Politenico di Milano

Middleware Technologies for Distributed Systems

Project Report

Food Delivery Application

Author:
Fabio Losavio
Matteo Velati
Niccolò Zangrando

Supervisor:
Dr. Alessandro Margara

April 11, 2021

1 Introduction

1.1 Description of the given problem

In this project, you have to implement a food delivery application. The application consists of a front-end that accepts requests from users and a back-end that processes them. There are three types of users interacting with the service: (1) normal *customers* register, place orders, and check the status of orders; (2) *admins* can change the availability of items; (3) *delivery men* notify successful deliveries. The back-end is decomposed in three services, following the microservices paradigm: (1) the *users* service manages personal data of registered users; (2) the orders service processes and validates orders; (3) the *shipping* service handles shipping of valid orders. Upon receiving a new order, the order service checks if all requested items are available and, if so, it sends the order to the *shipping* service. The *shipping* service checks the address of the user who placed the order and prepares the delivery.

1.2 Assumptions and Guidelines

Services do not share state, but only communicate by exchanging messages or events over Kafka topics; they adopt an event-driven architecture: you can read chapter 5 of the book "Designing Event-Driven Systems" to get some design principles for your project.
Services can crash at any time and lose their state; you need to implement a fault recovery procedure to resume a valid state of the services.
You can assume that Kafka topics with a replication factor ≥ 2 cannot be lost.
You can use any technology to implement the front-end (e.g., simple command-line application or basic REST/Web app).

1.3 Technologies

The technology involved in the development of this project is Apache Kafka.

2 Design and Implementation

As the project requires, the back-end has been structured as a Service Oriented Architecture (SOA) by decomposing it in three different microservices, each one directly connected to a different kind of users allowed to use this application: the *customers* interact with the `UserService`, the *admins* interact with the `OrderService` and the *delivery men* interact with the `ShippingService`.

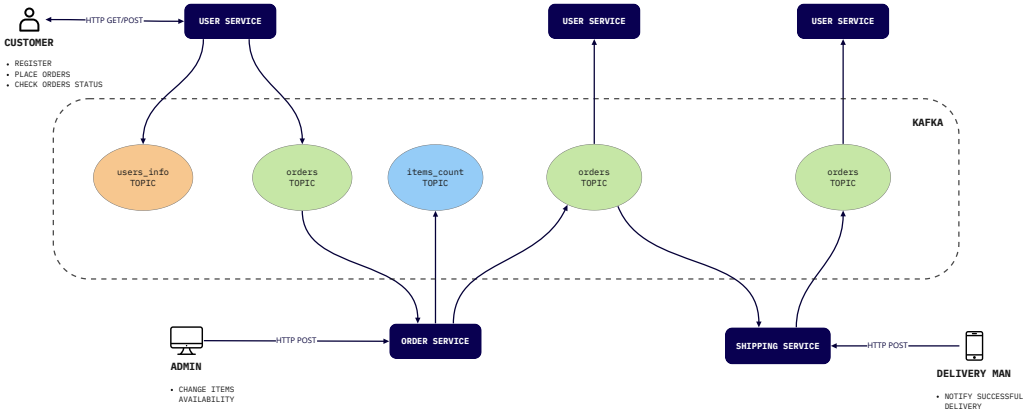

Figure 1: *Overview of the Kafka service-oriented architecture*

The communication between the microservices happens by exchanging events (*JSON* messages) over Kafka topics, following the event-driven system pattern: there is a particular topic, the ORDERS TOPIC in figure 1, which is used by all the microservices and takes an important part in the whole interaction flow, from the placing of a new order to the notification of a successful delivery. Each microservice publishes and is subscribed to that specific topic; to understand which service should react to a specific event, they filter all messages based on the order *status*, which can be:

Submitted Set by the `UserService`, meaning that the order has been just published by the customer

Accepted Set by the `OrderService`, meaning that the order has been successfully accepted

Refused Set by the `OrderService`, meaning that the order has been refused due to unavailability of the items requested

Invalid Set by the `ShippingService`, meaning that the order has been canceled due to unreachability of the delivery address

Delivering Set by the `ShippingService`, meaning that the order is being delivered

Delivered Set by the `ShippingService`, meaning that the order has been delivered

All the microservices have some common implementation parts: upon executing, the main function starts the recovery process by rolling back and reading the Kafka log of the topic, if any: by doing so, the state is recovered if a failure happened during the normal execution of the service, and a local map of java objects is created and stored locally.
After doing so, each microservice execute two threads in parallel: a *KafkaConsumerProducer* and a *InputManager*, respectively aimed at consuming Kafka events on the ORDERS TOPIC and managing *HTTP GET* and *POST* requests from the front-end.

