

**Applied Software Project Report**

By

<Nourth Arvinder Pal Singh >

**A Master’s Project Report submitted to Scaler Neovarsity - Woolf in partial fulfillment of the requirements for the degree of Master of Science in Computer Science**

<Month of Submission March, 2025>



**Scaler Mentee Email ID :** <arvinderpalsingh2321@gmail.com>

**Thesis Supervisor :** Anurag Khanna

**Date of Submission :** 25/03/2025

**Certification**

I confirm that I have overseen / reviewed this applied project and, in my judgment, it adheres to the appropriate standards of academic presentation. I believe it satisfactorily meets the criteria, in terms of both quality and breadth, to serve as an applied project report for the attainment of Master of Science in Computer Science degree. This applied project report has been submitted to Woolf and is deemed sufficient to fulfill the prerequisites for the Master of Science in Computer Science degree.

Anurag Khanna

…………………

Project Guide / Supervisor

**DECLARATION**

I confirm that this project report, submitted to fulfill the requirements for Master of Science in Computer Science degree, completed by me from 30-05-2024 to 20-11-2024 is the result of my own individual endeavor. The Project has been made on my own under guidance of my supervisor with proper acknowledgment and without plagiarism. Any contributions from external sources or individuals, including the use of AI tools, are appropriately through citation. By making this declaration. I acknowledge that any violation if this statement constitutes academic misconduct. I understand that such misconduct may lead to expulsion from the program and/or disqualification from receiving degree.

**<Nourth Arvinder Pal Singh>**

**<Signature of the Candidate>                                   Date: 25 March 2025**

**ACKNOWLEDGMENT**

I would like to express my gratitude to my family (Tunzala, and Sahib Singh) for their support, encouragement and belief in me throughout this journey. Their patience and motivation have been my greatest strength. I am thankful to the dedicated instructors and mentors of Scaler community, whose guidance, knowledge, and insightful teachings have played a crucial role in shaping my understanding of computer science and programming. I am thankful to everyone who has inspired and motivated me to push my limits and successfully complete this program. This journey would not have been possible without your support,

**Table of Contents**

[**List of Tables 6**](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.9nnr2lniv90f)

[**List of Figures 7**](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.ju1gc9w3iuai)

[**Applied Software Project 8**](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.b4cf8683b1wd)

[Abstract 8](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.sj7c7bghlznr)

[Project Description 8](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.1z5fx61h0cc)

[Requirement Gathering 9](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.joagy45av5k0)

[Class Diagrams 9](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.nvf4h831fm8o)

[Database Schema Design 9](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.ydqs8nkbe6m9)

[Feature Development Process 11](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.p6mfl8dwb9sy)

[Deployment Flow 12](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.2mk44ad33gi)

[Technologies Used 12](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.wn68bn10ag78)

[Conclusion 13](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.4yf46wt6rx84)

[**References 14**](https://docs.google.com/document/d/1Hm5pABxGPu1xOUB0P-2WA2siPD0uBkwU1vhaaDb-CKo/edit#heading=h.z0iyzog9l959)

List of Figures

**(List of Images, Graphs, Charts sequentially as they appear in the text)**

|  |  |  |
| --- | --- | --- |
| **Figure No.** | **Title** | **Page No.** |
| **1.1** | Project Development Process | 12 |
| **2.2** | Featured list | 14 |

List of Tables

Contents

[List of Figures 6](#_Toc201245654)

[List of Tables 7](#_Toc201245655)

[Applied Software Project 9](#_Toc201245656)

[Project Description 10](#_Toc201245657)

[Key Components: 10](#_Toc201245658)

[System Architecture Description 12](#_Toc201245659)

[Requirement Gathering 13](#_Toc201245660)

[1. Requirements 13](#_Toc201245661)

[1.1 Functional Requirements 13](#_Toc201245662)

[1.2 Non-Functional Requirements 13](#_Toc201245663)

[2. Users and Use Cases 13](#_Toc201245664)

[2.1 User Types 13](#_Toc201245665)

[2.2 Use Case Diagram 14](#_Toc201245666)

[Class Diagrams 15](#_Toc201245667)

[1. Class Diagram for User Management Service 15](#_Toc201245668)

[Description of the Diagram: 16](#_Toc201245669)

[2. Class Diagram for Product Management Service 16](#_Toc201245670)

[Description of the Diagram: 16](#_Toc201245671)

[Database Schema Design 17](#_Toc201245672)

