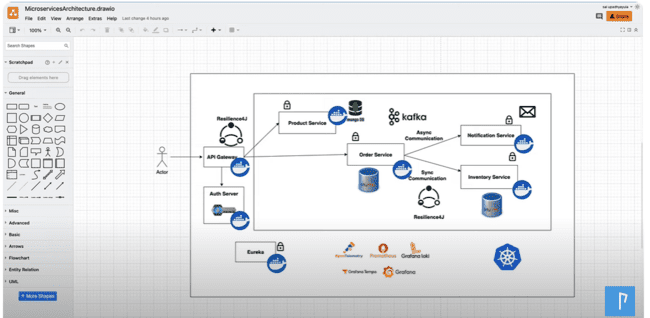
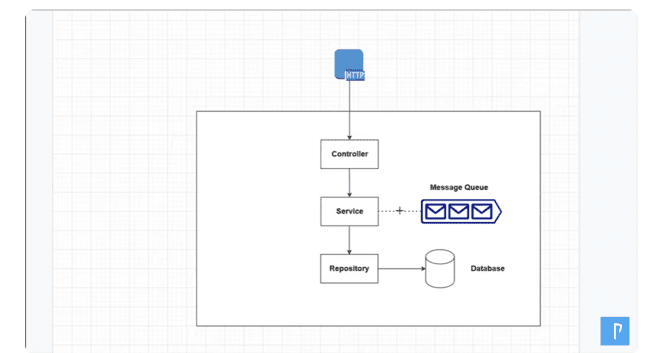
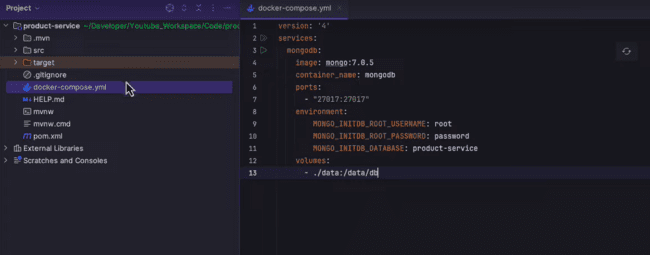
Microservices:



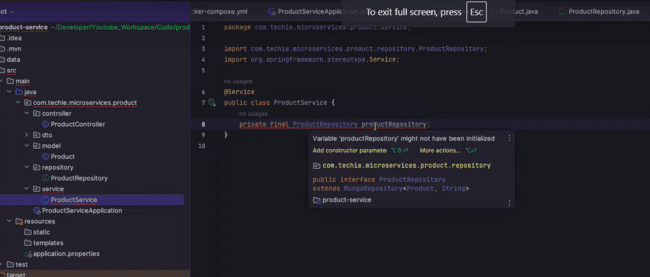
API gateway acts like a gatekeeper, which sends out the request from the user to the inner service, for which it wants to interact with.



As MongoDB is running inside a container, so we stop the container, all the data that will be inside the container will be lost, so we need to provide the volume folder where we need to store the data inside this part in our file system.



When compiler complains of adding the constructor for injecting the dependencies, we can inject that using the Lambok using the @RequiredArgsConstructor



In Java 21, we need to make a request class where we pass the full object:

example for class:

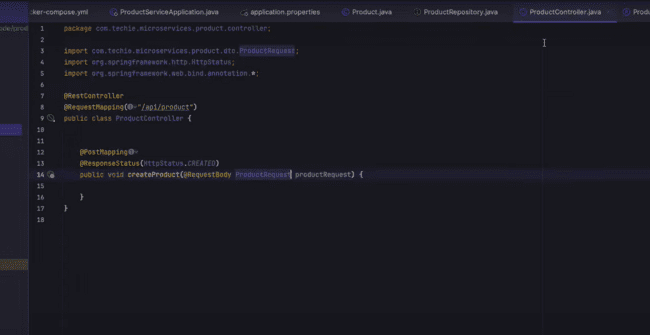
Product{id, name, description}

we need to use the ProductRequest class as follows:

public record ProductRequest(String id, String name, String description){}

And to access the same we need to write the

productRequest.name() instead of productRequest.getName();



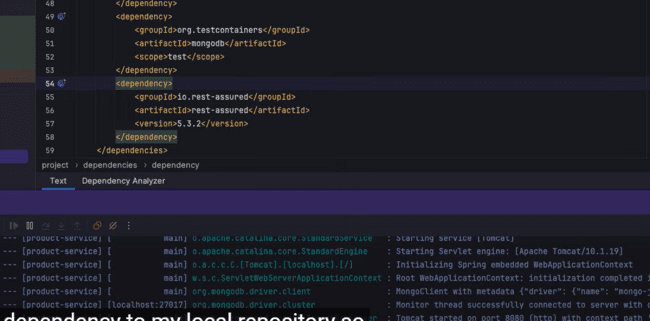
A screenshot of a computer

AI-generated content may be incorrect.

