Web Application 2

2020-21

Course Final Project: Distributed eCommerce Service

Build a headless (i.e. just the API) eCommerce application structured in a microservice architecture.

1 Project Overview

The application is composed of four microservices:

- **CatalogService:** For simplicity this is the only service handling authentication. Customers interact with this service only. They can list products, get their features and availability, without need of authentication. To each user is associated a wallet, they can use to purchase products. Authenticated users can place an order, provided that there is enough money in their wallet and the chosen products are available in the warehouses. As result they receive an Order object with an ID they can use to check its status or to cancel the order until shipping is not started. Customers also receive an email each time their order is updated. Authenticated users can also place an evaluation of the purchased product, assigning stars and leaving a comment. Admins can add and edit product properties or upload the product pictures. Customers and Admins should have an account on the platform, obtained via the registration phase. The registration should be confirmed via a link received via email.
- **OrderService:** It is the core of the system. It stores orders and their status and it is responsible of enforcing the rules on orders. Orders can be placed only if the customer has enough money in his/her wallet and products are available in warehouses. Order placing and all the related operations should be atomically isolated at system level and be transactional, assuring that system will remain consistent after any successful or unsuccessful operation, i.e. if some service fails, for any reason (even power-down), every other operation already done must be rolled back. After the purchase, the order status must be updated accordingly to the progress and both admins and users must be notified via an email. In case of failure or when an order is cancelled, the customer must be refunded and the items must be returned to the warehouses. This service APIs can be used only by other internal services (all services are considered trusted and they can perform any operation). Customers and Admins can query and modify order status only through the CatalogService accordingly to their permissions. The orders status can be for example:
  - Issued: order accepted and paid, ready to be delivered
  - Delivering: the items have left the warehouse
  - Delivered: successful order
  - Failed: it can fail in any moment. In case of failure, items are returned to the warehouse, if already taken, and the customer is refunded, in case s/he had been already billed
  - Canceled: canceled by the user. Cancel requests are considered only if the status is Issued: once the delivery process begins, it is not possible to cancel orders anymore.
- **WalletService:** Wallets handle customer money. The customer can query her/his total balance, the transactions list, and add a new transaction. Negative transactions are issued during order placement, positive ones (recharges) are issued by admins only**.**
- **WarehouseService:** It handles the list of products stored in any warehouse. Products can be in more than one warehouse, with different quantities. Each warehouse has a list of alarm levels for any product: when the quantity of a product is below the alarm level a notification email must be sent to the admins. The APIs allows the listing of products and their quantities, loading and unloading items and updating alarms.

2 System Organization

2.1 Overall Architecture

As a microservice architecture is composed by a set of loosely coupled services, they can be implemented using different technology stacks.

- You can choose between Spring Boot (Blocking or Non-Blocking) using Kotlin and NodeJS using JavaScript (or TypeScript)
- Each microservice should have its own database, following the shared-nothing architecture. You can choose a SQL or NoSQL database system. Even if databases are different, there can be a single DBMS hosting them all
- The communication should happen only across well-defined interfaces: these may be synchronous (you can choose between REST and GraphQL) or asynchronous (via Kafka)
- Services should be deployed as separate runtime processes, preferably in Docker containers, in order to host microservices inside a single operating system. All the containers should be interconnected leveraging Docker's internal network, allowing them to communicate. In addition, the CatalogService should be exposed to the external network in order to be reachable from outside. The DB instance runs in its own container.

2.2 Objects

The objects handled by the services are:

- **Product**: product name, description, picture URL, category, price, average rating and creation date. Comments with title, body, stars and creation date can be associated to purchased products
- **Warehouse**: information pertaining to storage of a product in a given warehouse; any warehouse has a different list of products, with different quantities; any product may be in one or more warehouses
- **Order**: any order has a buyer, a list of purchased products, their amount, the purchase price (product price may vary), a status
- **Delivery**: any order has a delivery list, indicating the shipping address and the warehouse products are picked from (it can be embedded in orders or be a separate part)
- **Wallet and Transaction**: each user has a wallet with the current amount and the transactions, containing the amount transacted and the reason (the order id or the recharge reference)
- **Users**: users have name, surname, email, delivery address and two roles: customer, admin

2.3 Endpoints

Internal APIs

Internal APIs are exposed by the internal services and are used to allow interactions among them. Those are not directly accessible from the external, neither from the users, nor from the admins. They can be organized as follows:

- **/orders**: the /orders endpoint is exposed by the OrderService and is used to retrieve and update order-related information
- **/products**: the /products endpoint is exposed by the WarehouseService and is used to retrieve and update product information
- **/wallets**: the /wallets endpoint is exposed by the WalletService and is used to manage user wallet and the related transactions
- **/warehouses**: the /warehouses endpoint is exposed by the WarehouseService and is used to manage warehouse information (e.g., the products availability)

The main orders endpoints are:

- **GET /orders**: Retrieves the list of all orders
- **GET /orders/{orderID}:** Retrieves the order identified by orderID
- **POST /orders**: Adds a new order
- **PATCH /orders/{orderID}:** updates the order identified by orderID
- **DELETE /orders/{orderID}:** Cancels an existing order, if possible

