET	/{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
PUT	/{id}	Updates an existing order On success return HTTP 200 OK, together with the server's representation of the updated order. If the JSON request is bad, return HTTP 400 Bad Request 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 Produces application/json
DELETE	/{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 410 Gone	No body content

All this logic is also documented in the corresponding test cases.

18. You will now need to access the incoming {id} in the URL Path.
    To do this, you need to use two assertions.
    The first assertion identifies the part of the path you are interested in and names it. The second assertion maps that name into your Java parameters.

    Here is a code snippet:

```
@GET
@Path("{id}")
public Response getOrder(@PathParam("id") String id) {
      // now you can use id in your logic

}
```

    You should now be able to create the required methods to support GET, PUT and DELETE.

Hypermedia (and Get all orders)

19. Our RESTful journey is nearly complete. In order to implement HATEOAS we need to support some links. A more developed HATEOAS application would offer links to other services/resources/APIs. For example, once the purchase order has shipped, we could include a link to the shipping tracker.

    However, there is one simple resource we can offer, which is to provide a resource that "lists" the existing orders, using links.

    This will return a JSON array of orders, in which each response is a valid link to the order. Here is a sample JSON output.

    ```
    {
      "orders": [
        {
          "href": "25c1667c-d56f-48b8-9f10-2c5fa3fd159f"
        },
        {
          "href": "3809bead-fd28-4cb0-bde5-dd13949585e3"
        },
        {
          "href": "05468171-40fe-4539-af78-30697b3b8de2"
        },
        {
          "href": "92732953-e3e6-417b-b53e-d32866b510d5"
        },
        {
          "href": "5def9450-8618-4551-b9a7-217475a6bb17"
        },
        {
          "href": "009671f8-70c1-4e40-b48b-87e45d649783"
        }
      ]
    }
    ```

    Uncomment Level3Test.java and work to make the final tests pass.

20. *Running the service as a WAR file*
    So far we have tested our REST API only within the Eclipse framework. Deploy the service in Tomcat, by locating the built WAR file in the build tree and copying to ~/servers/tomcat/webapps

    Note that there will be a warning:

    ```
    INFO:
    validateJarFile(/home/oxsoa/servers/tomcat/webapps/POResource/W
    EB-INF/lib/javax.servlet-api-3.1.0.jar) - jar not loaded. See
    Servlet Spec 3.0, section 10.7.2. Offending class:
    javax/servlet/Servlet.class
    ```

    You can safely ignore this!

The URL prefix of this service will be different to the previous deployment, because Tomcat names webapps individually, so the URL will be:
http://localhost:8080/POResource/purchase

You can now use the browser, curl or ARC to interact with this service.

21. *Extension 1:*
    There is a serious problem with our GET all orders logic. What is it? How can we improve it?

22. *Extension 2:*
    If you have completed all the above fully, you might wish to implement an improved flow.

    The proposed flow is that you allow an empty POST, which returns a location, followed by a PUT containing the order details, which "completes" the order. You can see that I have included some logic for this in the backend classes.

    Why is this better than just a single POST?