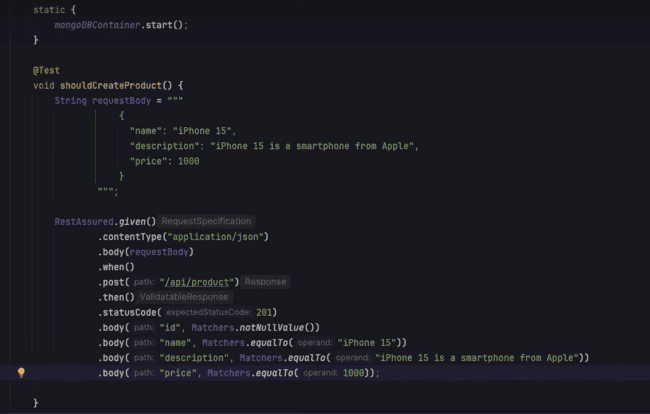
A screenshot of a computer screen

AI-generated content may be incorrect.

Io.rest-assured dependency to make use of the REST API calls in the TEST environment.



To check the MongoDB test:



A screenshot of a computer

AI-generated content may be incorrect.

HOW MONGODB interacts with spring boot using Docker:

We need to run Spring boot in one container and MongoDB in another container and then we need to check how they communicate between them using the **docker-compose file or we need to use Docker-Link**

The Docker Compose is a tool for running multi container docker application, we just need to make the configuration file, rest would be handled by the DOCKER itself.

A diagram of a blue whale with a blue container and green logo

AI-generated content may be incorrect.

We need to run both containers using the command prompt of docker using docker link command or we need to use docker-compose.yml to provide the configurations and can simply run the application.

The grep command is used to find out the word in the result:

A screenshot of a computer

AI-generated content may be incorrect.

***IMPORTANT: TEST CONTAINERS, we can use to test our application.***

***A screenshot of a computer

AI-generated content may be incorrect.***

Test Containers dependency

A screen shot of a computer program

AI-generated content may be incorrect.

If we want to provide any random port on the test part we can use the following annotation:

A computer screen shot of a program code

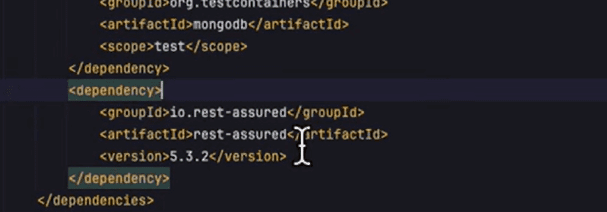
AI-generated content may be incorrect.

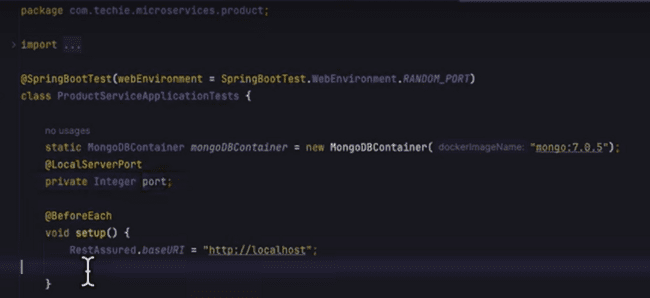
Connection to the mongo container:



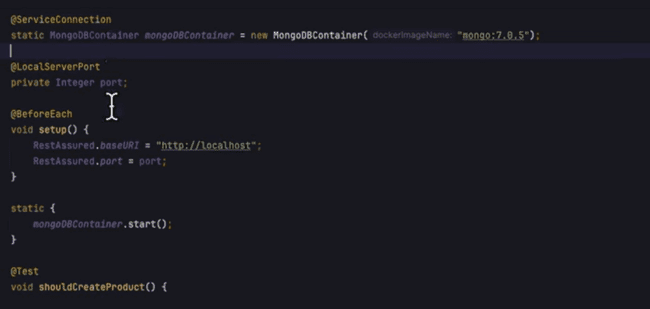
@LocalServerPort: This annotation is used to check the port on which the application is running right now.