The main products endpoints are:

- **GET /products?category=<category>:** Retrieves the list of all products. Specifying the category, retrieves all products by a given category
- **GET /products/{productID}:** Retrieves the product identified by productID
- POST /products: Adds a new product
- **PUT /products/{productID}:** Updates an existing product (full representation), or adds a new one if not exists
- **PATCH /products/{productID}:** Updates an existing product (partial representation)
- **DELETE /products/{productID}:** Deletes a product
- **GET /products/{productID}/picture**: Retrieves the picture of the product identified by productID
- **POST /products/{productID}/picture**: Updates the picture of the product identified by productID
- **GET /products/{productID}/warehouses**: Gets the list of the warehouses that contain the product

The main wallets endpoints are:

- **GET /wallets/{walletID}:** Retrieves the wallet identified by walletID
- **POST /wallets**: Creates a new wallet for a given customer
- **POST /wallets/{walletID}/transactions**: Adds a new transaction to the wallet identified by walletID
- **GET /wallets/{walletID}/transactions?from=<dateInMillis>&to=<dateInMillis>:** Retrieves a list of transactions regarding a given wallet in a given time frame
- **GET /wallets/{walletID}/transactions/{transactionID}:** Retrieves the details of a single transaction

The main warehouses endpoints are:

- **GET /warehouses**: Retrieves the list of all warehouses
- **GET /warehouses/{warehouseID}:** Retrieves the warehouse identified by warehouseID
- **POST /warehouses**: Adds a new warehouse
- **PUT /warehouses/{warehouseID}:** Updates an existing warehouse (full representation), or adds a new one if not exists
- **PATCH /warehouses/{warehouseID}:** Updates an existing warehouse (partial representation)
- **DELETE /warehouses/{warehouseID}:** Deletes a warehouse

External APIs

In order to use the internal APIs, is necessary to define some external APIs that can be accessed from an user or an admin, depending on their authorization. Those are all exposed by the CatalogService. The CatalogService behaves as an API gateway for the other microservices. As a matter of fact, it exposes some external APIs that are redirected to the internal APIs of the corresponding microservice without providing extra logic.

Since those APIs are exposed to the external, you can use REST o GraphQL to implement them. Use validation on input data to enforce constraints when a request is made by external users.

The normal users can:

- Retrieve his/her orders or a specific order
- Create a new order
- Cancel an order, until shipping is not started
- Retrieve products or a specific product, and its picture
- Retrieve his/her wallets
- Retrieve/update own user information

Admins have access to all information.

It is necessary to enforce security to the application implementing authorization (for customers and admins) and authentication.

In order to perform authenticated requests, additional endpoints will be required. Implement endpoints for registering new users, validating their email addresses through an email, support updating their password and login to the platform validating user's credentials with a JWT. Only admins will be able to assign the admin role to customers, calling a specific API exposed only to admins.

3 The Order Saga

Order placing is a critical process and must be handled transactionally at a system level, assuring that system will remain consistent after any successful or unsuccessful operation, trying to ensure ACID properties also in the microservices environment. Indeed, moving to microservices, while the goal is to create loosely coupled services, chances are high that one service needs a particular data set owned by another service, or that multiple services need to act in concert to achieve a consistent outcome of an operation.

Sagas allow for the implementation of long-running, distributed transactions, executing a set of operations across multiple microservices, applying consistent all-or-nothing semantics. For "rolling back" the overarching transaction in case of a failure, Sagas rely on the notion of compensating transactions: each previously applied local transaction must be able to be "undone" by running another transaction which applies the inversion of the formerly done changes, in atomically way. In this way, once the local transactions of the Saga have been committed, their changes are persisted and durable, also after a service failure and restart.

There are two general ways for implementing distributed Sagas—choreography and orchestration, so it is necessary to choose one of them.

- In the **orchestration approach**, the saga is managed by a dedicated service acting as orchestrator, in charge of dispatching local transactions to the services. If any service fails, the orchestrator sends rollback commands to all the services
- In the **choreography-based approach,** every microservice performs the local transaction and publishes domain events that trigger local transactions in other services.

As far as the communication between the participating services is concerned, this may happen either synchronously, e.g., via HTTP, or asynchronously, e.g., via message brokers or distributed logs such as Apache Kafka.

The Order Saga will include the OrderService, WalletService and WarehouseService: it will deal with the process of order verification, issuing and delivering, taking into account possible service and logical fails that may happen throughout the process. As the Saga coordinator should preferably communicate asynchronously with participating services, use Apache Kafka to communicate on specific topics.

It is also necessary to deal with logical errors, in particular if the service dies after it has persisted the needed changes to its local database, but before it has acknowledged the message to the Kafka broker. Indeed, in this case the broker has no way of detecting whether the service has completed processing its message and will proceed to resend it once the consumer goes back alive. It is necessary to guarantee that, if that happens, changes already persisted to the database are not repeated. Tools like Debezium (https://debezium.io) may help in this case.