Course Project Distributed eCommerce service

Services

Build a simple headless (i.e. just the API) eCommerce with Spring Boot composed by the following microservices

- **CatalogService**
  Customers interact with this service only. They can list products, get their features and availability (availability comes from warehouses)
  To each user we associate a wallet they can use to purchase products
  Customer can place any order, given there is enough money in their wallet and the products are available in the warehouses. As result they receive and **Order** object with a tracking id they can use to check the status or cancel the order until shipping is not started.
  Customers also receive an email each time their order is updated
  Admins can add and edit product properties
  Both Customers and Admins already have an account using their email is the username. For simplicity this is the only service handling authentication

- **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 a global level and be transactional, i.e. if some service fails, for any reason, every other operation already done must be rolled back (see later for details).
  After the purchase, the order status must be updated accordingly to the progress (picked up from the warehouse, shipping in progress, shipped, canceled, error - you can choose any set of states useful for modeling order handling) and admins and users must be notified via an email
  In case of fail or when an order is canceled the customer must be refunded and the items must be returned to the warehouse.
  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 the CatalogService accordingly to their permissions.

- **WalletService**
  Wallets handle customer money, they have simple API: you can query the total, the transaction list, and add a transaction.
  Negative transaction are issued during purchase, 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 list of alarm levels for any product; when the quantity of product is below the alarm level a notification must be sent to the admins
  The APIs allows the listing of products and their quantities, loading and unloading items and updating alarms.
  For simplicity you can use a single WarehouseService handling more than one warehouse, but in real life there can a separate instance for each warehouse.

Objects

The objects that must be handled by the services

- **Product**: product name, description, picture category, price
- **Warehouse**: where products are stored, 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 a 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, whis is an external key). The wallet may be either materialized or computed based on un transactions; transactions can't be deleted
- **Users**: users have name, surname, email, optional delivery address and two roles: customer, admin

Handling orders

Orders have a minimum set of states (it is possible to add more states if they useful for better handling the process, e.g. you can create a “pending” order and set it to “issued” later)

- issued: order accepted and paid, ready to be delivered
- delivering: the items have left the warehouse and they are being delivered
- delivered: successful order
- failed: it can fail in any moment, items are returned to the warehouse
- canceled: canceled by the user, they can be canceled only before delivering has started

Chose any strategy suitable for simulating the advance of the order process; e.g. in real life scenario warehouses will inform the OrderService when the products are ready for pickup and the courier will send updates of the delivery. You can use private APIs in the services of write some custom logic to advance them

Permissions and Roles

Services are trusted and all their APIs are used internally: in this simple scenario you can do any operation without authentication.

The CatalogService instead must check permissions on all the operations, since it is exposed externally.

Basic rules on exposed objects:

- Products:
  - read and listed by anonymous readers
  - modified only by admins
- Orders:
  - created by authenticated customers only
  - customers can modify their own orders in accordance to business rules
  - admins can do anything

Constraints

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 . Two types of fails must be handled:

- service fails, e.g. the operation is logically correct but the service stops working while performing it
- logic fails: e.g. two customers start placing and order of the last product in stock, but only one will succeed; a customer places two orders without enough funds for the second one, only one order must succeed (this is realistic for automatic ordering)

For scalability it's forbidden to have a global read lock during order placing therefore it is impossible to avoid some of the described logical fails, and they must be handled with rollbacks

For scalability and maintenance it is also forbidden to do unnecessary operations in sync with the order. For example all the notifications and the alarms must be handled separately.