Also RESTASSURED need to use one dependency in the pom.xml file as shown below:





@ServiceConnection: this annotation will dynamically assign the monoDB host and port while running the application as shown below:



We need to use the RestAssured for testing of the API:



Now for the second service, we need to use the following configuration of the mysql as shown in the below configuration:



SQL Docker:

A screenshot of a computer

AI-generated content may be incorrect.

SQL Configuration:

A screen shot of a computer

AI-generated content may be incorrect.

FOR THE ABOVE CONFIGURATION, if the table does not exist, we need to create using either mysql or using the docker file as following:

Creating a folder structure as: docker -> mysql -> init.sql

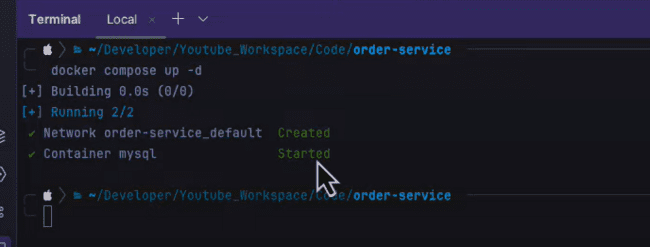
A screenshot of a computer

AI-generated content may be incorrect.

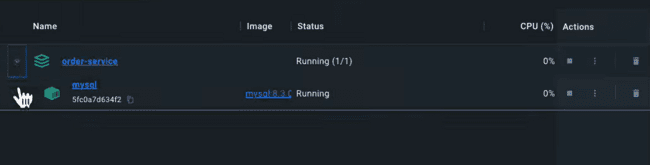
Docker-entrypoint-initdb.d command would be executed whenever docker mysql container would be starting up:A screenshot of a computer program

AI-generated content may be incorrect.

After adding this command, we docker compose command and we can verify that mysql database would be running:



This we can verify in docker that instance is running or not as shown below:



But one point to mention here that, the volume is not executed as per the logs shown below:

A screenshot of a computer

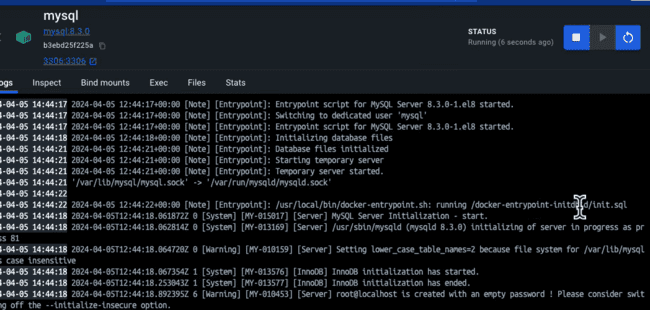
AI-generated content may be incorrect.

As the “mysql” folder is already created and we need to delete that part.

A screen shot of a computer

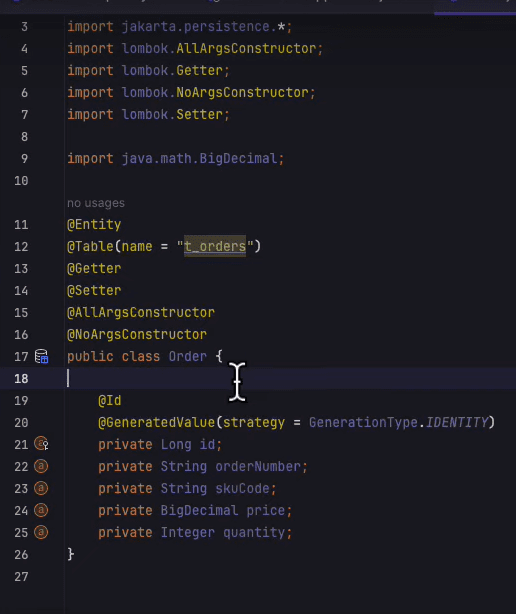
AI-generated content may be incorrect.

And need to down the docker and again run the docker using docker compose and if you open the mysql logs in the docker again, there would logs of running the entry point as show in the below screenshot:

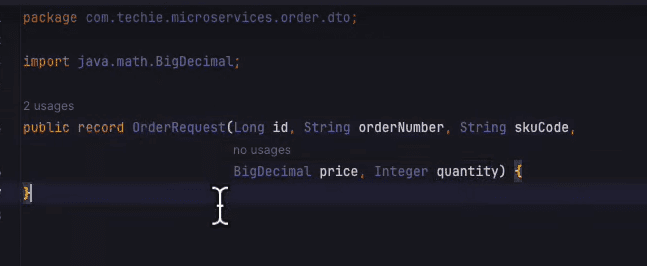


After this if you run the SQL workbench, you would be seeing that order\_service would be created.