[Textual Schema Design 19](#_Toc201245673)

[Diagrammatic Schema Design 21](#_Toc201245674)

[Explanation of the Diagram: 22](#_Toc201245675)

[Cardinality and Relationships: 22](#_Toc201245676)

[Feature Development Process 23](#_Toc201245677)

[b. Service Which Picks the Request 23](#_Toc201245678)

[Deployment Flow 24](#_Toc201245679)

[1. Code Commit and Version Control 24](#_Toc201245680)

[2. Continuous Integration (CI) 24](#_Toc201245681)

[3. Continuous Deployment (CD) 25](#_Toc201245682)

[4. Post-Deployment 25](#_Toc201245683)

[5. Rollback Strategy 26](#_Toc201245684)

[6. Deployment Diagram 26](#_Toc201245685)

[Summary of the Deployment Flow 27](#_Toc201245686)

[Technologies Used 27](#_Toc201245687)

[1. Kafka - Distributed Streaming Platform 28](#_Toc201245688)

[2. MySQL - Relational Database Management System (RDBMS) 28](#_Toc201245689)

[3. Spring Boot - Java Framework for Building Microservices 29](#_Toc201245690)

[4. Cloud (AWS, Azure, Google Cloud) 29](#_Toc201245691)

[5. Docker - Containerization 30](#_Toc201245692)

[6. Redis - In-Memory Data Store 30](#_Toc201245693)

[7. Kubernetes - Container Orchestration 31](#_Toc201245694)

[Conclusion 31](#_Toc201245695)

[Conclusion 32](#_Toc201245696)

[Conclusion 33](#_Toc201245697)

[Key Takeaways 33](#_Toc201245698)

[Practical Applications 33](#_Toc201245699)

[Limitations 34](#_Toc201245700)

[Final Thoughts 34](#_Toc201245701)

[References 35](#_Toc201245702)

# 

# 

# Applied Software Project

#### 

#### Abstract

Today E-commerce is one of the widely used application across the world. Its impact become more on the customers especially after covid-19 breakout. As most of people now prefer to buy the products rather visit market, as example of from children’s toys, medicine, grocery, furniture, food almost everything is possible to buy online. it saved valuable time and due to fast delivery services, it becomes more convenient and gain the trustable market.

In this e-commerce Project, I aimed to develop a secure, efficient and scalable platform using backend technologies like the system uses token based security system Oauth2 implemented by spring boot to secure user data and all passwords are encrypted by BcryptPassword Encoder.

To increase the performance Redis is integration so that customer will get fastest browsing experience, the platform also implements role based authentication which separates general users and admin and provide them access to appropriately.

For secure and reliable payments, the system integration stripe which I widely used in most of e-commerce platforms because of it stability and flexible. to provide customer real time update Kafka and java mail notification system provided to customer, the delivery service accurate calculate and displays estimated delivery time after success purchase of item.

To persist data MySQL is best of my knowledge which offers reliable and structured storage for transactional systems.

By completing this project the aim is to show how backend skills and real time integration with API’s can provide better results for both users and businesses in the online shopping industry.

## 

## Project Description

## this E-commerce backend platform project is based on spring boot microservice architect, which includes UserService, ProductService, CartService, OrderService, PaymentService, DeliveryServcie, Kafka and JavaMail service, and API-Gateway for API request routing, eureka server for service discovery, for data storage MySQL, this project will provide seamless communication between services by RestTemplates.

## The best of this project is user can freely browse products can category but when user want to purchase then signup will be required.

## Key Components:

UserService: Role based access, managed by authentication and authorization registration for protection of profile and BcryptPasswordEncorder to encode passwords.

ProductService: handling products and category crud operations, for better performance used Redis integration.

CartService: add or remove product from cart according to customer choice.

OrderService: if cart is confirmed then this service processing orders and making product ready to sell.

PaymentService: most advance payment service as Stripe by providing safe and secure transactions.

DeliveryService: delivery to European countries only and track order status.

KakfaService and JavaMail: used for sending email notification.

Api-Gateway: used as api gateway for product unified entry point for all client requests and routing them to appropriate services.

EurekaServer: it make sure each microservice communicate properly and can be tracked by this server.

