Advanced Programming - Project Report

Mattia Michelini (s291551), Manuel Peli (s291485), Francesco Piemontese (s291491), Marco Rossini (s291482)
Politecnico di Torino

***Abstract*—This report illustrates an implementation proposal for a headless e-commerce web application, developed in the Kotlin programming language using the Spring Boot framework. The application leverages an architecture based on four microservices, and features robustness to both logical and physical failures.**

INTRODUCTION

The project consists in the creation of a headless e-commerce web application, implementing the main functionalities required by such a service. The developed system is comprised of four microservices, each exposing their APIs.

This report is divided into five sections. Section I outlines the overall project structure, describing the functionality of each microservice. Section II illustrates the way in which inter-service communication takes place when an order is issued by a customer, providing a detailed overview of the whole process. Section III discusses the types of failure which can occur in order's life cycle, considering the way in which errors are handled by the system. Section IV provides additional details concerning inter-service communication, security and information caching. Finally, Section V addresses the devised application deployment methodology.

The complete source code of the described application can be found in the Git repository available at this link.

I. APPLICATION STRUCTURE

This section discusses the overall structure of the project and its microservices. Each microservice's main purpose and exposed API is presented, together with a description of its core functionality.

Before diving into each microservice, it is essential to discuss our technological stack. The application was developed in the Kotlin programming language, making use of the Spring Boot framework. Elements of the system which require database persistence were recorded using MongoDB. Inter-service communication was implemented via a combination of REST requests and Kafka messages. Specifically, the latter was employed in a publish-subscribe configuration to handle all situations in which, for performance or consistency reasons, REST was not a suitable option.

The implemented solution is comprised of four microservices, a description of which is provided in the following sections.

A. Catalog Service

The Catalog Service is the only service with which customers interact directly. Its main purpose is to provide a set of externally exposed APIs which can be invoked by a customer in order to perform typical e-commerce operations. Being the only externally exposed service of the whole system, the Catalog Service is in charge of checking all the permissions on all the requested operations. If authenticated, a customer can place an order containing the products they wish to purchase, and this request will be forwarded to the Order Service which will manage this functionality. If placed successfully, an ID for the newly created order is returned to the customer, who can then use it for monitoring or cancelling purposes. Through the Catalog Service, it is also possible to interact with the Wallet Service: if authenticated, customers can get information about their current funds and transactions, while admins can get this information for all users of the system. Products can only be added or created by admins, while product information can be retrieved by anyone (even by non-authenticated users).

By means of the associated controller, the Catalog Service exposes the following API functions:

Publicly available:

- `getAll()`: retrieves all the products stored in the database;
- `getProductByName()`: retrieves a product based on its name;

Restricted to admins:

- `addProduct()`: adds a product to the database;
- `editProduct()`: updates the properties of a product;
- `deleteProductById()`: deletes a product based on its ID in the database;

Restricted to authenticated users and admins:

- `placeOrder()`: places an order given a list of products and a shipping address;
- `changeStatus()`: updates the status of an order;
- `getWalletFunds()`: retrieves the current balance of a given user's wallet;
- `getWalletTransactions()`: retrieves the list of transactions performed by a specific user's wallet;

Restricted to internal use:

- `getAdminsEmail()`: returns a list of admins to whom to forward a notification e-mail;
- `getEmailById()`: retrieves a user's e-mail based on their ID in the database.

B. Order Service

The Order Service defines the core business logic of the whole system. Its main purpose is to provide a set of internally exposed APIs which can be invoked by other services in order to perform all order management operations. Each operation can be requested without authentication, as the service

is designed to communicate exclusively with other trusted components. The Order Service can be requested to place an order, triggering an interaction both with the Wallet Service and the Warehouse Service: the former interaction is required to check if the customer has enough funds to perform the purchase, whereas the latter is required to check if the required products are present in the warehouses. If these conditions are met, the Order Service will return the newly created order's ID. This ID allows the customer (or an admin) to check or change the status of the order: a customer can perform a status change only from `ISSUED` to `CANCELLED`, whereas an admin can perform any status change. When the status of an order is modified, a notification e-mail is sent to the customer associated with it and to multiple admins (chosen by a custom-defined logic[1]). If an order is cancelled by the customer, or its status is changed to `CANCELLED` or `FAILED` by an admin, a rollback procedure (managed by the Kafka messaging system) is triggered on the Wallet and Warehouse Services.