And we need to create the model class for the Order, as shown below:



OrderRequest record class:



Spring.jpa.hiberate.ddl.auto=none, we can use this as none but cannot use this as create in production.

So, we use the FLYWAY MIGRATION, which allows us to version control the database.

So, if we want to add a new field in database, so we add that in java and simply commit that code, but to roll back that is a tough option.

So, with Flyway, we have the option to roll back to the previous versions and, write own SQL queries to update the database.

The scripts are under db.migration folder and dependency is mentioned in pom.xml.

We need to

A screen shot of a computer

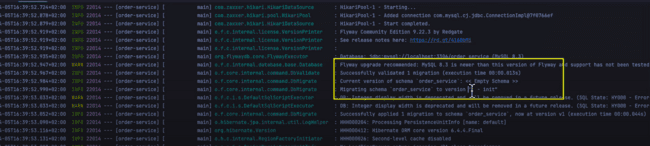
AI-generated content may be incorrect.

So, we need to run the SQL using the script placed in the ***V1\_\_init.sql*** and script is as follows, which allows us to create a new table in the MySQL:

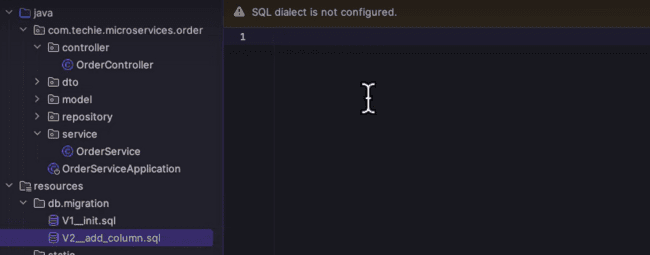
A screenshot of a computer

AI-generated content may be incorrect.

If we run the application, then it is clear from the logs that migration script has been executed.



And now if we need to update some column in the database, we need to make the following file and then we need to write our logic in that file:



Now if restart the application, then we’ll be able to see that, it had skipped the migration of script one, as it is already there which has been shown in the logs as well:

A computer screen with white text

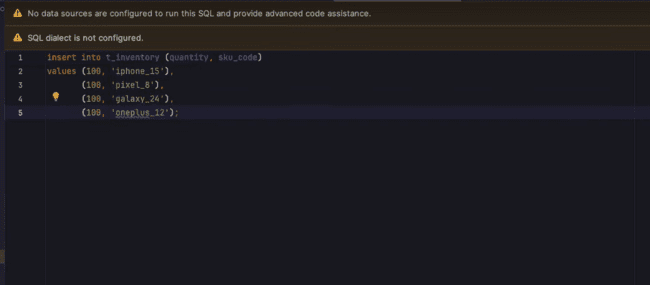
AI-generated content may be incorrect.

So, when the tables are getting created, there is one more table that is getting created for the flyway\_schema\_history, which shows which script has been created for the flyway\_schema:

A screenshot of a computer

AI-generated content may be incorrect.

For the inventory service, we are inserting the value using the second script which is V2\_\_add\_inventery.sql as shown below:



A screen shot of a computer

AI-generated content may be incorrect.

***If we want to inject the constructor dependency in the class then we need to use the @RequiredArgsConstructor otherwise we can use the @Autowired in that case***

***We need to use the IsgreaterThanEqual instead of IsGreaterThanEquals in the database repository.***

**SYNCHRONOUS COMMUNICATION BETWEEN THE ORDER AND INVENTORY SERVICE**

FEIGN CLIENT ALSO WORKS AS A LOAD BALANCER……..?

With the help of SPRING CLOUD OPENFEIGN, we need to use the inter process communication or the Synchronous communication, between the services:

A diagram of a process

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

Also, we need to include the spring cloud dependency and its version:



A screen shot of a computer

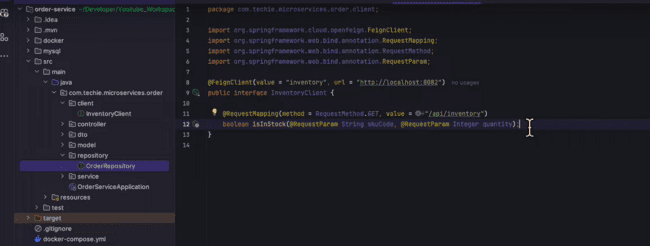
AI-generated content may be incorrect.