**Technologies and tools used:**

* **Programming Language:** Java 17
* **Framework:** Spring Boot
* **Database:** MySQL, database for storing data, user, product and category,cart, orders, and delivery destinations.
* **Notifications and JavaMail :** Kafka, configured via Docker, is used and JavaMail for sending email notifications.
* **Redis:** used for cache via docker.
* Spring Boot Starter OAuth2 Resource Server (for authentication)
* **JWT(JSON WebTokens for security)**
* **API-Gateway : routing requests to services.**
* **Eureka server:** microservice communicate

**E-commerce Project Development Process**

**Figure 1.1**: Project Development Process

* **Team formation** (since I am working solo, I took responsibility for all roles)
* **Topic Selection** -I selected theme of e-comProject Platform because of its real word relevance and potential to showcase multiple backend technologies.and most important popularity of e-commerce websites these day which will give me good learning experience with hands on this project.
* **Creating Project** -I created a detailed plan outlining project goals where I choose microservice architect rather then monolith architect because of it independency of functionality and scalability as (UserService, ProductService, CartService, OrderService, PaymentService, DeliveryService, and separate server for Kafka, Api-Gateway, and Eureka Server)

**Requirement Gathering**

* User-friendly
* Fast
* Loosely coupled
* Scalable
* secure

**User Requirements**:

1. Users should be able to browse products and categories without authentication or authorization.
2. Users must sign up to add products to the cart.
3. After successful signup or login, users should receive a confirmation email.
4. Users can change the quantity of selected products before placing an order.
5. Once the order is confirmed, users should receive a payment link.
6. After payment confirmation, users should receive a delivery notification.
7. Users should have the option to add or update the delivery address before the order is dispatched.
8. Users should see the estimated delivery time or date.

**Admin Requirements** :

1. Admin can add, delete, and manage all services.
2. Admin has access to API Gateway, Eureka Server, and Kafka topics.
3. Admin should have the ability to control and manage APIs.

## 

## 1. Requirements Gathering

### 1.1 Functional Requirements

* User Registration and Role based Authentication (Signup,Login,View all users, delete user)
* Product Management (Add, Update, Delete, View Products,Delete Products,)
* Shopping Cart Management (Add, Remove, Update Items)
* Order Processing (Place Order, Delete Order, Cancel Order)
* Payment Integration (Secure Online Stripe Payments)
* Delivery Tracking (hardcoded calculations)
* Notification System (Order Updates via Kafka,user login and signup, delivery status)
* Service Discovery and API Gateway for Communication

### 1.2 Non-Functional Requirements

* Scalability: the system must support concurrent usage by thousands of customers, especially peak sale periods.
* Security: user data or personal data should be encrypted and handled securely using authentication mechanisms.
* Performance: api reponse should be within 300ms under average load
* Availability: application should have uptime with redundancy across critical services not less then 96%
* Maintainability: each funcanality and service should be independent to each other.

## 2. Users and Use Cases

### 2.1 User Types

* **Customers:** Browse products, select and add to cart,confirm order, make payments, and track orders.
* **Admin:**delete and add products , update products, delete users, check list of users, check list of users, check role based users, edit stock of products after purchase.
* **Delivery Personnel:** Update delivery status and track orders.

### 

### 2.2 Use Case Diagram

### 

### 2.2 Use Case Diagram

### Use Case List of features:

|  |  |  |
| --- | --- | --- |
| **Feature ID** | **Feature Name** | **Description** |
| 1 | User Authentication | Users can register, log in, and manage profiles securely. |
| 2 | Product Management | Admins can add, update, and delete products. Users can view products. |
| 3 | Shopping Cart | Users can add, remove, and update items in the cart. |
| 4 | Order Processing | Users can place and track orders. Admins can manage orders. |
| 5 | Payment Integration | Secure payment processing for orders. |
| 6 | Delivery Tracking | Users and delivery personnel can track order status. |
| 7 | Notification System | Kafka-based notifications for order updates. |
| 8 | Service Discovery | Eureka-based service registry for microservices communication. |

