**Building Microservices with Spring Boot**

Microservices are an architectural approach to application development. A microservices architecture differs from a classic monolithic approach in that it breaks down the application to isolate key functions. Each of these functions is called a “service” and these services can be developed and deployed independently of each other. Thus, each can function (or malfunction) without affecting the others.

Diagram

Description automatically generated

Photo by [Alex Barashkov](https://dev.to/alex_barashkov)

With the growing popularity of microservices in the industry, there has been a boom in technologies and platforms from which to choose to build applications. Sometimes it’s hard to choose something to start with. In this article, I will show you how to create a microservices application based on Spring Boot 🔥 🔥.

Let’s create a small e-commerce application and focus on the product and cart parts only.

Our application will have the following scenarios :

* Add one or many products to the application’s database
* Get product list
* Add product to cart
* Get cart products
* Delete product from the cart
* Delete all products from the cart

Our application will be divided into two µservices, product-service and cart-service . These services will use a message broker to communicate with each other. The product-service will be the **producer**and the cart-service will be the **consumer.**

Now let’s start the application building 😍 .

Generating the product-service

We will generate our project using the [***Spring Initializr***](https://start.spring.io/)***.***

Our application will have 5 dependencies:

* Lombok
* Spring web
* Spring Data JPA
* PostgreSQl
* Eureka cloud client

Graphical user interface, text, application

Description automatically generated

Generating the product-service on Spring Initializr

After generating the project, add it to your favorite IDE 😎.

Now we need to create the **Product**entity :

Then the **ProductRepository** interface :

Then the **ProductController** class :

Now our product-service is ready to run, by default the application runs on port **8080**, but I want to change it because we will have many µservices, and this port we will use later 👌 .

So in the **application.properties** file we will put it at the **8045**and will give a name to our µservice.

server.port=8045spring.application.name=product-service

Now let’s start the product-service 😀:

Graphical user interface, text

Description automatically generated

Product-service booting logs

Our product-service is ready to use 😁. I’m going to use [Postman](https://www.postman.com/) to test it🔥.

Remember, our application uses an H2 database auto-configured. This means we will lose our data once the server is stopped or restarted. we are on a learning project so that’s fine to make things faster 👌

**Add product list :**

<http://localhost:8045/product/addList>

Graphical user interface, text, application

Description automatically generated

Postman UI: add product list

**Add product :**

<http://localhost:8045/product/addOne>

Graphical user interface, application

Description automatically generated

Postman UI: add product

**Show product list :**

<http://localhost:8045/product/getAll>

Graphical user interface, application, Teams

Description automatically generated

Postman UI: retrieve products

Ok, now everything is fine 💪 . As I said before our e-commerce application will use a message broker so µservices can communicate between them.

Now let’s build the cart-service😍 .

Generating the cart-service

As usual, we will generate our project using the [***Spring Initializr***](https://start.spring.io/)***.***

Our application will have the same 5 dependencies comparing to the **product-service**.

Graphical user interface, text, application

Description automatically generated

Generating the cart-service on Spring Initializr

After generating the project, add it to your favorite IDE 😎.

Now we need to create the **Product**entity. It will be the same as the product-service’s entity except for one thing.

We will remove the @GeneratedValue(strategy = GenerationType.IDENTITY) .

Then the **ProductRepository** interface, the same as the previous service :

Next, we will create the **JmsConsumer**,which will listen to **ActiveMQ**. Once there is new data, will consume it.

Now, let’s create the **CartController :**

We are almost done 😎😎. We just need to add some configuration to the properties file :

Now let’s start the cart-service😀:

Text

Description automatically generated

cart-service booting logs

Our cart-service is ready to use 😁.

Let’s check if the cart-servicereceived the product or not 🤔.

<http://localhost:8050/cart/getProducts>

Graphical user interface, application, Teams

Description automatically generated

Postman UI: retrieve cart products

Yes 💪, Our cart-service received the product we sent.

You can also play in the same way with the rest of the scenarios 😄.

Until now, we have something like this :

Diagram

Description automatically generated

Our application architecture

You can notice here that the user can find out that our application is divided into two µservices.

So we need to do something to hide our architecture 🤔.

To achieve this, we need to create an **API Gateway.**

An API Gateway is the single entry point for defined back-end APIs and microservices. Sitting in front of the APIs, the API Gateway acts as a protector, enhancing security and ensuring scalability and high availability.

Now let’s build the API Gateway😍 .

Generating the API-gateway

Our application will have only one dependency this time :

* Gateway

Graphical user interface, text, application

Description automatically generated

Generating the API-gateway on Spring Initializr

After generating the project, add it to your favorite IDE 😎.

This time we have only some configurations to do. No code to add 😜.

For the configuration, I prefer to use the **yml**file instead of the **properties** one.

So you just need to edit the application file extension from**.properties** to **.yml** and add the following configuration :

So here our api-gateway will be listening on **8080**port. Every call to the path /productwill be redirected to the **8045**port and the same thing for /cart to the **8050.**

That’s it, Our API gateway is now ready to run 😎.

Graphical user interface, text

Description automatically generated

API-gateway booting logs

Now, we can run all the calls using the **8080**port.

<http://localhost:8080/product/getAll>

Graphical user interface, text, application, email

Description automatically generated

Postman UI: retrieve products

<http://localhost:8080/cart/getProducts>

Graphical user interface, application, Teams

Description automatically generated

Postman UI: get cart products

After adding the api-gateway , the project’s architecture becomes like this :

Graphical user interface, application

Description automatically generated

Our application architecture

Now, what happens if the cart-service or the product-service changes address or port?

We will have to change the gateway’s configuration too. But as we said before, the µservice should be independent of others.

So there is a solution for this using **Spring Cloud Netflix — Eureka**

Eureka is a client-side service discovery allows services to search and communicate with each other without hard-coding the hostname and port. The only “fixed point” in such an architecture is a service registry that each service must register with.

Let’s generate our Eureka server 😍

Generating the Eureka-server

Our µservice will have only one dependency :

* Eureka Server

Graphical user interface, text, application

Description automatically generated

Generating the Eureka server on Spring Initializr

After generating the project, add it to your favorite IDE 😎.

Inside the **EurekaServerApplication.java**we need to enable the Eureka server using @EnableEurekaServer

Then add the following configuration to the **application.properties :**

Run the service and go to <http://localhost:9000/>

Graphical user interface, text, application

Description automatically generated

Eureka server UI

As you can see, our Eureka-serveris ready now 🔥.

Now we need to configure the other µservices to register within the server.

In the pom file of each µservice we need to add the **Eureka Client** dependency :

For each µservice, we need to enable the Eureka client using the following annotation @EnableEurekaClient :

ProductServiceApplication.java

CartServiceApplication.java

ApiGatewayApplication.java

Now, we will add the following configuration for the product-server and the cart-server :

application.properties

The following lines for the API-gateway :

application.yml

Now let’s restart these 3 µservices, then refresh the Spring Eureka page to see what will happen :

Table

Description automatically generated

Eureka’s registred services

Yes, our services are successfully registered to the Eureka server 💪.

Note that when starting services the Eureka Server must be already up else µservices will fail to register and will show error in logs.

Now the last thing to do is to update the gateway URIs:

After this modification, when addresses and ports change the API-gateway doesn’t have to worry about forwarding requests 😎.

In the end, our project architecture looks like this :

Graphical user interface, diagram

Description automatically generated

Our final application architecture