Exercise 4

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
- Jersey and Spring Boot
- Visual Studio Code
- curl
- Google Chrome/Chromium plus Chrome Advanced REST extension

Overview
The service we developed in Exercise 6 was frankly 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.

You will need to write Java code that adds RESTful JAXRS annotations to create a service.

There are a set of increasingly more demanding test cases that take us through Richardson's REST maturity model. Your task is to make those test cases pass.

Introduction

1. I have created a project (like in Exercise 3). The project has all the Spring Boot boilerplate, and a pre-configured gradle build.

2. It also includes a "backend" that implements a model and some simple database code so that you can focus on writing the JAX-RS resources and annotations.

3. The idea of this service is that we are going to allow users to post, put, get and delete PurchaseOrders from a system. The interface we want to design is a RESTful interface using JSON payloads. You will create orders by *posting* and/or *putting* JSONs. This will create a unique UUID and new hyperlink based on that UUID. You can then *get* or *delete* that resource using that hyperlink.

4. I have chosen to implement serialization into and out of JSON as part of the backend. Jersey, JAXRS and Java have many ways of automating the serialization into and out of JSON, XML and other formats. (I recommend you look up Moxy and Jackson as two approaches). However, I felt that obfuscated what is happening.

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

6. There is a class OrderRedis which implements 5 main methods (in pseudo-code)
   a. orderid = createOrder(JSON)
   b. updated = updateOrder(orderid, JSON)
   c. JSON_array = getOrders()
   d. JSON = getOrder(orderid) throws NotFoundException
   e. deleted = deleteOrder(orderid)

7. I have started you off with a "skeleton" resource class, and a comprehensive set of tests. Your job is to evolve the RESTful resources until the tests completely pass.

   Basically, I want you to be able to focus on getting the JAX-RS annotations right, not worry about the backend too much.

8. The project requires a simple Name/Value database called Redis to be running. Redis is a high-performance in-memory datastore which also supports different levels of on-disk persistence. It is a highly featured solution that supports clustering, replication and many other useful capabilities.

9. I have chosen a simple way of storing the data in redis which is to create three keys for each entry:
   `e078c9ce-c7f4-4a23-9bd3-04e60b3d8a95:json -> { the json }`

```
e078c9ce-c7f4-4a23-9bd3-04e60b3d8a95:complete -> true/false
e078c9ce-c7f4-4a23-9bd3-04e60b3d8a95:deleted -> true/false
```

10. Let's check redis is running before we try to test anything. On the command line type:
```
redis-cli
```

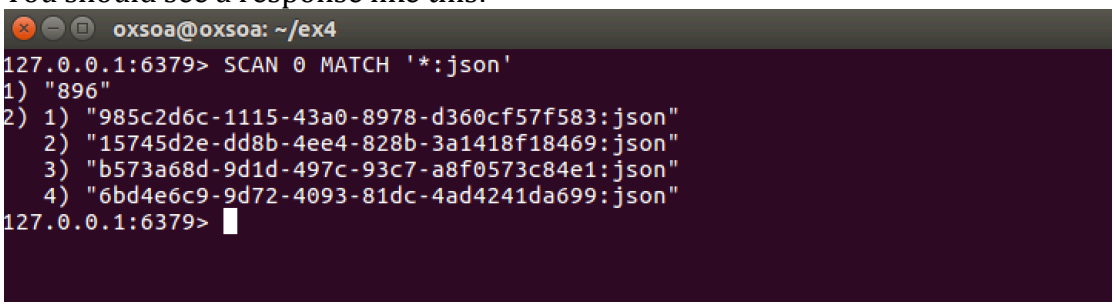
    If redis is running you will see:
```
127.0.0.1:6379>
```

    If redis is *not* running you will see:
```
Could not connect to Redis at 127.0.0.1:6379: Connection
refused
not connected>
```

    To start redis type:
```
sudo service redis-server start
```

11. In the Redis CLI, type:
```
SCAN 0 MATCH '*:json'
```

    You should see a response like this:



Steps

1. Download the existing project code:
```
curl -L https://freo.me/soa-ex4-code -o ~/Downloads/backend.zip
cd ~
unzip Downloads/backend.zip
mv PurchaseSpringBoot-master/ ex4
```

2. You should now have a project in the ~/ex4 directory.
   You can open the project view in vscode by typing:
```
code .
```

3. Try building the project (gradle build). It should compile but you also will see test failures.

4. This table captures the whole service at a glance

POST	/	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

5. Look at:

   src/test/java/org/freo/purchase/PurchaseApplicationTests.java

   This defines the set of tests that you need the service to pass. There are 6 main tests and they follow the "Richardson Maturity Model".

   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

```
Run Test | Debug Test
@Test public void testPOST_Level1() {···
}

Run Test | Debug Test
@Test public void testPOST_Level2() {···
}

Run Test | Debug Test
@Test public void TestGets_Level2() {···
}

Run Test | Debug Test
@Test public void TestPut_Level2() {···
}

@Test
Run Test | Debug Test
public void TestDelete_Level2() {···
}

@Test
Run Test | Debug Test
public void testGET_Level3() {···
}
```

6. I would recommend *you comment all the tests out apart from the first one*, and then uncomment them one-by-one, addressing each one in turn. *Hint: Ctrl-/ "toggles" comment and uncomment for a selection in vscode.*

7. Another thing you may wish to do is to change the test to go via MITMDUMP so you can see the requests and responses.

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

9. *Level 1*

   Let's first support creating an 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

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

}
```

   this order. The return code should be 201 Created, and the POST body needs to contain the server's representation of the resource.

   Hint #1:
   The following code demonstrates how to create and return a location in JAX RS neatly.
   Hint #2:
   You will need to handle the error case as well.

```
// need annotations here
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();

}
```

10. 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)

11. We now need to enable the correct tests for this level. You are aiming to support the following logic:

12. You will now need to access the incoming {id} in the URL Path.

13. 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.

Level 3 *Hypermedia (and Get all orders)*

14. 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"
    }
  ]
}
```

    Now get the final tests to pass.

15. *Running the service as a JAR file*
    You can run your service using

    ```
    java –jar build/libs/purchase–0.0.2.jar
    ```

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

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

17. *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?