Exercise 5

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)

- Exercise 4 + requirements
- 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 **ex4** directory from Exercise 4.

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

```
implementation('io.swagger.core.v3:swagger-jaxrs2:2.0.10')
implementation('io.swagger.core.v3:swagger-jaxrs2-servlet-initializer:2.0.10')
```

3. If you do this in vscode, it should automatically refresh the dependencies and download them.

4. We now need to tell the Spring Boot about Swagger and get Swagger's code running in our microservice.

5. In PurchaseConfiguration, we need to add a resource to configure Swagger. Underneath the existing *register(..) line,* add a line with:

   register(OpenApiResource.class);

   If the gradle / vscode plugin has worked, vscode will help you add the import line.

```
import io.swagger.v3.jaxrs2.integration.resources.OpenApiResource;

@Configuration
@ApplicationPath("/")
public class PurchaseConfiguration extends ResourceConfig {
    public PurchaseConfiguration() {
    }

    @PostConstruct
    public void setUp() {
        register(Purchase.class);
        register(OpenApiResource.class);
    }
}
```

6. This is enough to enable some Swagger generation from your code. However, it would be nice to give the API definition a bit more information. To do this we need to add an annotation to Purchase.java:

   Annotate the Purchase class with the following:

```
@OpenAPIDefinition (info = @Info (
      title = "PurchaseAPI", version = "0.0.2"
))
```

   VSCode should add the correct imports for you, so your code should look like:

```
import io.swagger.v3.oas.annotations.OpenAPIDefinition;
import io.swagger.v3.oas.annotations.info.Info;

@Component
@Path("/purchase")
@OpenAPIDefinition (info = @Info (
    title = "PurchaseAPI", version = "0.0.2"
))

public class Purchase {

    OrderRedis backend = new OrderRedis();
```

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. In a later exercise we will also see another way to do this.

8. Running the Swagger UI in a separate process ends up looking like a cross-site scripting attack to the browser, and so we need to use the CORS spec to avoid this problem.

9. We can do this with a servlet filter. Create a new Java class `org.freo.purchase.CORSFilter.java`

Copy the following code into the file.
This is available at: https://freo.me/soa-cors

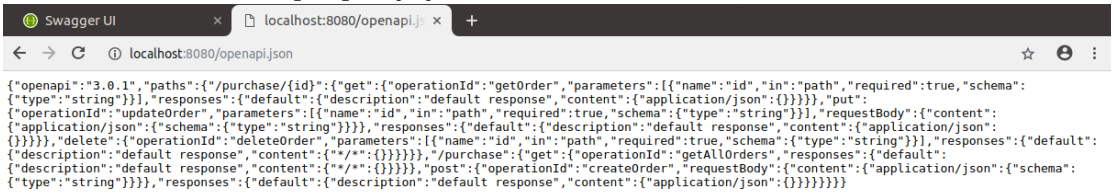
```
package org.freo.purchase;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Component;

@Component
public class CORSFilter implements Filter {

    @Override
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
    throws IOException, ServletException {
        HttpServletResponse response = (HttpServletResponse) res;
        HttpServletRequest request = (HttpServletRequest) req;
        response.setHeader("Access-Control-Allow-Origin",
request.getHeader("Origin"));
        response.setHeader("Access-Control-Allow-Methods", "POST, GET, OPTIONS, PUT,
DELETE");
        response.setHeader("Access-Control-Max-Age", "3600");
        response.setHeader("Access-Control-Allow-Credentials", "true");
        response.setHeader("Access-Control-Allow-Headers", "Foo, Bar, Baz");
        chain.doFilter(req, res);
    }

    @Override
    public void init(FilterConfig filterConfig) {}

    @Override
    public void destroy() {}
}
```

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

11. Start the Jar using:
    `java -jar build/libs/purchase-0.0.2.jar`

12. Browse to http://localhost:8080/openapi.json

    If this has all worked properly, you should see a lot of JSON like this:



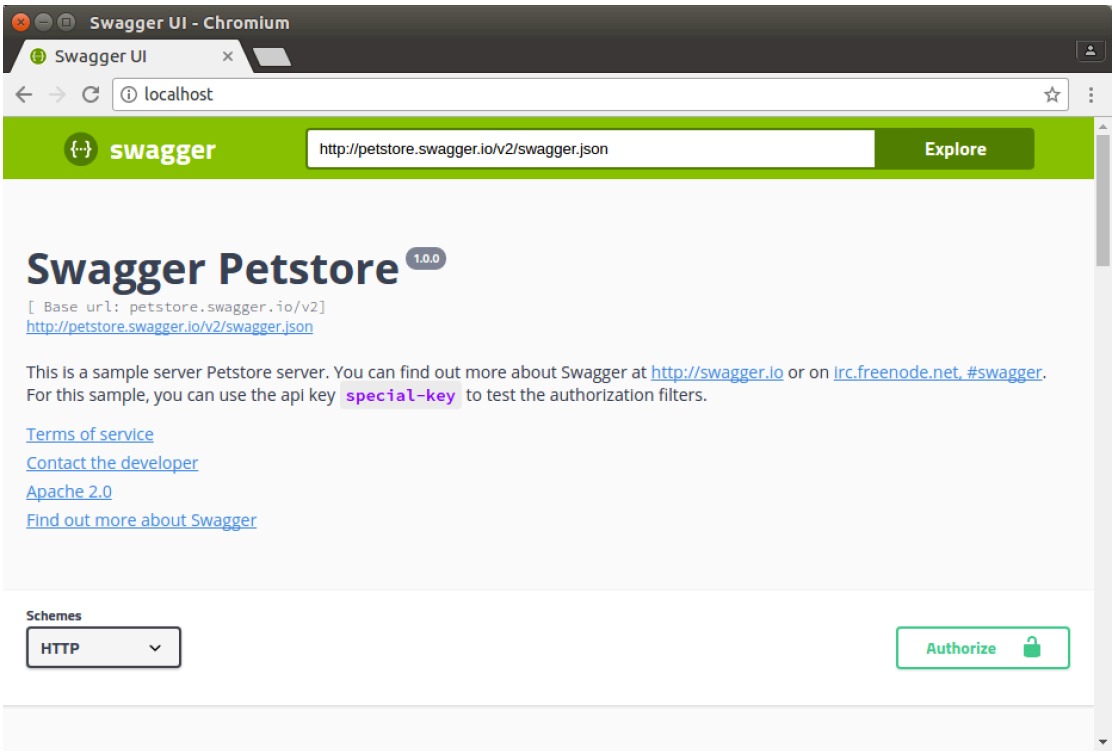
    You can also browse http://localhost:8080/openapi.yaml

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

In a new shell window, type:

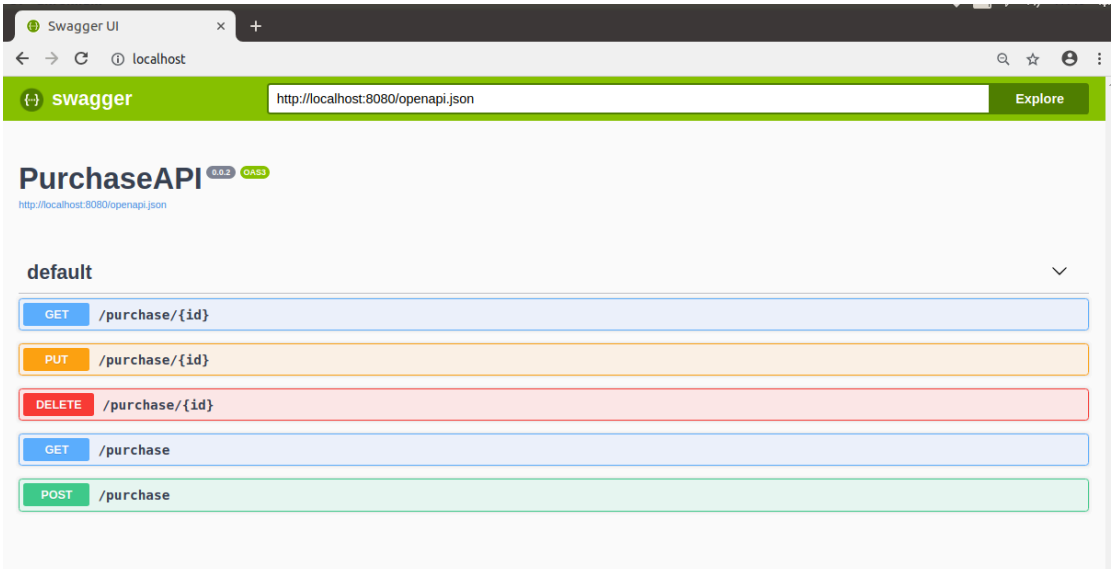
```
docker pull swaggerapi/swagger-ui
docker run -p 80:8080 swaggerapi/swagger-ui
```

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



14. In the URL box set the URL to be http://localhost:8080/openapi.json

15. You should see this:



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

17. That's all!

18. Extension:

    Sign up with Swagger Hub's free trial:

    https://swagger.io/tools/swaggerhub/

    Use the tool to design an API from scratch. You can use your own ideas, but if you want one from me, how about creating an API for a simple todo list tracker.