

Pierre-Émile Brassard

Repo : [https://github.com/ThePhobosDeimosSuite/laboratoire-0\\_log430](https://github.com/ThePhobosDeimosSuite/laboratoire-0_log430)

---

Projet is available in production here : log430@10.194.32.176

Run `docker compose build` and `docker compose up` in projet folder which is located at `~/lab0`

# Introduction and Goals

---

## Requirements Overview

A small business composed of 5 different shops, a supply center and a main shop needs an application to operate. As such, the app needs to keep track of sales and the inventory for each shop on top of a wide array of product. There's also an online shop where users can create an account and fill in their cart with items before checking out.

## Quality Goals

Data needs to be saved and shared amongst all the different users. The service has to be hosted on a VM.

## Stakeholders

Name	Expectations
Manger	Create and update product.
Supply center employee	Deliver new stock to a store, see current stocks.
Store employee	Search product, create sales, search sales, cancel sales, see stocks, order more stocks.
Online customer	Create an account, add items to cart, see items in cart, checkout.

# System Scope and Context

---

## Business Context

The app has to execute 16 different use cases, each being initiated by request made to the REST API.

## Technical Context

Users will communicate with the service with http request. Data will then be stored in a database for persistence.

# Solution Strategy

---

The service has been split into 6 different parts, each being implemented into a standalone microservice:

- Product
- Stocks
- Sales
- Account
- Shopping Cart

These will have their own database and be able to communicate with each other when needed. (For instance: reducing stocks after a sale) An api gateway will analyse requests and forward it to the corresponding microservice.