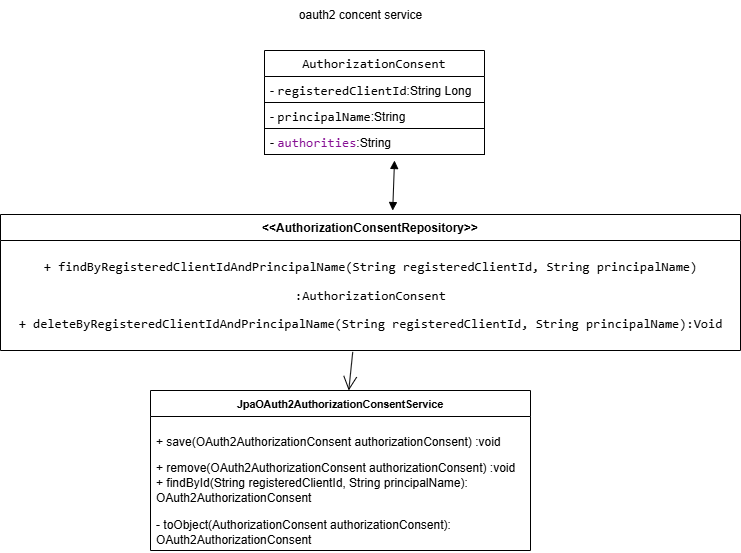
Table 2.2: **Feature List**

# 3. Class Diagrams

# 3.1 User Service

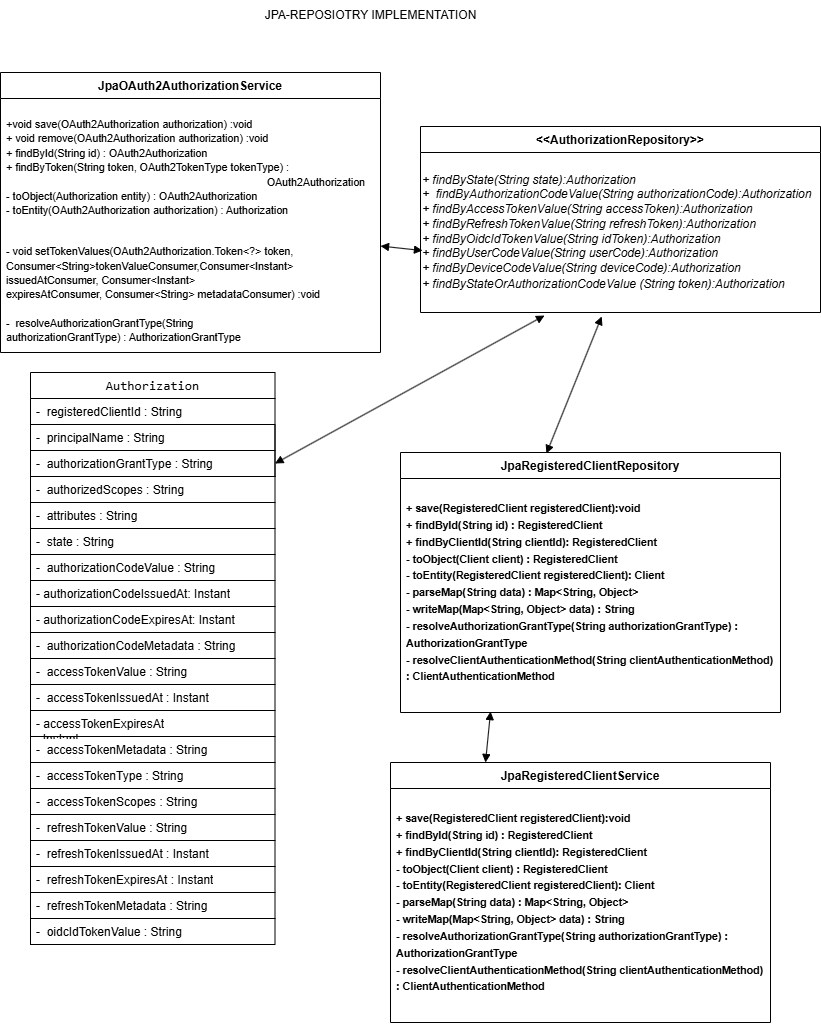
## User service implementation Fig 3.1

3.2 User service Authorisation server concent service



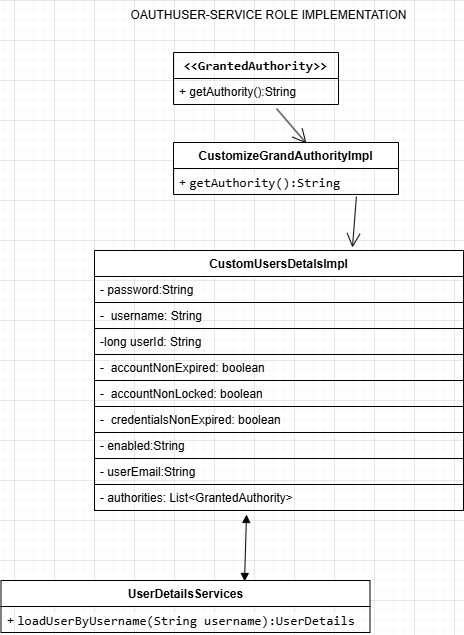
User service Authorization server consent service Fig 3.2

