# Exercise 7b

*Documenting your service with Swagger*

**Prior Knowledge**
Basic understanding HTTP verbs, REST architecture
Some Java coding skill

**Objectives**
Understanding Swagger and how to embed support into JAX-RS

**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
- Swagger UI

**Overview**
Swagger - also now known as the Open API Initiative specification - is a simple JSON model for describing RESTful services.

We are going to use some Java tools to create a Swagger description of our API and then use the Swagger tooling to view this.

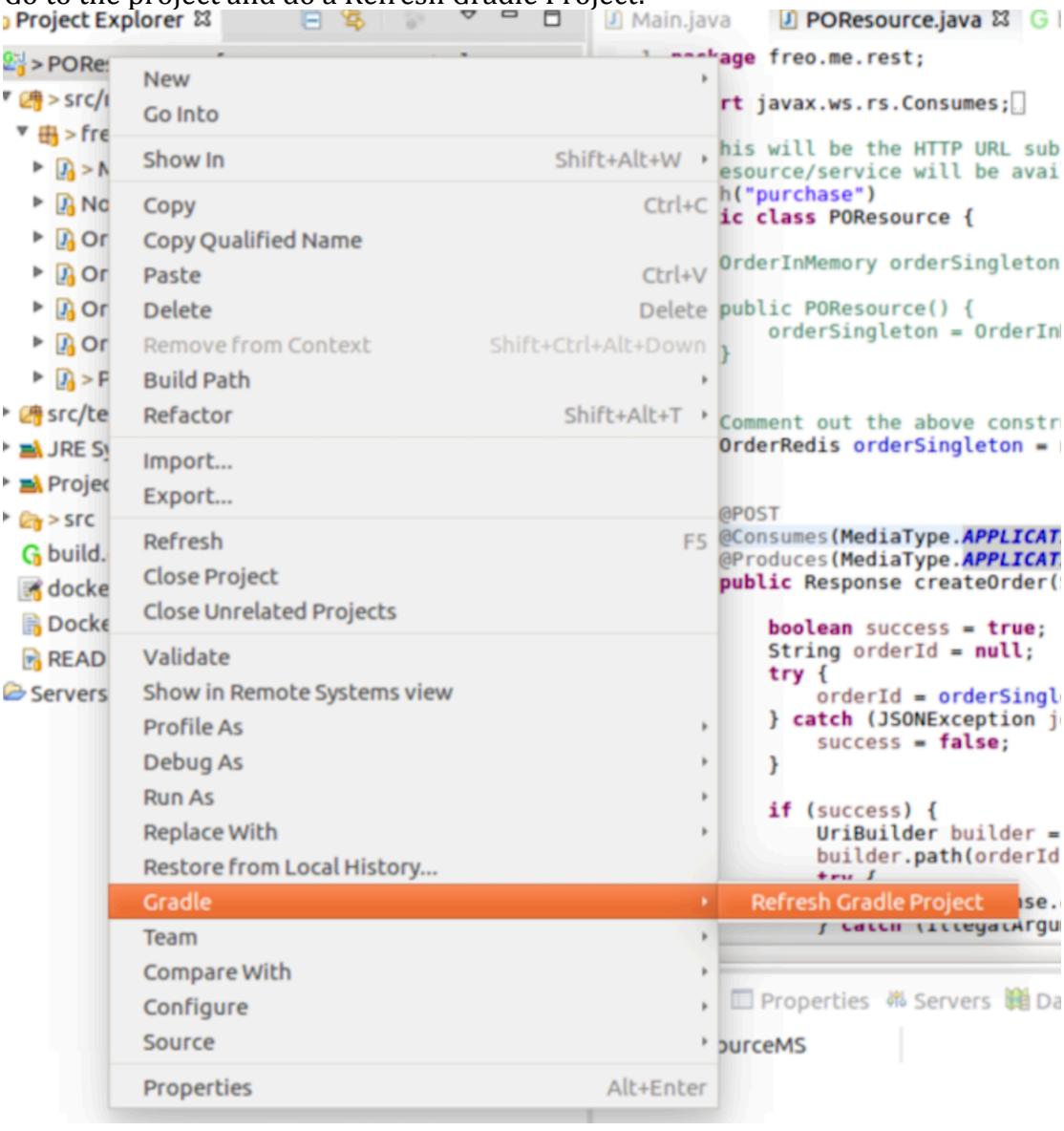
Steps

1. Start with your POResourceMS directory from Exercise 7.

2. Edit the build.gradle to add the following new dependency:

```
compile 'io.swagger:swagger-jersey2-jaxrs:1.5.9'
```

3. Go to the project and do a Refresh Gradle Project:



4. Go to the POResource.java and add a second annotation to the class just above @Path:
   `@Api("purchase")`

   Eclipse should automatically add the swagger package import, but you may need to add:
   `import io.swagger.annotations.Api;`

5. We now need to tell the JAX-RS system about Swagger and get Swagger's code running in our microservice.

   *(Hint: if you are using a WAR file model instead, there is a completely different way of doing this, which is documented in the Swagger docs).*

6. Edit the Main.java

   We need the constructor to look like this:

```
public Main() {
      packages("freo.me.rest");

      register(ApiListingResourceJSON.class);
      register(SwaggerSerializers.class);
      register(CORSFilter.class);

      BeanConfig beanConfig = new BeanConfig();
      beanConfig.setVersion("1.0.0");
      beanConfig.setSchemes(new String[]{"http"});
      beanConfig.setHost("localhost:8080");
      beanConfig.setBasePath("/");
      beanConfig.setResourcePackage("freo.me.rest");
      beanConfig.setScan(true);
}
```