By means of the associated controller, the Order Service internally exposes the following API functions:

- `placeOrder()`: places an order for a specific user;
- `orderStatus()`: retrieves the status of an order given a specified order ID;
- `changeStatus()`: updates the status of an order given a specified order ID.

C. Wallet Service

The Wallet Service is in charge of handling all the operations related to the customers' funds. Its core element is the `Wallet` model class, which is composed by the available funds they have on their e-commerce balance, together with the list of all the transactions that have been issued on it. Every transaction is composed of multiple elements, among which the issuer and the motivation for the funds update. For robustness reasons, it was decided to keep in memory all the transactions (including cancelled ones) issued in a customer's wallet, in order be able to reconstruct the history, and hence the available funds, at any given time. In general, this service provides a simple set of APIs which allows other internal services to retrieve information about a customer's wallet, or to add a new transaction to it (e.g. a wallet recharge performed by an admin, or the payment for an order performed by a customer).

By means of the associated controller, the Wallet Service exposes the following API functions:

- `availableFunds()`: retrieves the current balance of a given customer's wallet;
- `transactionList()`: retrieves the list of transactions performed on a specific customer's wallet;
- `addTransaction()`: adds a transaction to a specific customer's wallet.

[1]In the case of this project, a dummy logic which retrieves the first 3 admins in the database was defined. However, a more sophisticated logic can be employed in a real use-case. For instance, e-mails can be sent according to a geographical criterion (e.g. to the regional sales managers).

D. Warehouse Service

The Warehouse Service is in charge of administering the multiple warehouses in which products may be stored[2].

The representation of products employed in this microservice is quite simplified compared to the one used in the Catalog Service, as it does not take prices, descriptions or categories into consideration (the responsibility of tracking these properties is delegated to the Catalog Service). Each warehouse contains a list of products (referred to as `inventory`), each of which is characterized by an ID and a non-negative quantity. Warehouse inventories are independent of each other, meaning the same product may be found in multiple warehouses with different quantities.

Each warehouse product is additionally assigned an integer alarm threshold (which also may vary depending on the warehouse), meant to notify supervisors of stocks running low. If at any point the quantity of a product drops below the threshold, a notification e-mail is sent to a list of supervisors, which are specified as part of the `Warehouse` model object. This implementation means to replicate a practical use case, in which each warehouse would have a designated worker responsible for handling restocks.

By means of the associated controller, the Warehouse Service exposes the following API functions:

- `createProduct()`: creates a new product in a specified warehouse;
- `editProduct()`: edits the quantity of a warehouse product (creating it if it is not already present);
- `editAlarm()`: edits the alarm threshold associated with a warehouse product;
- `warehouseInventory()`: returns a list of the products stored in a specified warehouse;
- `deliveryList()`: given an order cart (containing the requested products and the associated quantities) computes a list of deliveries capable of satisfying the request.

Note that, because a product may be present in multiple warehouses, there may exist multiple ways in which a given order may be fulfilled. Whenever a request is made to `deliveryList()`, the Warehouse Service must therefore decide which products should be pulled from which warehouse. In practice, this decision would be based on both product availability and the warehouses' geographical location; however, because the latter information is not available in this project, the following heuristic process was adopted instead:

1) select from the cart the product p for which the requested quantity is greatest;
2) select the warehouse w containing the largest amount of product p;
3) withdraw from warehouse w any available product which is needed in the fulfillment of the order;
4) remove the withdrawn items from the cart.

[2]For the purposes of this project, it was deemed sufficient to employ the Warehouse Service as a central warehouse coordinator. Note however that in a practical use case, each warehouse may be governed by a separate microservice instance.

Steps (1) through (4) are repeated iteratively until either all cart products have been successfully withdrawn (in which case the request succeeds) or no warehouse containing a requested item can be found (in which case the request fails and a rollback is issued).

In order to model the loading and withdrawal of products from a warehouse's inventory, the concept of transactions was introduced into the `Warehouse` model object. Conceptually, this approach is very similar to the one employed in the Wallet Service to handle the modification of user funds (see Section I-C). In the Warehouse Service, transactions are modelled by the `WarehouseTransaction` class, comprised of the following attributes:

- `productId`: the unique identifier of the modified item;
- `quantity`: the amount by which the product's quantity was altered (negative if withdrawn, positive if deposited);
- `issuerId`: either an admin's or an order's unique identifier, depending on whether the product modification was issued as part of a restock or to fulfill an order;
- `motivation`: a descriptive attribute denoting the reason behind the quantity change.