2.1 User microservice

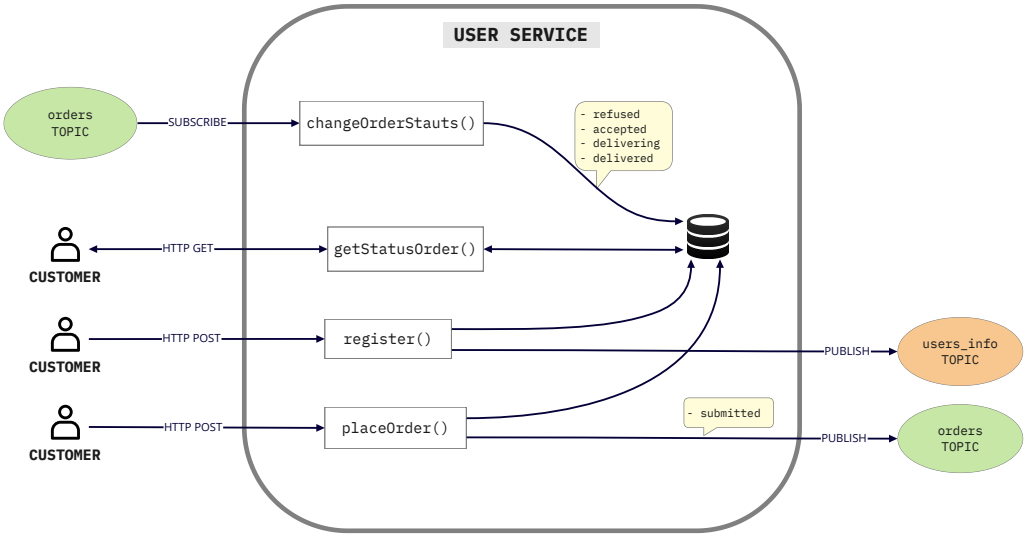

Figure 2: *Detailed view of the User microservice*

This microservice is subscribed to the ORDERS TOPIC in order to be able to update the status of the orders for the front-end users: whenever a publication is made onto the topic from another service, it means that the status of the order has changed and the update is reported on the screen. This topic also works as a recovery topic for all the orders, and its read from the beginning by the service upon starting up.
When the *InputManager* thread is executed, it starts an *HTTP* server on a dedicated port with two different contexts: one dedicated to the log in and sign up processes, the other for submitting new orders and retrieving the status of those.
When a new user registers, his information are stored both locally and on the USERS INFO TOPIC, which works as a recovery topic for the customers. Upon receiving a new order, the microservice changes the status of the order to `SUBMITTED` and publishes on the ORDERS TOPIC the order in *JSON* format.