## UML

---

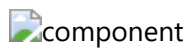
### Class Diagram



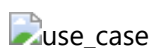
### Deployment view



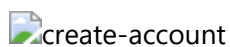
### Component view



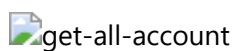
### Use case



#### Create account



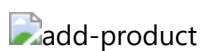
#### Get all account



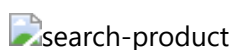
#### Checkout sales



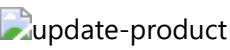
#### Add product



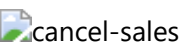
#### Search product



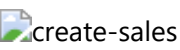
Update product



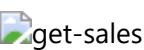
Cancel sales



Create sales



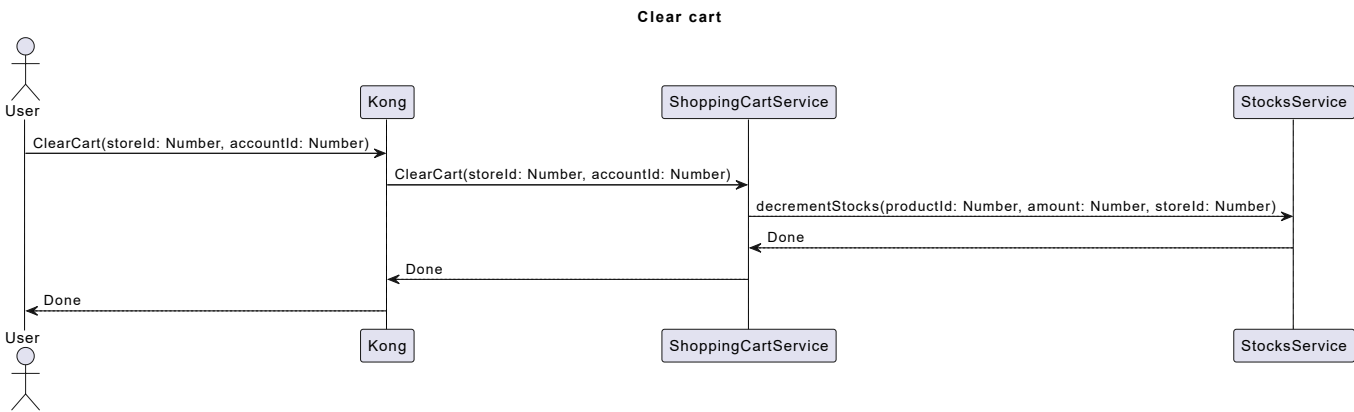
Get sales



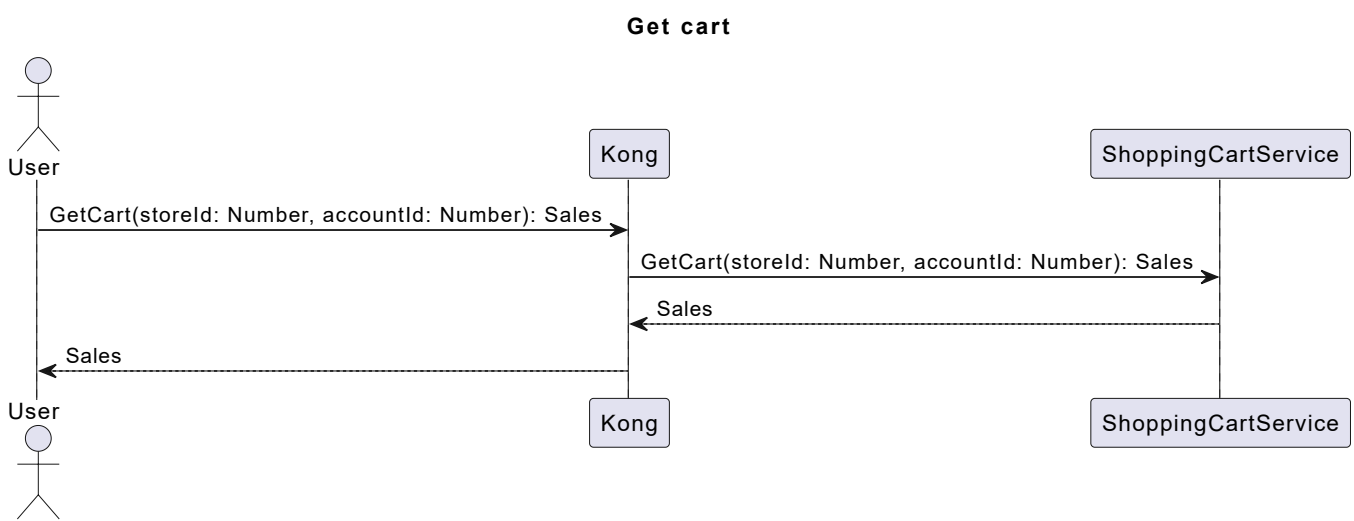
Add items to cart



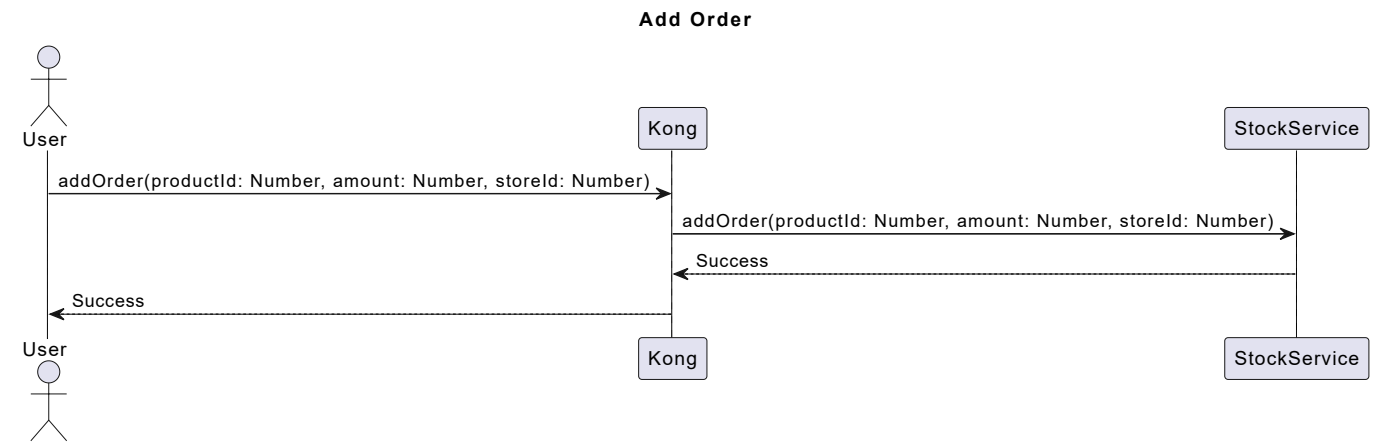
Clear cart



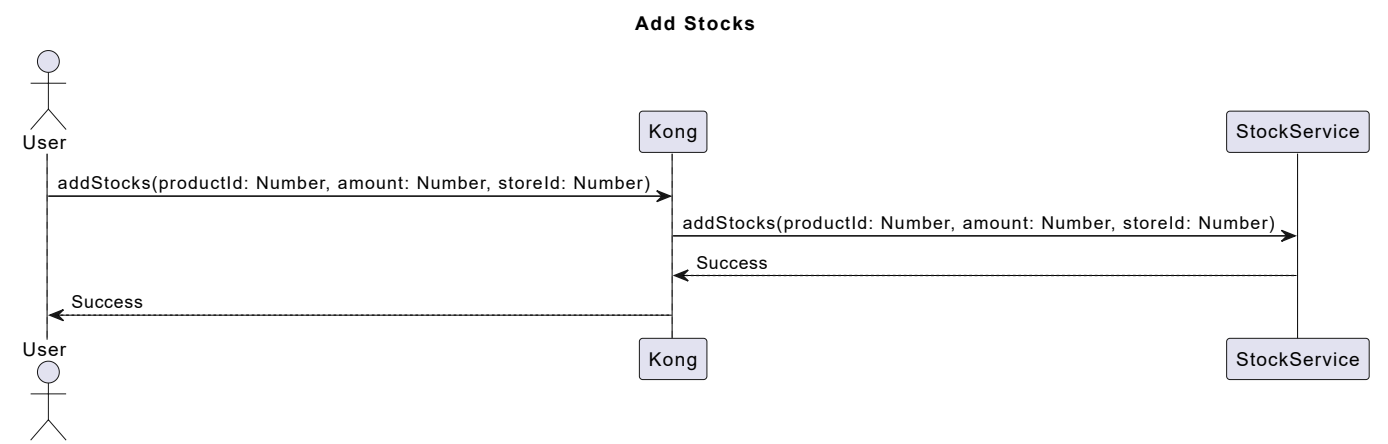
Get cart



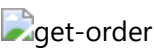
Add order



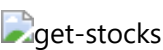
Add stocks



Get order



Get stocks



# Design Decisions

Kafka

Context

Each microservice has it's own database, which means we need a way to tell the *StocksService* when a sales has been completed. We thus need a library to allow microservices to speak to each other and stay up to date.

Decision

An event based messaging system like *Kafka* would allow microservices to send messages between each other and stay in sync.

Status

Accepted

### Consequence

After a sale has been complete, the *SalesService* will send an event to the *StocksService* to decrease the stock of the sold items.

When starting the project using *docker-compose*, there's no way of waiting until *Kafka* has initialized before sending request to it. Hence, the microservices start sending message while *Kafka* is still down and i wasn't able to find a good way to fix this.

Also, i can't find a way to make Kafka work with Jest, which means that the unit tests can't run anymore for any service implementing a Kafka consumer/producer.

---

## Shared utils

### Context

The project being split into 6 smaller ones, some piece of code can be used between two or more microservices.

### Decision

Using the *workspaces* feature in *npm*, it's possible to create a */shared-utils* and import it has a package to each individual microservice. As such, code used to parse request body can be shared.

### Status

Accepted

### Consequence

This is great because it avoids copying code but i've had issues with the implementation. Packaging each microservice to a docker image was really difficult because the */shared-utils* folder has to be included. Also, I've had so many issues with *rootdir* in *jest.config.ts* and to this day running jest on Linux still isn't working. **UPDATE:** After working on *lab6*, I found a way to make Jest work on Linux, the fix is implemented in *lab5*.

## Risks and Technical Debts

---