The need for a `motivation` field stems from the decision (mirroring the approach taken in the Order Service) of prohibiting the deletion of warehouse transactions. In order to counteract a modification in quantity, a new transaction for the opposite amount must be created instead.

Through the use of transactions, it was possible to make the application resistant to physical failures of the Warehouse Service. In order to achieve this, an additional attribute named `transactionList` was created in the `Warehouse` model class. Whenever an instruction is issued which would modify the quantity of a product in a warehouse, two operations are performed on the underlying database document:

- The appropriate `quantity` field is updated in the warehouse's `inventory`;
- A transaction describing the change is created and pushed into the warehouse's `transactionList`.

By carrying out these tasks as an atomic pair, the Warehouse Service is guaranteed to never perform an untraceable operation. If, while withdrawing the items required for the fulfillment of an order, the service were to physically crash, upon coming back online it would always be able to retrace its steps, returning the collected products to the appropriate warehouse.

II. Order placement

Order placement is the key functionality provided by our application, involving all of the developed microservices. Figure 1 provides a detailed overview of the inter-service communication which occurs each time an order is placed by a customer. Full-headed arrows are used to denote REST API calls, while hollow-headed ones indicate the exchange of Kafka messages. The numbered red dots represent points in which a service may fail, either logically due to some problems with the request (e.g. insufficient funds in the user's wallet) or physically due to a system crash. An in-depth analysis of how these errors are handled is provided in Section III.

The entry point for an order placement request is the Catalog Service, the only microservice that is exposed to customers. Here, checks are performed to asses whether the user performing the request is authenticated, as well as whether the requested products are present in the application's catalog (rejecting requests for products which either do not exist or are not eligible for sale).

Subsequently, the request is propagated to the Order Service which, after performing some additional consistency inspection, creates an empty `Order` object, publishing its ID via a Kafka message. This is done to ensure that, even in the occurrence of physical failures, the system is able to detect whether orders were successful, and to return to a consistent state (for details, refer to Section III).

Once this is done, the Order Service contacts the Wallet Service, asking whether the customer's funds are sufficient for the purchase of the requested products. If the answer is affirmative, a negative transaction for the total amount is pushed in the user's wallet, and the Order Services proceeds to request a delivery list from the Warehouse Service. If this step is also successful, the created `Order` object is persisted in the database, and its ID is returned to the customer for tracking purposes. Additionally, confirmation e-mails are sent to both the user and the responsible administrators.

When an order is placed successfully, its status is set to `ISSUED`; if its life cycle progresses correctly, it is later switched to `DELIVERING` once the requested items have left the respective warehouses, and finally to `DELIVERED` once the indicated shipping address has been reached. These status transitions are carried out by system administrators, such as warehouse supervisors approving item shipments.

The customer who issued the order may decide to cancel it, provided that it is still in the `ISSUED` state; if this happens, the status is changed to `CANCELLED`, and a rollback message is generated by the Order Service. If the items are already on their way, the customer may not alter their decision. On the other hand, admins may change the order status to `CANCELLED` or `FAILED` at any time, as unforeseen issues may arise during the delivery of products (e.g. a breakdown of the means of transport). Note that all the aforementioned status alterations are performed by means of REST calls.

For testing purposes, we included in the root of the project's Git repository a Python script named `orderRoutine.py`, which simulates the advancement of an order's life cycle, through operations such as placing an order or modifying the status of an existing one with administrative privileges.

III. Failure Handling

The way in which the rollback was handled is both straightforward and effective. Immediately after receiving a purchase request, the Order Service generates an empty order, whose ID is published by means of a Kafka message. Upon receiving this message, a Kafka listener (also placed in the Order Service)