2.2 Order microservice

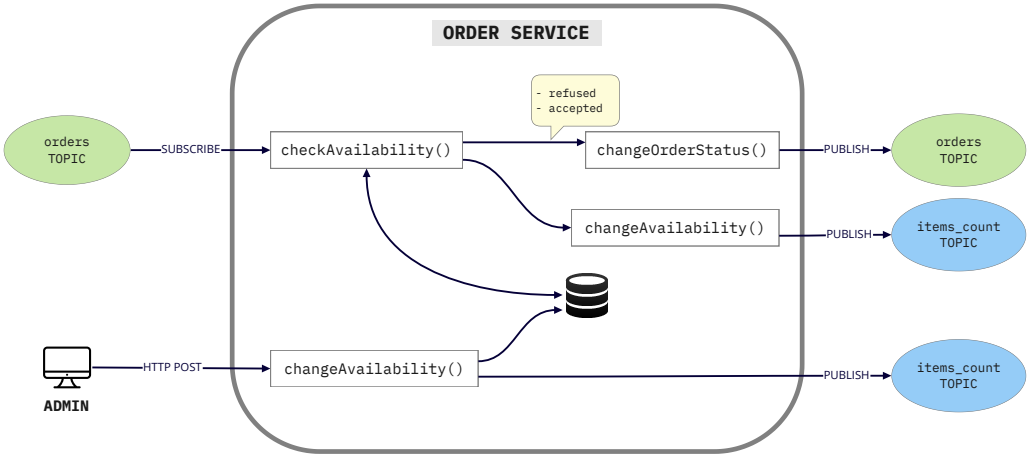

Figure 3: *Detailed view of the Order microservice*

The microservice manages the order validation process and the quantity of resources in stock. Another *HTTP* server is started from the *InputManager* thread and the *admin* user is able to interact with this service to retrieve and change the availability of the items for the restaurant.
On the other thread, the microservice handles the orders which status is `SUBMITTED` or `INVALID`. For submitted orders, it checks the availability of the items requested and it changes the status accordingly before re-publishing the order on the topic. Whether the order has been accepted or not, it also subtract the quantity of item requested from the local database and publishes an update on the ITEMS COUNT TOPIC, a recovery topic for the items.
Regarding invalid orders, the microservice simply recalculates the item availability for the restaurant after the order has been rejected from another service.

2.3 Shipping microservice

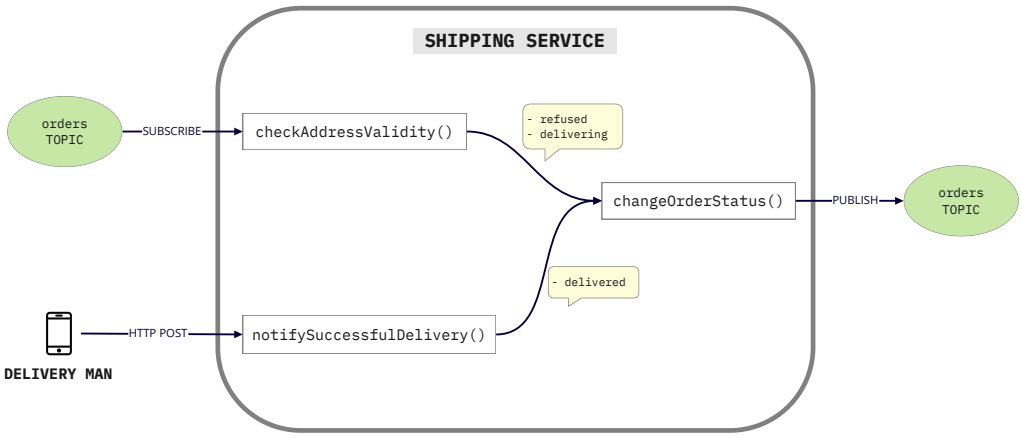

Figure 4: *Detailed view of the Shipping microservice*

The `ShippingService` manages orders that has been accepted: it checks for the reachability of the delivery address and changes the status of the order to `DELIVERING` or `INVALID` accordingly.
On the *InputManager* thread, the *delivery men* can retrieve all the orders that are in the delivering phase and notify to the back-end that the delivery has been completed.

2.4 Front-End

We have implemented a simple web-app to interact with all the services offered by the *Food Delivery Application*.

2.4.1 Customer

After the login the customer can check the status of all the orders already submitted and can create a new one.

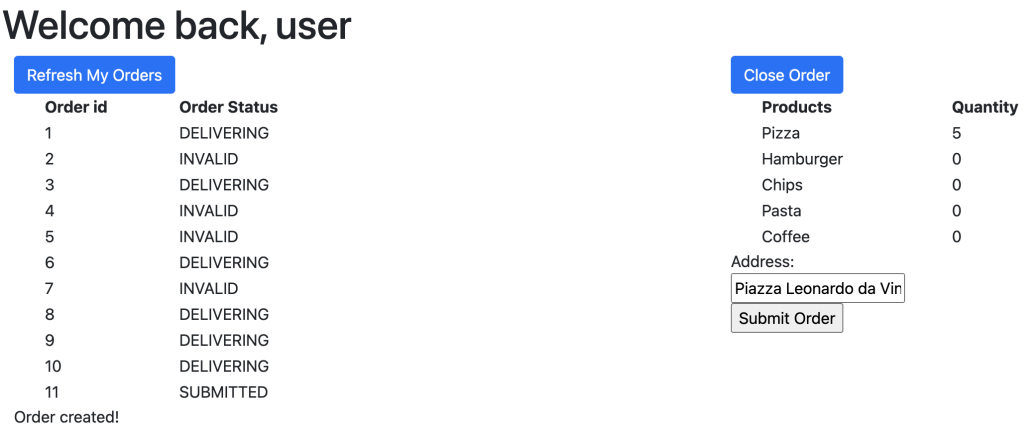

Figure 5: Customer page

2.4.2 Admin

The administrator can change the products availability and can add a new product to the inventory.

