Exercise 7a

Making our service into a single JAR microservice with a Redis backend

Prior Knowledge
Exercise 6b

Objectives
Looking at deployment models for Services
NoSQL backends

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
- Redis

Overview
We have built a reasonable RESTful service, but which can be exported as a WAR and run. It has no real backend as it is based on an in-memory singleton. However, we would like to create a simpler deployment model based on a single JAR, and we would like a reasonable backend database to house the results.

Steps

1. You can do this to your existing POResource project. However, because I'd like us to add *redis* support, I propose that we start from my completed version of Exercise 6.
   You can checkout this version by doing the following command-line magic.

   mkdir ~/ex7
   cd ~/ex7
   git clone https://github.com/pzfreo/POResourceMS.git
   cd POResourceMS

2. We would like to change the build to support creating a single JAR file. We also need to create a new class that supports this.

3. First, lets add some new parts to the build.gradle file.
   Hint: You could load the project into Eclipse, but I propose that we get the gradle build improved first, so I would suggest using a Linux editor like Atom, gedit or nano. This means that when we load the project into Eclipse, it will be aware of the new plugins.

   In the `plugins {}` section, add the following additional line:

   ```
   plugins {
     id 'com.github.johnrengelman.shadow' version '1.2.3'
   }
   ```

   This is a plugin that mirrors the shadow plugin for maven. This packages all the code required including all dependencies into a single JAR file. The result is a JAR file that has no external dependencies.

4. Having defined the plugin (using the gradle plugin extension mechanism) we now need to use it:
   Add the line to the main section of the gradle build (under the other similar lines)

   ```
   apply plugin: 'com.github.johnrengelman.shadow'
   ```

5. We also need the *application* plugin, which works with shadow to build self-contained executable JARs (see https://en.wikipedia.org/wiki/JAR_(file_format)#Executable_JAR_files)

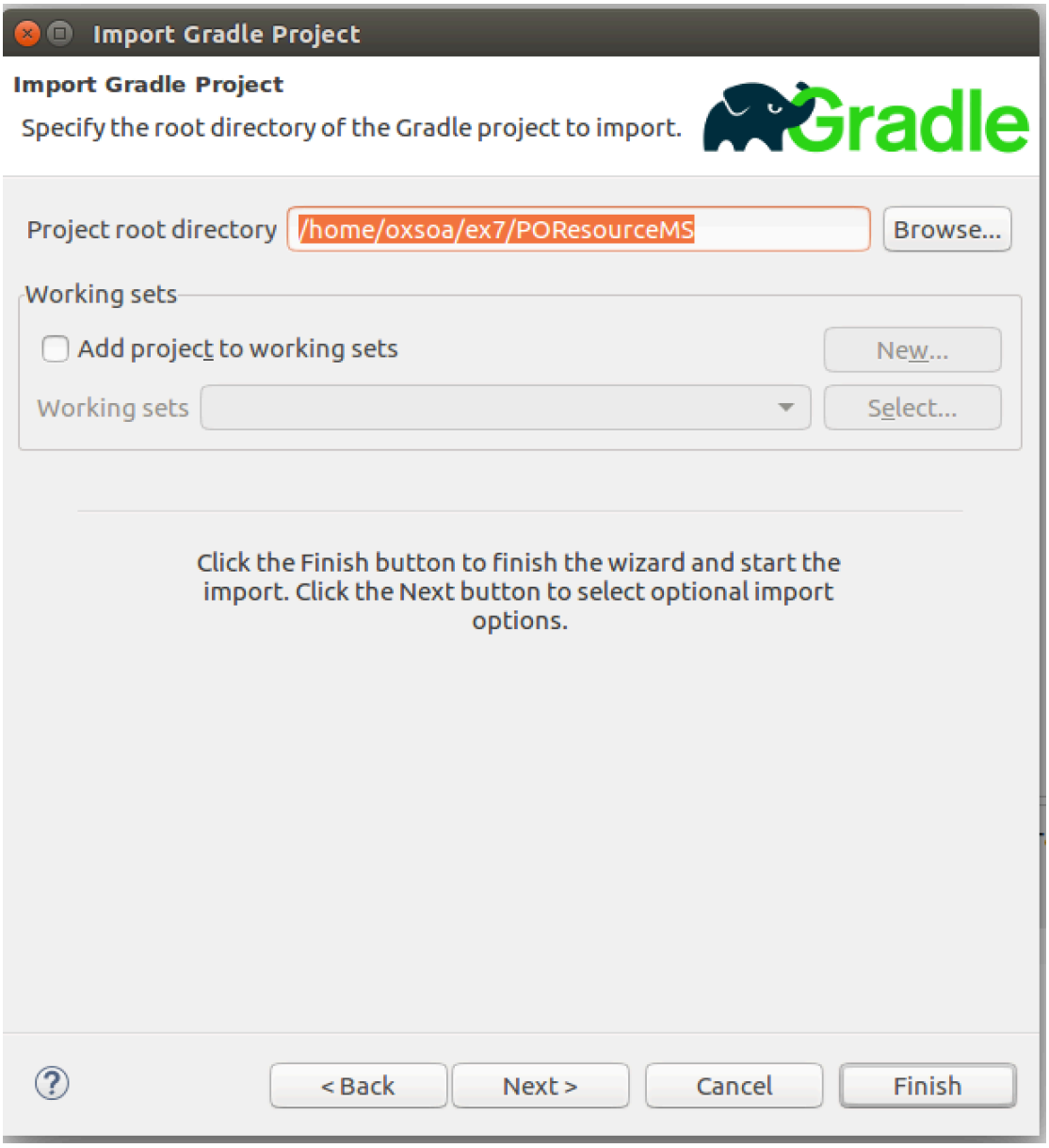
   Add the line:
   ```
   apply plugin: 'application'
   ```

6. We need to tell the application plugin the name of our "Main" executable class:

   Add the following line underneath the "apply plugin" lines
   ```
   mainClassName = 'freo.me.rest.Main'
   ```

7. Save your changes to the file.

8. Now import the project into Eclipse:
   File->Import Gradle->Gradle Project
   Next>
   Browse to the Project root directory: /home/oxsoa/ex7/POResourceMS
   Finish



9. Now go to the class called **Main** in the package freo.me.rest. **Uncomment** it.

   Here is a code listing:

```
package freo.me.rest;

import java.net.URI;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.UriBuilder;

import org.eclipse.jetty.server.Server;
import org.glassfish.jersey.jetty.JettyHttpContainerFactory;
import org.glassfish.jersey.server.ResourceConfig;

@ApplicationPath("/")
public class Main extends ResourceConfig {


public Main() {
        // this is the package to scan for Resources
        packages("freo.me.rest");
}

public static void main(String[] args) throws Exception {
        URI baseUri = UriBuilder.fromUri("http://localhost/").port(8080)
                        .build();
        // This is fairly self-explanatory.
        // You can define the URL on which the server will listen.

        ResourceConfig config = new Main();
        // 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();
                // Obvious!
        }

}

}
```

10. Now you can build this into a shadowJar. You can either use the gradle plugin, or the command line:

```
cd ~/ex7/POResourceMS
gradle clean shadowJar
```

11. This creates a file:
    `build/libs/POResourceMS-all.jar`

12. Try it out by executing:
    `java –jar build/libs/POResourceMS-all.jar`

13. Test it. The URL is http://localhost:8080/purchase

14. Extension: Check out the other build targets:
    `gradle runShadow`

```
gradle distShadowZip
gradle installShadowApp
```
Hint: execute these with –info to see more of what is happening

15. *Adding Redis support*

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.

We are going to use it as a sample backend for our Microservice, replacing the existing singleton Java object.

If you have checked out my version of the service, you will already have a new backend class OrderRedis.java. If not, you can find a copy here: http://freo.me/OrderRedis

Review the code.
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
```

This is probably not the best implementation, but it meant very few changes from the in-memory implementation. Note also that I did not implement paging in the getAllOrders and unlike the singleton version, this will not return all orders but a paged set. This would need fixing in a production system.

You can simply replace the singleton usage of OrderInMemory with an instance of OrderRedis. It is thread safe so you can simply instantiate a single object and call it from the POResource service with impunity.

In other words, you can change **FROM**:

```
OrderInMemory orderSingleton = null;

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

TO:

```
OrderRedis orderSingleton = new OrderRedis();
```

*If you really care then you can also refactor the name **orderSingleton** throughout your code since its no longer a singleton! But I'm not sure its worth it.*

16. Make sure the jedis client is part of the gradle build.

    If you need to uncomment it or add it then select the Project and Right-click **Gradle->Refresh Gradle Project** to let Eclipse know about the new dependency.

17. 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
    ```

18. You can do a gradle build to re-run the tests with the redis backend.

19. In a terminal window start the redis-cli and type
    SCAN 0 MATCH '*:json'

    You should see a response like this:

    ```
    127.0.0.1:6379> scan 0 MATCH '*:json'
    1) "288"
    2) 1) "528551f0-810b-4c74-ab89-0d20bde584c1:json"
       2) "e685fa1f-cdd4-4ace-87e4-6421c67f54fb:json"
       3) "e078c9ce-c7f4-4a23-9bd3-04e60b3d8a95:json"
       4) "a235c1f9-da10-4be8-b9c3-29a6ef1d7894:json"
       5) "43d90e04-2c1c-40d5-9ab1-ac254ed310a2:json"
    ```

20. Build into a shadow JAR and test using java –jar

21. Congratulations the lab is over!