@EnableFeignClients annotation need to be use at the top of the spring project:

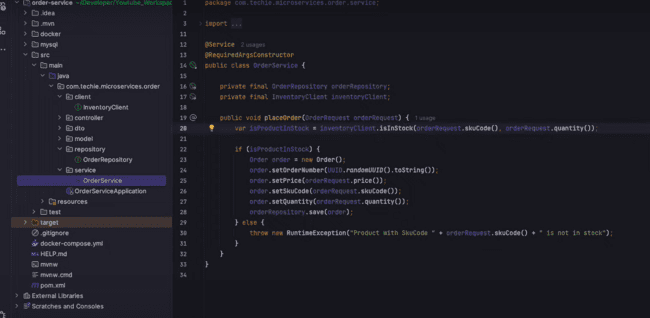
A computer screen with text

AI-generated content may be incorrect.

We need to create the inventory client interface in the java class using the annotation as @FeignClient and we need to provide the URL of the service where inventory service is running, and client would look like this:

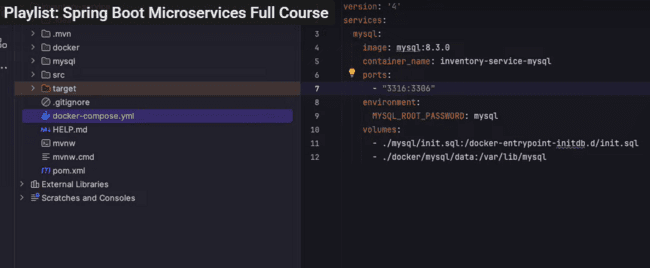


Now we can call the client in the Order service to check if the product is available or not, and if that is available, then we need to create a new order, else throw an exception as shown below:

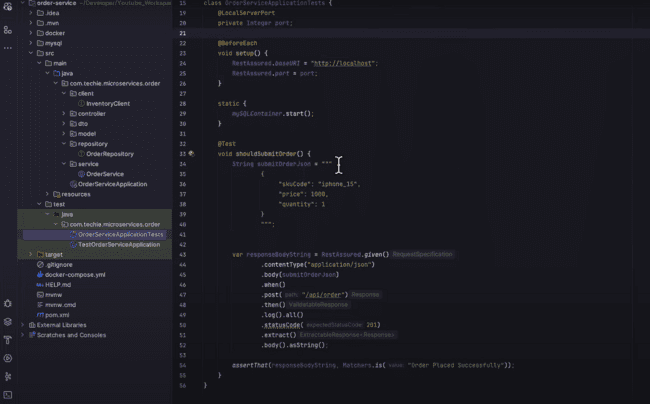


We are using the ***var*** in the java to check for the response of the inventory client.

We need to use the new name of the container\_name as other and also change the port for the inventory service as shown below:



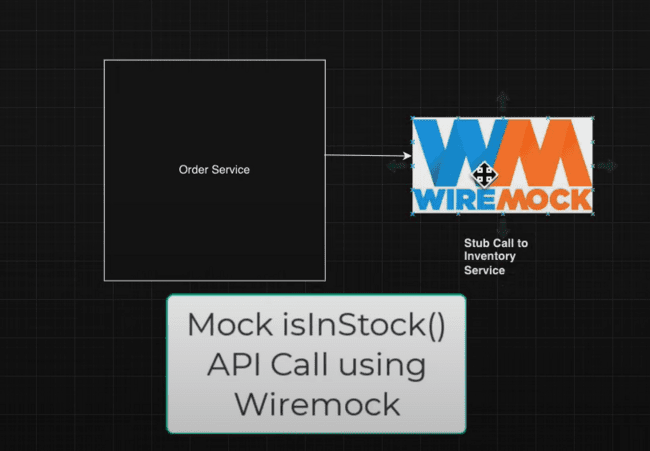
Integration test in the order service as shown below:



Now to check the integration test, we need to call the inventory client, but it would call the API again and again, which would incur additional cost, as there might be charge for each API call, so we need to mock the same to test this. And also, if the service is down, we cannot check this.

Using mocking, we are just actually mocking the method, but we are not actually testing the HTTP call.

But with the wiremock, we can mock out the API, as well we can actually test the HTTP interaction.



Order service will call the wiremock stub and it stub the call to the inventory service.

There is one dependency needed for the wiremock which needs to be injected in the pom.xml file.

A screen shot of a computer

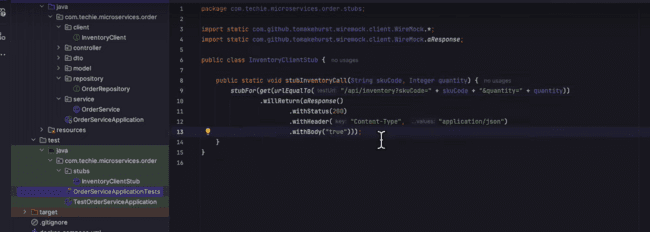
AI-generated content may be incorrect.

