Technical Design - Backend

Installations:

|  |  |
| --- | --- |
| Software | Link |
| IntelliJ community | <https://www.jetbrains.com/idea/download/#section=windows> |
| Java 17 for windows | <https://www.oracle.com/il-en/java/technologies/downloads/#jdk20-windows>  java -version |
| Mongo | <https://www.mongodb.com/try/download/community> |
| RabbitMQ | <https://www.rabbitmq.com/download.html> |
| Redis | <https://redis.io/download/> |
|  |  |

Mongo DB model

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Users (Root) | | Id | | FullName | | Password | | Address | | Role Id | | Company (DBRef) | |  | | Audit Data | |  | | |  | | --- | | Roles (Root) | | Id | | Name | | Desc | | Audit Data | |  | |  | | *Available options: Admin, employee, customer* | | |  | | --- | | Address (child) | | Telephone | | Address | | Email | |  | |  | | |  | | --- | | Company (Root) | | id | | Name | | Currency | | Audit Data | |  | |  | |
| |  | | --- | | Product (Root) | | Id | | Name | | Desc | | Price | | Discount | | Discount Type {Percentage/Fixed Amount} | | Category (DBRef) | | Inventory | |  | | Company (DBRef) | | Audit Data | |  | | |  | | --- | | Product Category (Root) | | Id | | Name | | Desc | | Company (DBRef) | | Audit Data | |  | |  | |  | | |  | | --- | | AuditData (child) | | Create Date | | Update Date | |  | |  |
| |  | | --- | | Orders (Root) | | Id | | Employee (DBRef) | | Customer (DBRef) | | Total amount | | Order Items List | | Order Status Id | | Company (DBRef) | | Currency | | CreditCardNumber | | ExpiryOn | | Cvc | | NotificationFlag | | Audit Data (DBRef) | | |  | | --- | | Order Items (Child) | | Product id | | Amount | | Quantity | |  | |  | | |  | | --- | | Order payment (Root) | | Order Id | | User | | Amount | | Invoice number | | Credit/debit | | Audit Data | |  | |  |

APIs

|  |  |
| --- | --- |
| **Signup** | |
| Parameter | Company Name, Full Name, Email, Password |
| Action | Populate new contact and new user as admin |
| Return value | Token: compose of company id, user id, role id |
| **Login** | |
| Parameters | Email, Password |
| Action | Authentication validation: check user and password in users table, associated to company |
| Return value | Token: compose of company id, user id, role id |
| **Product Category (Create, Edit, Delete)** | |
| Parameter | Name, Desc |
| Action | Crud for product Category, associated to company |
| Return value | success/failed |
| **Product (Create, Edit, Delete)** | |
| Parameter | Name, Desc, inventory, Price, Category, Discount list |
| Action | Crud for product, associated to company |
| Return value | success/failed |
| **User (Create, Edit, Delete)** | |
| Parameter | Full Name, Address, Role |
| Action | Crud for customer, associated to company |
| Validation | Verify unique user by email |
| Return value | success/failed |
| **Get Users** | |
| Parameter | Page Number |
| Action | Select users with pagination, associated to company |
| Return value | List of uesers order by role and update date |
| **Get Customers** | |
| Parameter | Prefix Name |
| Action | Select the customer based on name associated to company |
| Return value | List of Pair <Name, User Id> |
| **Get Products** | |
| Parameter | Prefix Name |
| Action | Select the products based on name associated to company |
| Return value | List of Pair <Name of product, product Id> |
| **Calculate Order Amount** | |
| Parameter | Order |
| Action | For each product, calculate the amount based on discount and Currency. Calculate also total amount. See more details below |
| Return value | Map like this:  {0} <PRODUCT\_X, Map<Amount, Discount>>  {1} <PRODUCT\_Y, Map< Amount, Discount >>  {2} <-1, Map <Total Amount, null>> |
| **Create Order** | |
| Parameter | Employee (user Id), Customer (user id), list of items {product id, quantity, amount}, Currency |
| Action | Create new order. Handling queue. |
| Validation | Verify the Status: New or Approved |
| Return value | Order Id |
| **Update Order** | |
| Parameter | Order Id, updated list of items {product id, quantity, amount}, Currency |
| Action | Update the list of items, add new item/delete items/update amount or quantity. Handling queue. |
| Validation | Check the previous and target status. Valid options:  New -> Approved  Packing -> Delivered  Packing -> Cancelled |
| Return value | success/failed |
| **Get Orders** | |
| Parameter | Employee (user Id), Status, Page Number |
| Action | Select orders with pagination, associated to company |
| Return value | List of products order by update date |
| **Get Currency** | |
| Parameter | None |
| Action | Define 5 currencies as hardcoded |
| Return value | List of hard coded currencies |
| **Top sold product (Bar Chart)** | |
| Action | For last three months, associated to company, select top sold product based on delivered orders |
| Return value | List<Month, List <Product Name, Count>> |
| **Deliver/Cancel Orders (Line Graph)** | |
| Action | For last three months, associated to company, select orders based on cancel/deliver status |
| Return value | Map <Month, Map<Count cancelled, Count delivered >> |
| **Top employee (Pie Chart)** | |
| Action | For last three months, associated to company, select top employees based on deliver orders |
| Return value | List <User, Count of delivered orders > |
|  |  |

**High Level Design**

This application designed for manage printing houses. As Muti-tenancy application, single instance of application serves multiple stores. For registration new store, the admin user should sign up first time, by signup the new store registered as new company with generic details like company name and currency.

Admin user has ability to add new employees and manage the product catalog, both employees and catalog are related to admin user’s company.

All the APIs is associated to company. Once user login to application, the login API retrieve token that contains the company id (related to the user), this token is send by every API header.

Employee has ability to manage the customers and orders that also associated to company based on employee’s company.

**Order State Machine**

A picture containing screenshot, text, diagram, design

Description automatically generated

Once new order is created by employee, the immediate status can be New for further changes, Cancelled (upon customer request) or Approved.

Once order gets status Approve, charging process is starting for update inventory and perform payment, the next status is Packing.

If the product is OOS or the payment was declined the order is Cancelled.

From status Packing, there are 2 options, move to Cancelled upon customer request or Delivered.

There are two microservices for manage the application:

* Order manager service (below first service)
* Payment processing service (below second service)

**Order manager service functionality**

This service is central component of the application:

* The interaction between frontend and backend will be done by this service.
* Manage the product catalog, users, customers, and orders.
* This service exposes the above APIs list for react app.
* The data is saved to dedicated schema in MongoDB. All collections should be in this schema exclude order payment collection.

**Payment processing service functionality**

This service processes the payments asynchronously.

* This service is not exposed to frontend at all.
* The data is saved to dedicated schema in MongoDB, Only Order payment collection should be in this schema.
* Apply payment by third party application (mocking server in our implementation by postman)

**Calculate Order Amount API**

Since the order currency may be different than product currency, system should calculate the exchange rate. For better performance system should get the today’s rate for order’s currency by external web-service (in our implementation postman mocking server) and put it in Redis, in next time, first need check in Redis if already exists rate for order’s currency and today and save the calling to external we-service.

**Communication between the microservices**

To support large scale of orders simultaneously. The communication should be done be queue (RabbitMQ) and not directly by API. The first service should place the order in queue named “PaymentWaiting” and the second microservice should listen to this queue and process the order. Once it done the order should be placed again now in another queue named “CompletedPayment”, and the first service should listen to this queue and continue the process.

**When put order in “PaymentWaiting” queue?**

* By “Create Order” API, if the immediate status is Approved.
* By “Update Order” API, if status is Approved (changed from New)

Status should be changed to Charging before putting in queue.

**Charging step BL**

Once order changed to approved by user, system should change the order status to Charging and do below steps.

* Handle Inventory
  + verify the order quantity in products’ inventory.
    - In case of OOS, change order to Cancelled (OOS)
    - Otherwise change product inventory (according to order items list)
* Handle Payment
  + Create OrderDTO object and put in correct queue (see more details below)
  + Add listener to “CompletedPayment” queue.
    - If status in OrderDTO is Approved - change order status to packing
    - Otherwise – change order status to Cancel (Payment Decline) and update back the product inventory accordingly.

**Payment processing BL**

Create new DTO object {Data Transfer Object} and call it OrderDTO. The object should be placed in both services.

The fields of this object:

* Order Id
* Customer Id
* Payment Amount
* Status
* Credit/Debit
* Credit Card Details

The first service should populate the DTO with relevant attributes and put in “PaymentWaiting” queue.

If status is Approve – Debit, If status is Cancel – Credit.

Create mock server by postman for simulate third party payment approval, the mocking server should retrieve status approved/declined, in case of done return also invoice number.

The payment should be saved in MongoDB.

Once the processing is done, the orderDTO should be placed again in “CompletedPayment” queue.

The first service should pop messages from “CompletedPayment” and check the status.

**Email Async Process**

Create Spring async job, run every 30 mins. This job should select orders from tables with notificationFlag = false.

Once order status is changed from one status to new status – the notificationFlag should be set as false

Once email/SMS send to user with order details – the notificationFlag should be set as true

**Some technical points to start with:**

The microservices are spring boot applications.

Initialize by <https://start.spring.io/>. Spring boot version 3.1.0, Maven, Java 17.

Dependencies:

* Spring Data MongoDB
* Spring for RabbitMQ
* Lombok
* Spring Data Redis

Create mock server in postman, please follow [link](https://learning.postman.com/docs/designing-and-developing-your-api/mocking-data/setting-up-mock/)

More info about Data Transfer Object [here](https://www.baeldung.com/java-dto-pattern)

More info on spring cron process [here](https://www.baeldung.com/spring-scheduled-tasks)

**Extra Tasks:**

* add docker compose for mongo, rabbit and redis
* add logging configuration by log-back
* install elastic search and shift logs
* end email notifications