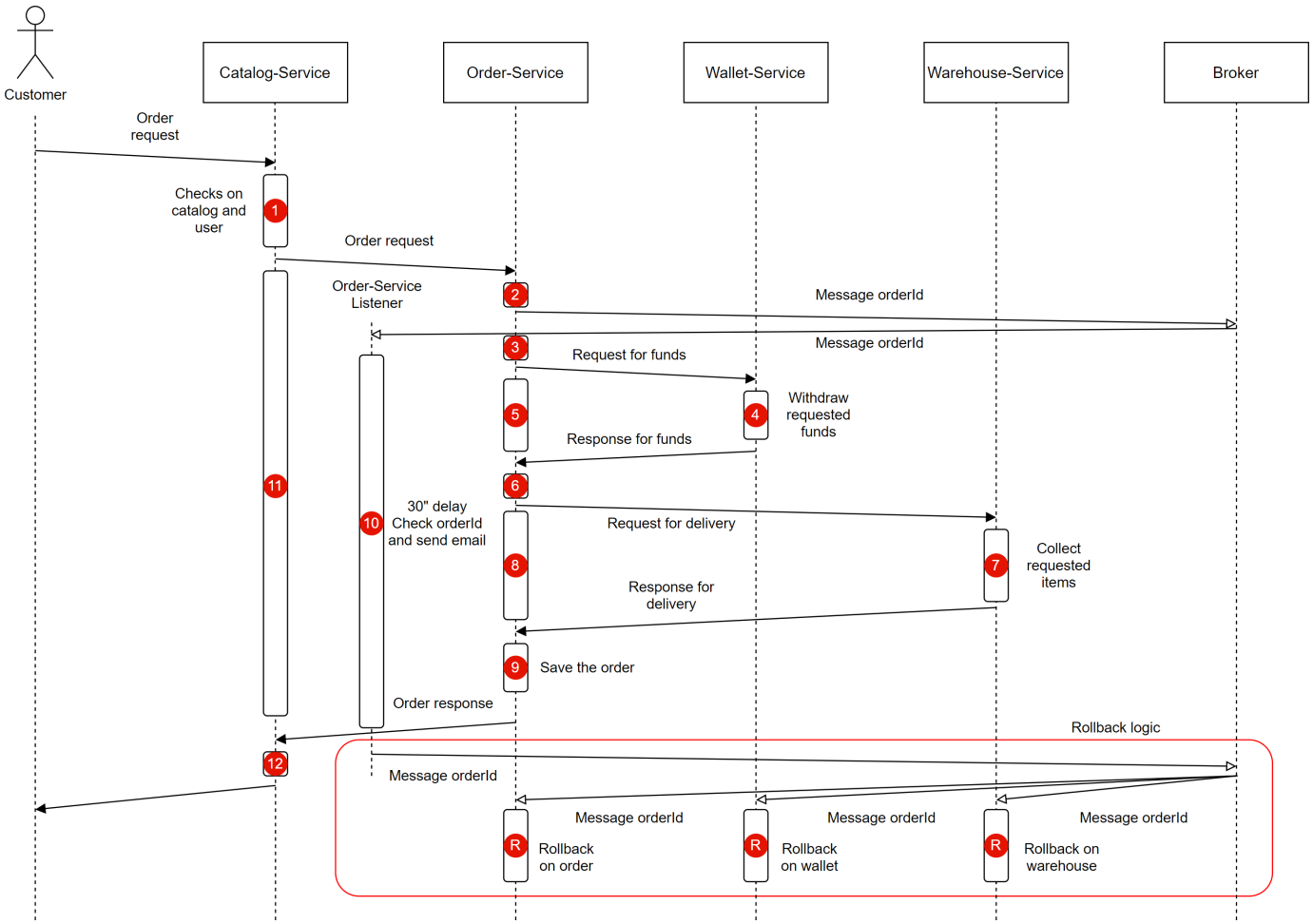

Figure 1. Description of the service interactions involved in an order placement.

allows a time of 30 seconds[3] to elapse, and subsequently checks whether a matching order ID is present in the database. If this is not the case, the order placement is assumed to have failed, and a rollback message is broadcasted. Two additional Kafka listeners, placed within the Wallet and Warehouse Services, are in charge of consuming this instruction, initiating the rollback procedure of the respective microservice.

Thanks to this messaging architecture, it was possible to ensure the system's robustness to the (temporary) physical failure of any of its composing microservices. This was achieved by leveraging two properties of Kafka message handling:

1) if a listener is offline when a message is published, the message is not lost, but is instead processed as soon as the listener is back online;
2) if a listener fails while processing a message, the message is not consumed, but it is instead processed again once the listener is back online (up to a specified number of times).

The above properties guarantee that the Warehouse and Wallet Services always receive and complete rollback requests, even in the event of them being offline when the message is published or crashing while carrying out the procedure. Moreover, property (2) ensures that, despite physical failures of the Order Service, all order ID consistency checks are eventually be performed, leading to no order failure going unnoticed.

In the following, we provide an in-depth description of our application's error handling methodology. The items of the numbered list refer to the coloured dots employed in Figure 1, and mark the position of possible errors in the order placement timeline. Round markers (•) indicate failure types (as more than one error may occur in the same numbered location), while arrows (→) denote the way in which they are handled.