We need to configure the wiremock using the @AutoConfigurationWireMock(port=0) at the top which would enables the wiremock server, and port =0 means the server can use any random port for that.

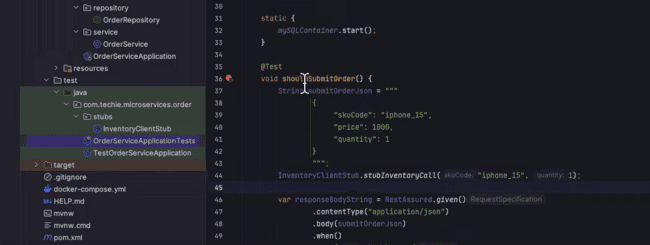
A screenshot of a computer program

AI-generated content may be incorrect.

So, we need to create a new package for the stub method which needs to be called inside the test service.



We would be calling this inside the test method as:



But connection refused error is there as we have configured the hardcoded port:

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

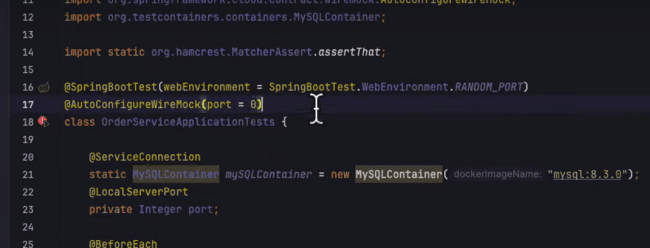
We can give the value of that port in the method using the above.

A screen shot of a computer program

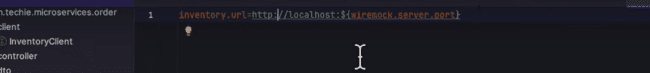
AI-generated content may be incorrect.

In the similar way, we can use the application.properties in the test package and us the same port as that has been used by the wiremock server.

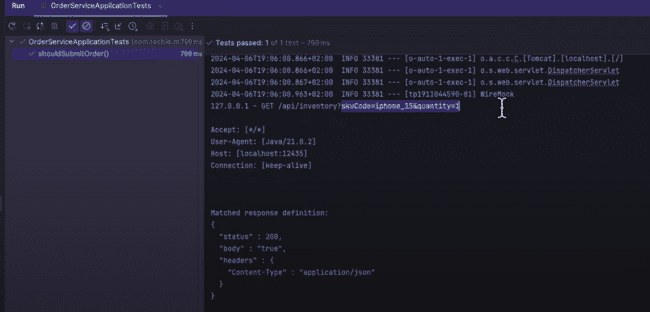
Here we have given the random port, so same way we need to use in the application.properties file.



${wiremock.server.port} will inject the port same used by the wiremock server used in the test package.



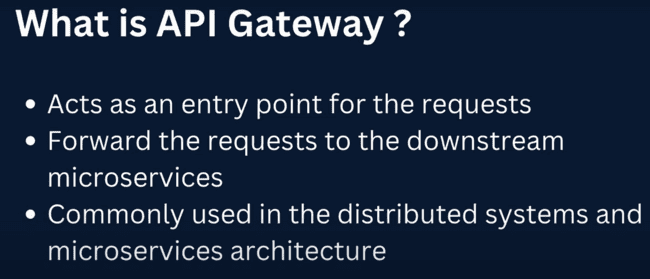
Response by the wiremock server:



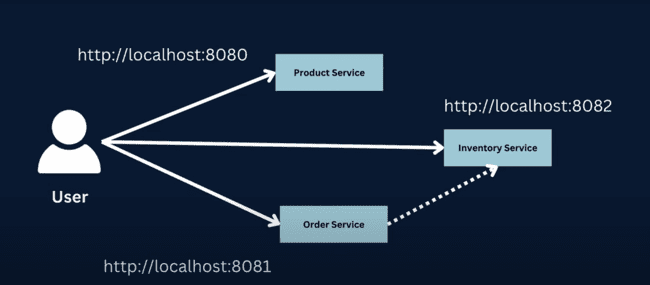
A screenshot of a computer

AI-generated content may be incorrect.