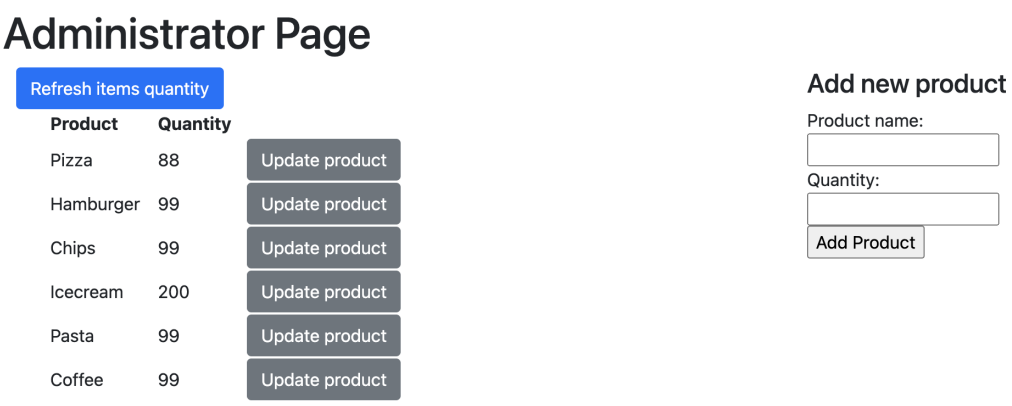

Figure 6: Admin page

2.4.3 Delivery man

The delivery man can check all the orders ready to be delivered and can flag the ones he has already delivered.

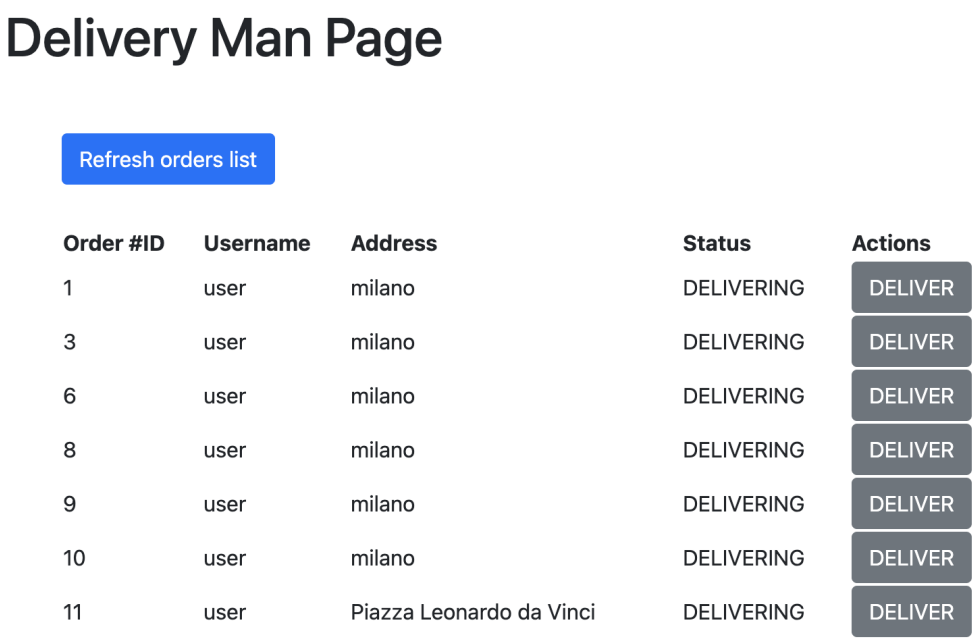

Figure 7: Delivery man page

3 References

- Designing event driven systems book:
  https://www.confluent.io/designing-event-driven-systems/
- Apache Kafka official documentation:
  https://kafka.apache.org/documentation/