1) • Product not present in the catalog
   • Physical failure of the Catalog Service
   → Order fails, no rollback needed
2) • Physical failure of the Order Service
   → Order fails due to order request timeout, no rollback needed
3) • Physical failure of the Order Service
   → Order fails due to order request timeout, rollback is performed (as the Kafka message has already

[3]This threshold was selected for demonstration purposes, and should be modified in practical use cases.

been sent) but has no effect

4) • Physical failure of the Wallet Service
   → Request for funds fails, order fails, rollback is performed but has no effect
5) • Physical failure of the Order Service
   → Order fails due to order request timeout, rollback is performed (affecting only the Wallet Service)
6) • No wallet associated with the user
   • The user's wallet is present, but the contained funds are insufficient
   → Order is not placed, rollback is performed but has no effect
   • Physical failure of the Order Service
   → Order fails due to order request timeout, rollback is performed (affecting only the Wallet Service)
7) • Physical failure of the Warehouse Service
   → Request for delivery list fails, order fails
8) • Physical failure of the Order Service
   → Order fails due to order request timeout, rollback is performed (affecting both the Wallet and the Warehouse Services)
9) • The requested products are not in stock
   → Order is not placed, rollback is performed (affecting only the Wallet Service)
   • Physical failure of the Order Service
   → Order fails due to order request timeout, rollback is performed (affecting both the wallet and the Warehouse Services)
10) • Physical failure of the Order Service
   → When the service returns online, the Kafka listener restarts processing of the order ID message, ensuring eventual order consistency
11) • Physical failure of the Catalog Service
   → The customer does not receive an order ID within the initiated session, but does receive a confirmation e-mail once the consistency check is performed
12) • Physical failure of the Catalog Service
   → The customer does not receive an order ID within the initiated session, but does receive a confirmation e-mail once the consistency check is performed

IV. Implementation details

In this section, we provide insight into a number of implementation choices we made during this project's development, out of either performance or robustness considerations.

Throughout the application, inter-service communication is mainly handled through REST requests: provided that no error arises, placing an order results in the generation of a single Kafka message, both produced and consumed by the Order Service to ensure system consistency. In order to improve the system's scalability, we therefore chose to make use of Kotlin coroutines. This decision is especially significant considering the fact that, to carry out a single order placement, at least five REST calls need to be performed. The use of coroutines allowed us to write asynchronous code using a clear, synchronous-like syntax. As previously discussed, Kafka messages were instead utilized as the primary tool for handling consistency checks and rollback operations. The advantages of this approach are detailed in Section III.

As per the assignment specifications, all requests exchanged between microservices are considered trusted: the only point in which security is taken into account is the communication between the user and the Catalog Service. Some functionalities, such as searching and displaying products, are allowed to all user types, while others, such as reading one's order status or wallet funds are only permitted to logged-in users; administrators have access to all functionalities.

Finally, we elected to use caching in order to speed up the computation of frequently used used elements. Specifically, caches were implemented in the Catalog and Warehouse Services, for the quick retrieval of product and warehouse information. In both these cases, we opted to employ a short time to live for elements inserted in the cache, thus preventing data from becoming stale, especially considering that we do not know our system's selling volume and the frequency at which item prices will change.

V. Deployment

Docker-based containerization was leveraged to facilitate the deployment and testing of the developed application. In the root of the project's Git repository, we included two files, named `docker-compose-build.yml` and `docker-compose.yml`. The terminal commands required to run both files are available in the project's `README`.

When run, the former file creates a single Docker container named `build-service` which, using the `gradle bootJar` command, builds the JAR files required to run each microservice.

After completing the build process, the latter file may be employed to run the application. Seven containers are spawned in total: these are required to run an instance of each of the developed microservices, as well as MongoDB, Zookeeper and the Kafka broker.

A few concluding remarks, concerning the way in which the Docker network was created, are in order. Firstly, it should be noted that while all services rely on the same instance of MongoDB, each may only access and modify the collections of which it is responsible. In a practical use case, a separate Mongo instance may be employed for each service.

Secondly, it must be pointed out that although the Catalog is the only service meant to be reachable by customers, all four microservices were assigned a port number. On top of facilitating testing, this choice is meant to simulate a real-world use case, in which system administrators would be able to access individual services (e.g. to restock warehouse inventories) through a private API.