### Jest

~~Like mentionned previously, jest doesn't work on Linux because of the shared packages between microservices.~~ It's also very annoying to test with *jest* because you need a database up and running with the correct schema. A different connection URL to *PostgreSQL* is defined in *jest.setup.js* but it's not the cleanest way in my opinion. (**Edit:** This should be somewhat fixed in *lab7* as I've added a way to mock the database)

To run Jest, you need to manually generate the prisma client with `npx generate prisma` and build with `npm run build` for **each** modules beforehand. This is annoying.

Also, any service that has a Kafka producer/consumer doesn't run with Jest which is definitely something that has to be fixed at some point because the unit test for `SalesService` always fail now.

## Docker image

Each microservice generates a huge docker image. I had a lot of issues trying to add the `/shared-utils` to each image so instead i'm copying the entire project to each image (minus what's defined in `.dockerignore`). This on top of `/node_modules` taking around 300MB means alot of storage and memory is taken by Docker.

## Adding product description

When looking up the stocks of a store you get the `productId`. Before changing everything to a microservice architecture it was pretty easy to incorporate the product information to the request as such :

```
{
  "productId": "1",
  "product": {
    "name": "Orange",
    "price" : 5
  }
}
```

But now the `StocksService` doesn't have access to the product information so we get less info, which is the reason why the dashboard and sales-report are deprecated now. A way to fix this would be to have the api gateway query the `ProductService` and append the info.

# Technology chosen

---

## Node.js

*Javascript* is untyped so it's flexible with how it processes data. I think it's useful when creating a small app where you don't really want to pay too much attention to what type a certain field is and instead focus on making the app work.

## Typescript

I still think having types is handy sometimes, of instance it can be useful when dealing with data coming from the ORM.

## Prisma

I've never worked with this library before, but there seems to be a huge amount of documentation online. I'm also somewhat unfamiliar with ORMs so i just asked *ChatGPT* for the best library for this particular project.

## Jest

This is what I've always used. It was my first time setting up the environment for Jest and I had a hard time making it work with *ESM* and *Typescript*.

## Kafka

I've never implemented a system like *Kafka*, so i picked this one because it's well known and popular.

## Kong

I used Apollo in another project which i quite liked, but this was the chance to try something new. I had a real hard time with the documentation for *Kong* so i ended up relying alot on *ChatGPT* which wasn't any better.

## File structure

- `/src/module` has all of the microservices, which the structure is defined below.
- `/src/shared-utils` has common code shared between all the microservices.

### Microservice

- `/test` has the unit tests.
- `/prisma` contains the database schema.
- `/src` has the source files for the microservice.