3.3 JpaAuthorisation service



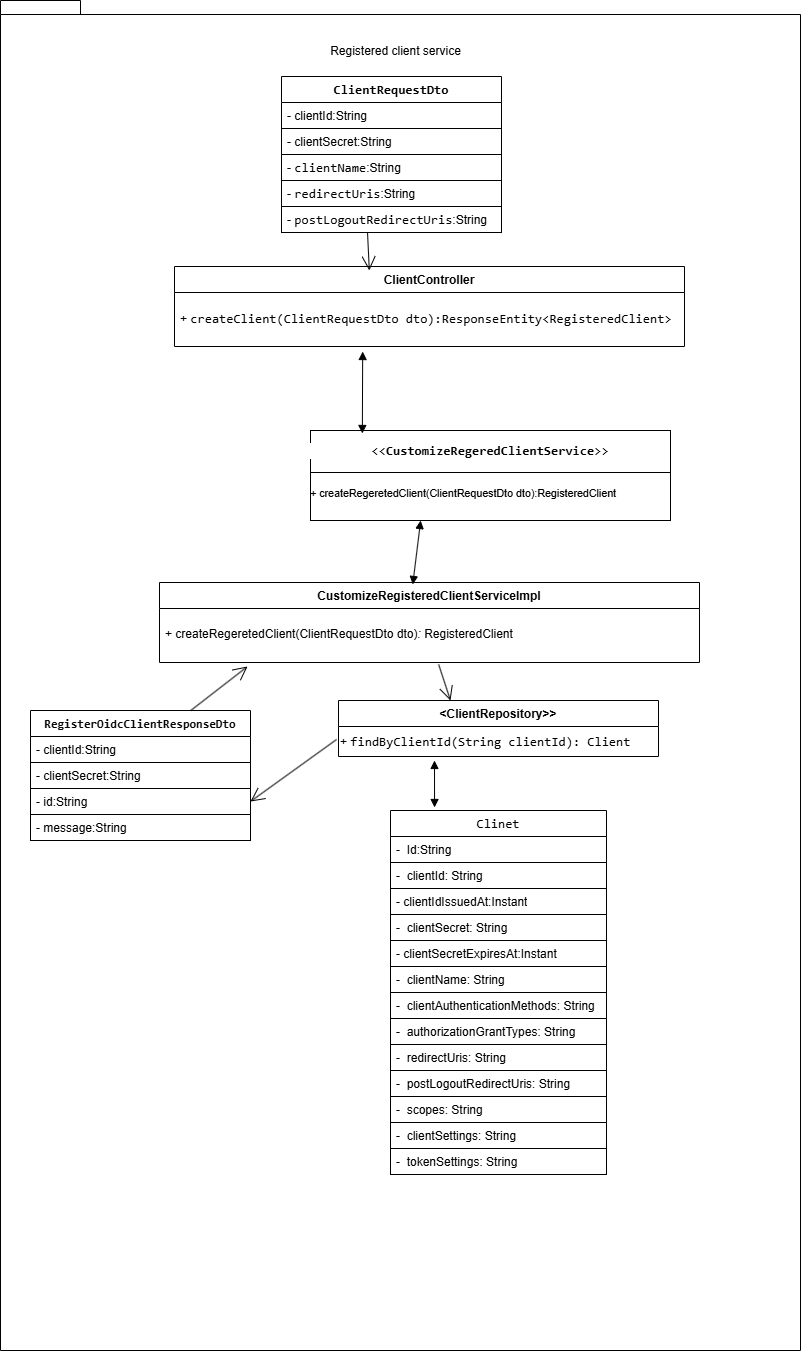
JpaAuthorisation service Fig.3.4

## 3.5 Role Authority Service



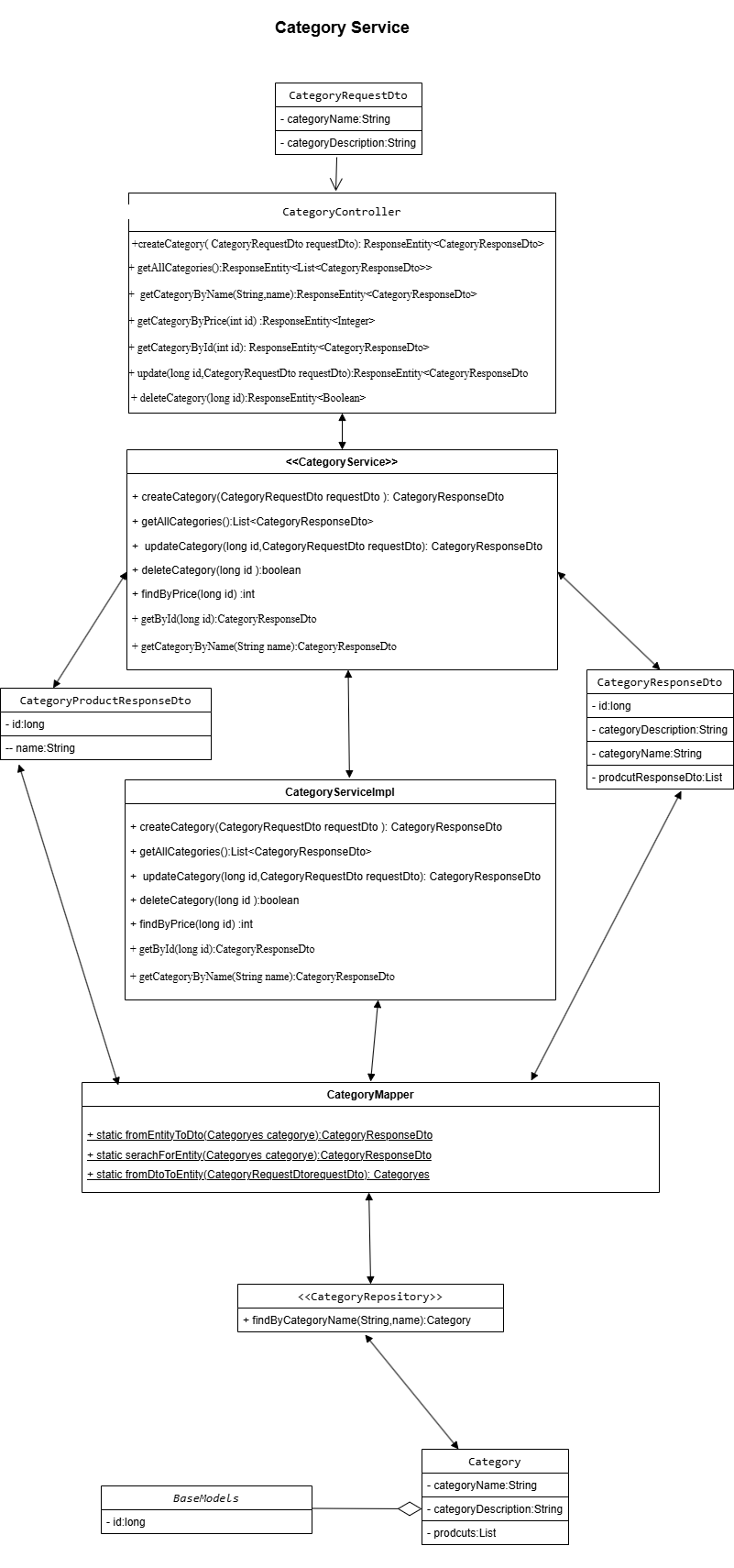