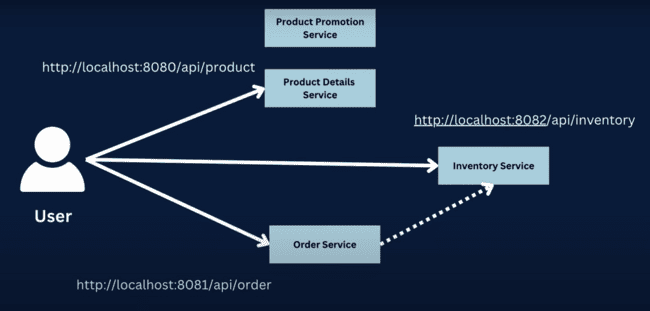
**API Gateway using Spring cloud gateway MVC**



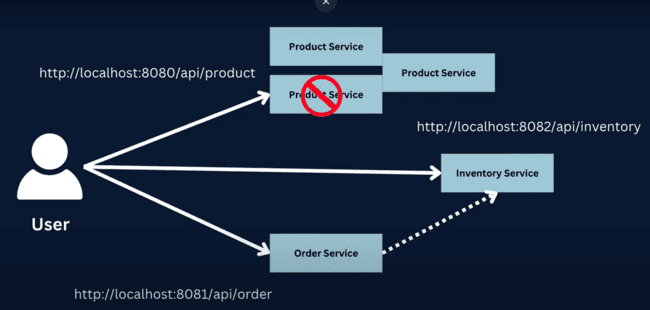
Now our services are hosted at the different URL’s like the following:



And if we create a new microservice as “Product Promotion Service”, then URL needs to adjust for that as well:



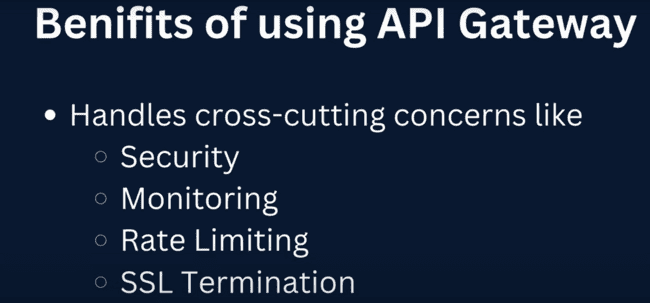
Also, there would be case if there are multiple instances of same services running for the one microservice, then if the one service fails, then client won’t able to access the other one, which is connected through this microservice as shown below:



Due to above and other reasons, we need to use the API gateway, which acts like an entry point, which redirects to other microservices.

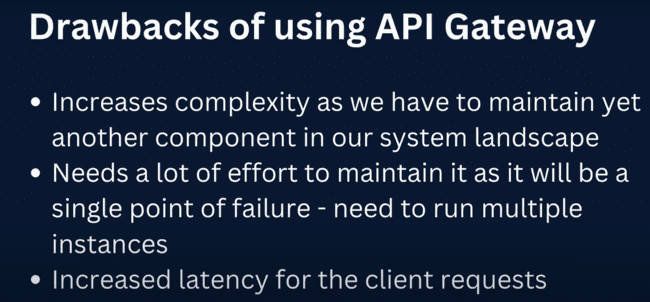
And, in the future, if there is change in the URL of the microservice, then we just need to do the update in the API gateway, and client don’t even know that there is any change in any microservice.

API gateway handles the following cross cutting concerns:

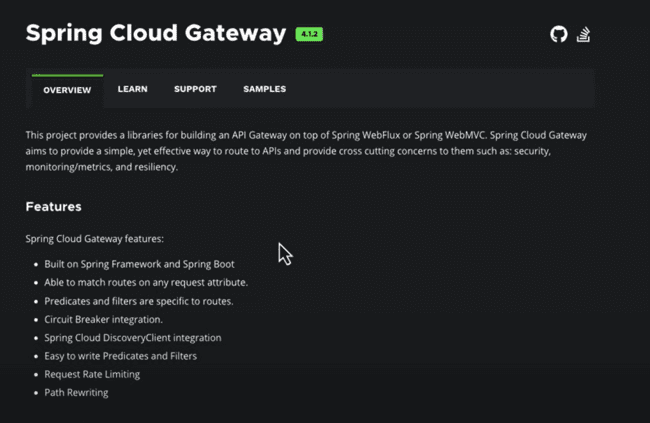


Following are also the drawbacks of the API gateway:

As the client calls the API gateway first and then that calls the internal microservice, then there are certain chances of latency, as it needs to redirect the call to the internal microservice.