   When you add this, the classes should all get resolved and the right imports added, with the exception of CORSFilter.class, which we will create in a minute.

   Basically this is telling JAX-RS about some extra handlers that will be registered to add Swagger description to our code. The code will be automatically scanned and the Swagger description created from our existing JAX-RS annotations, but we need to give a little extra information (via the BeanConfig) to the system.

7. We also want the Swagger system to be able to generate a nice tool from our Swagger output. There are two ways to do this. We could embed the whole SwaggerUI into our app but this is complex and would make the lab overly confusing. Instead, we are going to run the Swagger UI and point it

to our Swagger definition. This looks like a cross-site scripting attack, and so we need to use the CORS spec to avoid this problem.

8. To allow the SwaggerUI to read our JSON, we need to add some headers into the responses given by Jetty. We can do this with a JAX-RS filter.

9. Create a new Java class `freo.me.rest.CORSFilter`
   Copy the following code into the class.

```
package freo.me.rest;

import java.io.IOException;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.core.MultivaluedMap;

public class CORSFilter
implements ContainerResponseFilter {

        public void filter(ContainerRequestContext requestContext,
                           ContainerResponseContext responseContext)
           throws IOException {

        MultivaluedMap<String, Object> headers = responseContext.getHeaders();

        headers.add("Access-Control-Allow-Origin", "*");
//      headers.add("Access-Control-Allow-Origin", "http://wherever.org");
//      allows CORS requests only coming from wherever.org

        headers.add("Access-Control-Allow-Methods",
            "GET, POST, DELETE, PUT");
         headers.add("Access-Control-Allow-Headers",
            "X-Requested-With, Content-Type");
         }

}
```