Role Authority Service Fig .3.5

## 3.6 Authrisation server Registered clinet or OICD client



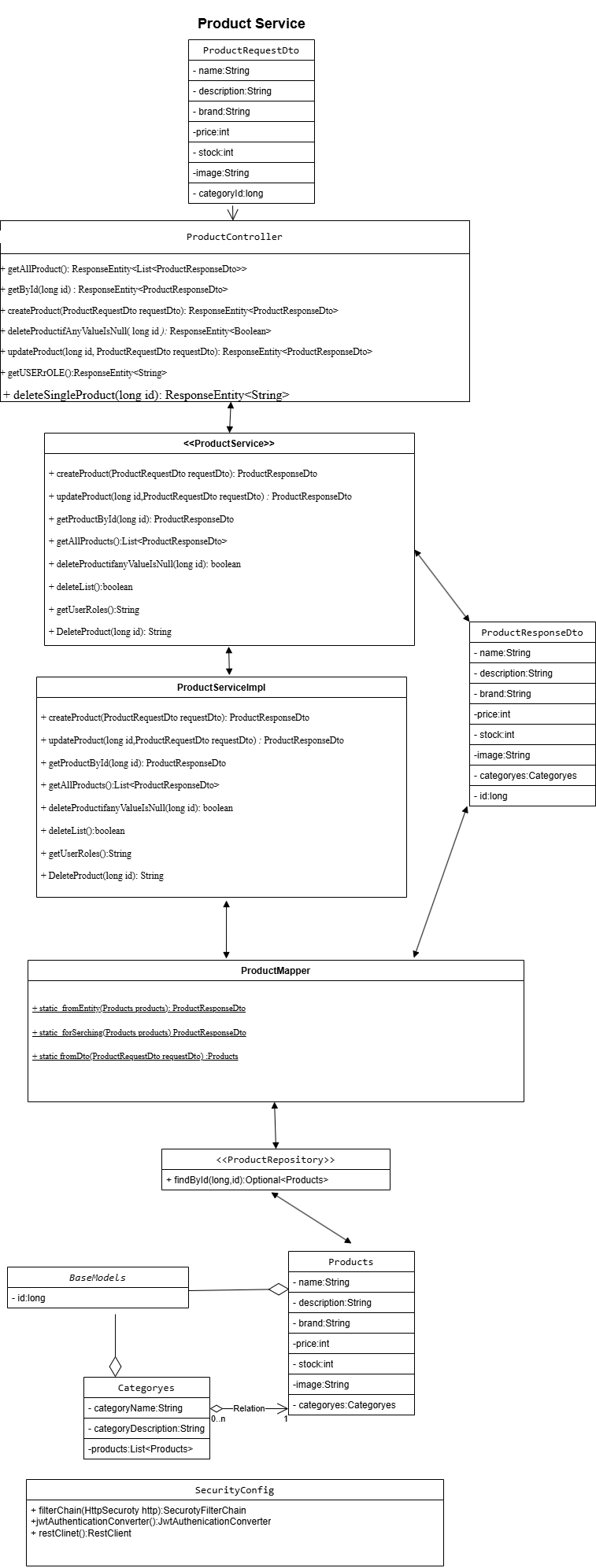
Authrisation server Registered clinet or OICD client Fig. 3.6