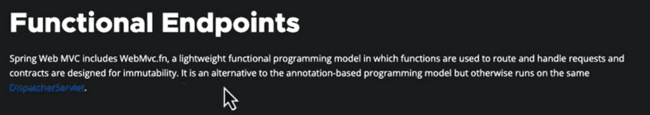
Some of the features of the spring cloud gateway are:



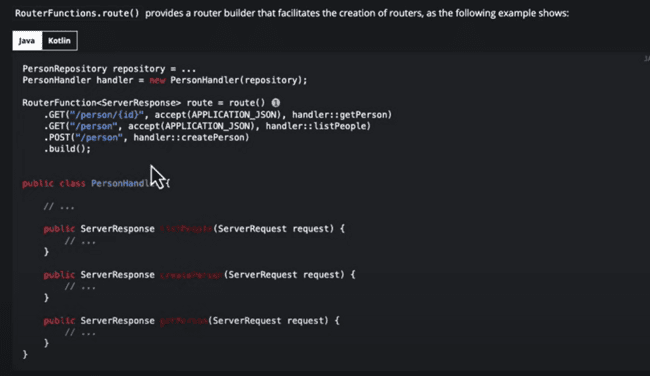
We need to define the different paths in the API gateway to redirect the requests to the specific microservice.

**We are using here FUNCTIONAL ENDPOINT PROGRAMMING MODEL here.**

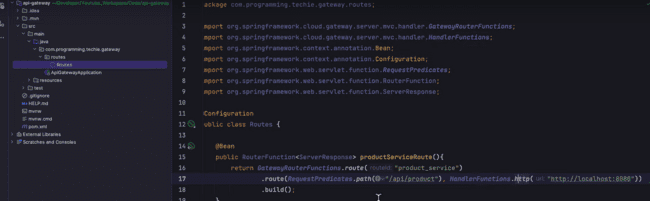
This is an alternative to the annotation programming model and runs on the same dispatcher Servlet.



Functional to annotation example:



Following request coming from the “/api/product” would be forwarded to <http://locahost:8080> and **RequestPredicates** and **HandlerFunctions** are used to handle this functionality.

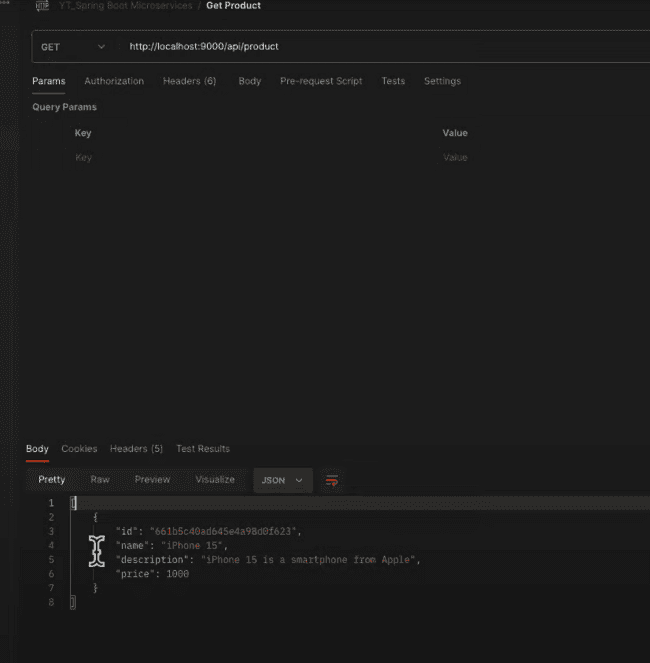


RequestPredicates class contains a lot of methods to headers, content type, whenever there is a header, we can forward that request.

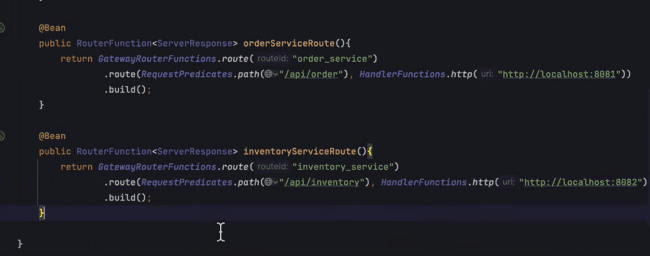
GatewayRouterFunctions.route is the name of the service.

HandlerFunctions.http sends the request to the http.

After deploying the application, we can get the response using the gateway function as:



In the similar way, we need to provide the route for the other services as:



**SECURING THE MICROSERVICES USING THE KEYCLOAK:**

Keycloak is the tool to provide the security to the microservices with minimal efforts and it also do the following as well:

