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.**

I. Project overview

Kotlin e Spring Boot

This section provides an overview of the project and specifies the associated goals, specifications and requirements. The complete source code of the application covered by this report can be found in the Git repository available at this link.

II. Application structure

TODO ancora da dire che usiamo NoSQL

This section presents and discusses the overall structure of the project and its microservices. For each microservice, a description of its main purpose and exposed APIs is provided, together with a description of its core functionality.

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 he or she 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, and he or she can send a request to cancel the order (only if its status has not already been updated to the `DELIVERING` status or any of the following ones). 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 products 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 his or her 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 internal 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 services. 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 one 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 changed, a notification e-mail is sent to the customer associated to 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, then a rollback procedure is triggered on the wallet service and the warehouse service, which is managed by the Kafka messaging system.

[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 defined for a real use-case. For example, e-mails can be sent according to a geographical criterion (e.g. regional sales managers).

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 customer's funds. Its core element is the `Wallet` model class, one associated to each user, which is composed by the available funds that a customer have on its e-commerce balance and the list of all the transactions that has been done on it. Here, every transaction is composed by many information and among those, the issuer and the motivation of this operation are the most important in order to assure that everything is working correctly. In addition, we decided to keep all the transactions that have happened in the customer's "life", even the one that have been cancelled, in order be able to reconstruct the history, and so 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 user's wallet, or to add a new transaction to it, like a wallet recharge by an admin or the payment of an order. Specifically, the following functions are exposed:

- `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.

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 catalog, as it does not take prices, descriptions or categories into consideration (the responsibility of tracking these properties is delegated to the catalog). 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 warning email is sent to a list of supervisors, which are specified as part of the `Warehouse` model object.

[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.

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 to 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.

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 II-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.

III. Security

*__TODO__: check if ok

Parti sincrone fatte con coroutine We chose to use kotlin coroutines in this project in order to be able to scale better our system. This is particularly significant considering the fact that to carry out a single order placement at least 5 REST requests need to be performed. Using coroutines allows us to simply write the functions as they were sequential, but they are suspended when they have some active waiting time.

Parti fatti con Kafka In the project, we decided to use kafka as the core system to carry out the rollback throughout the system, because of its features that allow us to be able to resist physical failure of our services. More on this later, where we describe how we use it to build our rollback routine.

How caching is used We chose to use caching in this project in order to speed up the computation of frequently used elements. We chose to implement it in the catalog service, when we retrieve the products from the catalog database, as well as in the warehouse service, when we retrieve information about the warehouses from the its database. In both these cases we decided to have a short life time for the elements inserted in the cache, because we were scared of stale data, considering the fact that we have no idea about the selling volume of our system and the frequency at which the products change their price. For these reasons, we chose the safest trade-off in the use of the cache.

Given the specifications of the project, all communications among micro services are considered to be trusted, the only point where security is taken into account is in the communication between user and catalog service.

Some functionalities, such as searching and displaying products are allowed to all types of users; other functionalities such as placing an order, reading the order status, modifying it, reading the funds and transactions on one's own wallet are only allowed to logged in users; admins have access to all functionalities.

Communication mode is publish/subscribe for what concerns Kafka

IMPORTANTE: dove mettere questa parte? Magari dove si parla di Order? When an order is placed correctly, its status is set to `ISSUED`; if its lifecycle progresses smoothly, it will switch to the `DELIVERING` status when the products leave the warehouses and finally to the `DELIVERED` status when it reaches the shipping address indicated. These canonical changes are performed by admins (e.g. the warehouse manager when the products leave the warehouse), but the customer that issued the order can also decide to cancel it, but this is possible only if it is still in `ISSUED` status. In this case the order will become `CANCELLED`; so if the products are already on their way, the customer cannot cancel his or her decision. Admins, on the other hand, can change the status to `CANCELLED` or `FAILED` at any time, as there may be problems with the delivery of products (e.g. a breakdown in the means of transport).

IV. 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 V.

The entry point for an order placing 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 indeed 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 V).

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

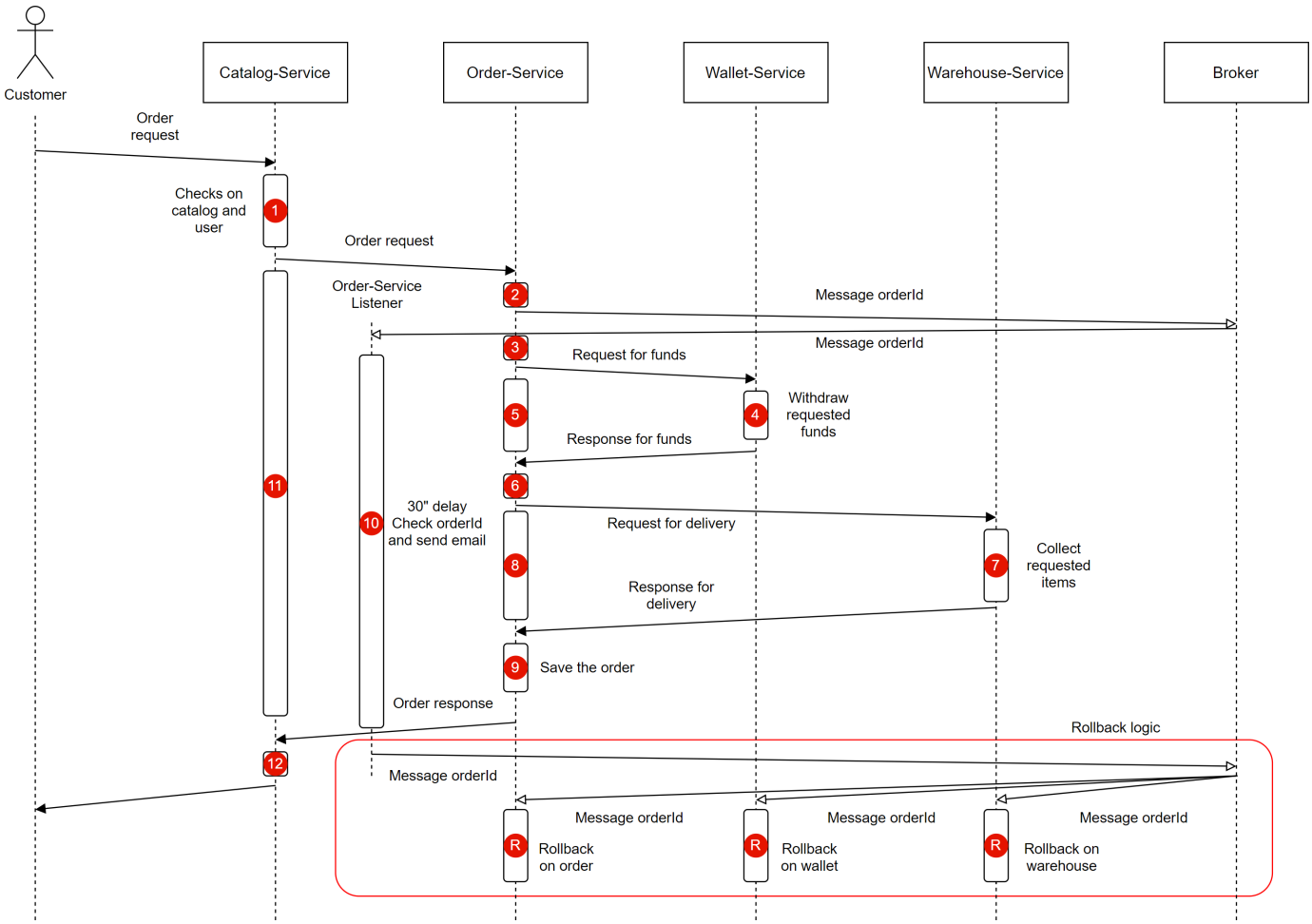

Figure 1. Order placement timeline.

purposes. Additionally, confirmation emails are sent to both the user and the responsible administrators.

V. 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) 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 would 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.

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

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 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 listeners 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 email 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 email once the consistency check is performed

VI. 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`.