   This is available at: http://freo.me/ex7b-cors

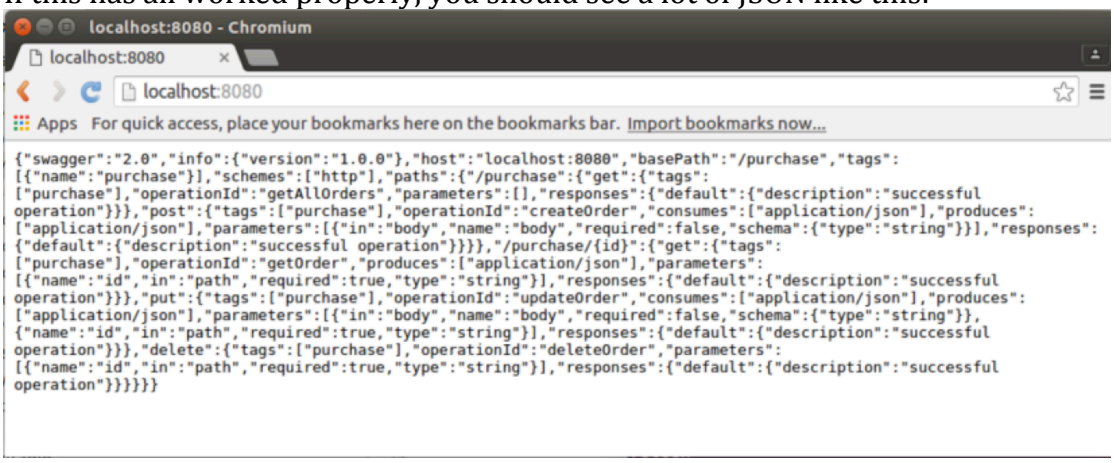
10. Rebuild the app using **gradle shadowJar**

11. Start the Jar using:
    `java -jar build/libs/POResourceMS-all.jar`

12. You should see an additional line of log compared to before:

```
oxsoa@oxsoa:~/ex7/POResourceMS$ java -jar build/libs/POResourceMS-all.jar
[main] INFO org.reflections.Reflections - Reflections took 795 ms to scan 1 urls
, producing 9729 keys and 18518 values
[main] INFO org.eclipse.jetty.server.Server - jetty-9.1.z-SNAPSHOT
[main] INFO org.eclipse.jetty.server.ServerConnector - Started ServerConnector@b
842275{HTTP/1.1}{0.0.0.0:8080}
```