## 3.7 Product-service (Category-service)



## Product-service (Category-service) Fig 3.7

## 3.8 Product service

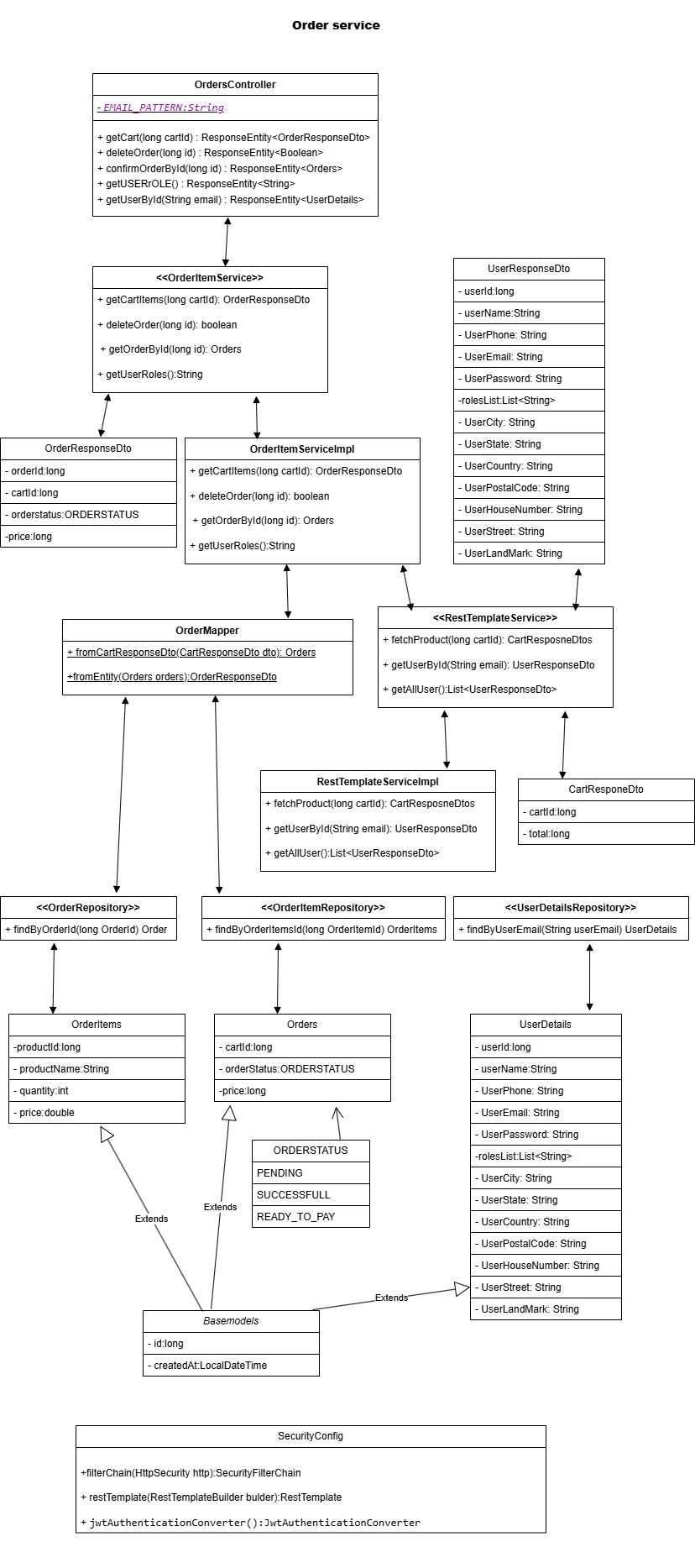