13. Browse to http://localhost:8080
    If this has all worked properly, you should see a lot of JSON like this:

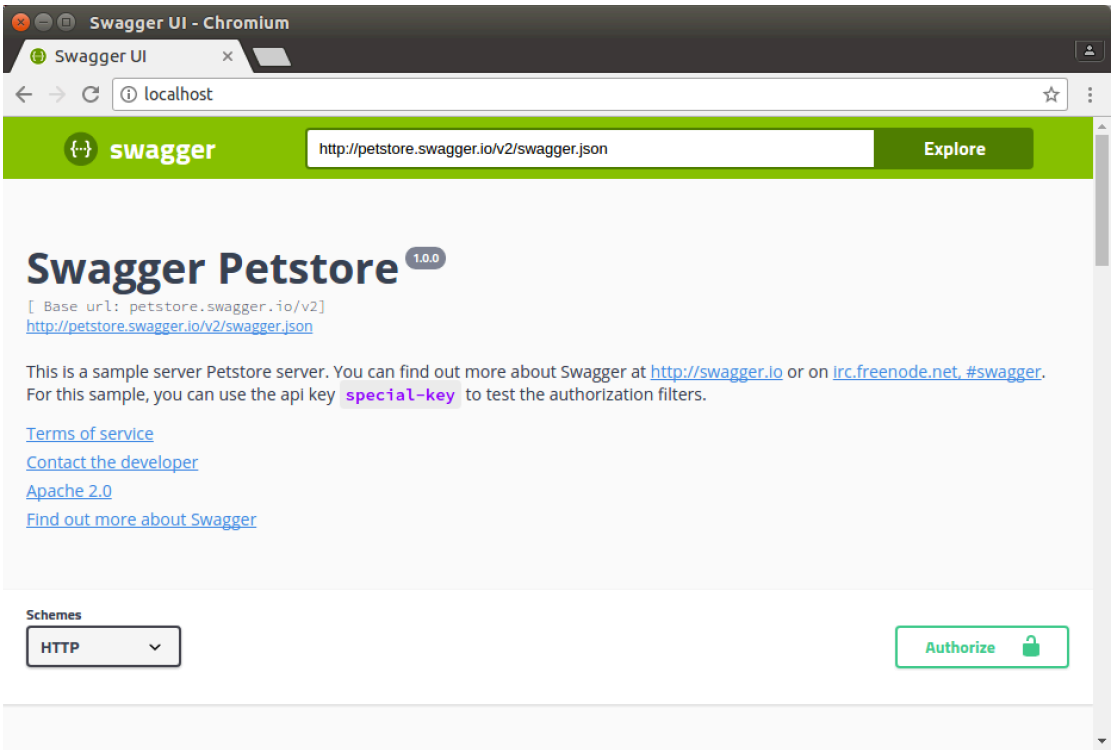



14. We will use Docker to run the Swagger UI (this will be explained in another exercise).

    In a new shell window, type:

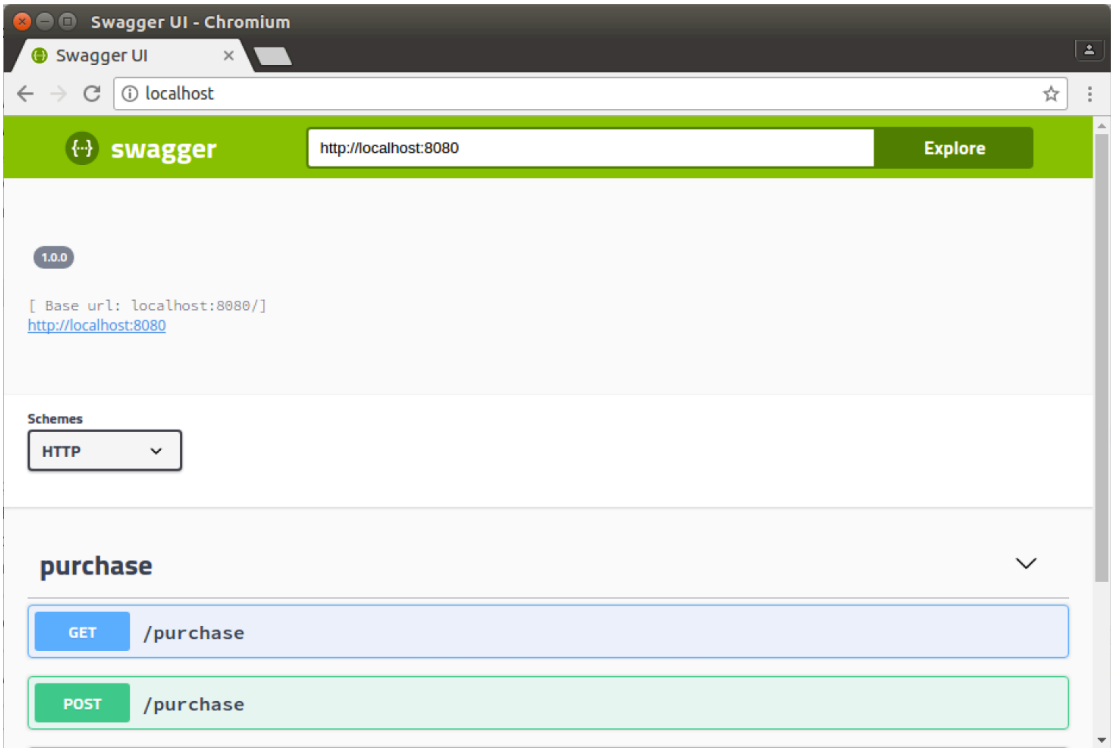
```
sudo docker pull swaggerapi/swagger-ui
sudo docker run -p 80:8080 swaggerapi/swagger-ui
```

15. This will start the Swagger UI running on port 80. Browse to:
    http://localhost
    You should see a nice UI like this:




16. In the URL box set the URL to be http://localhost:8080

17. You should see this:




18. Explore the API and try it out using the Swagger test tool.

19. That's all!