Product service Fig 3.8

## 3.9 Cart service

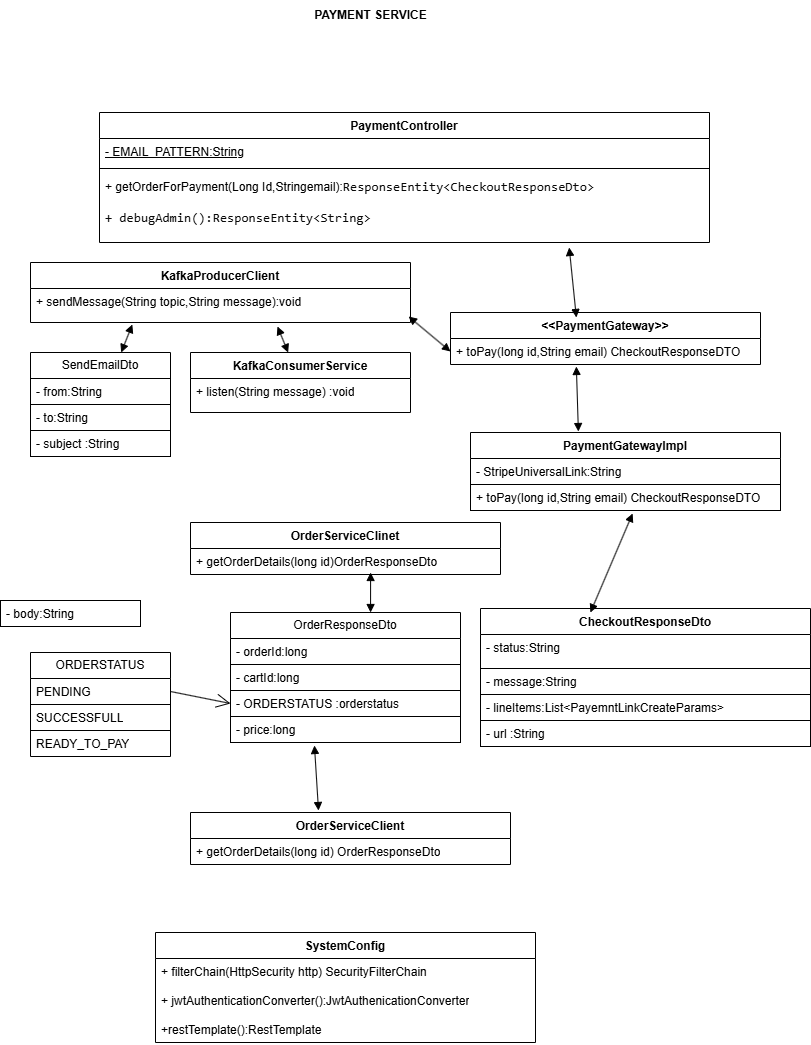
## 

## Cart Service Fig 3.9

3.10 Order service

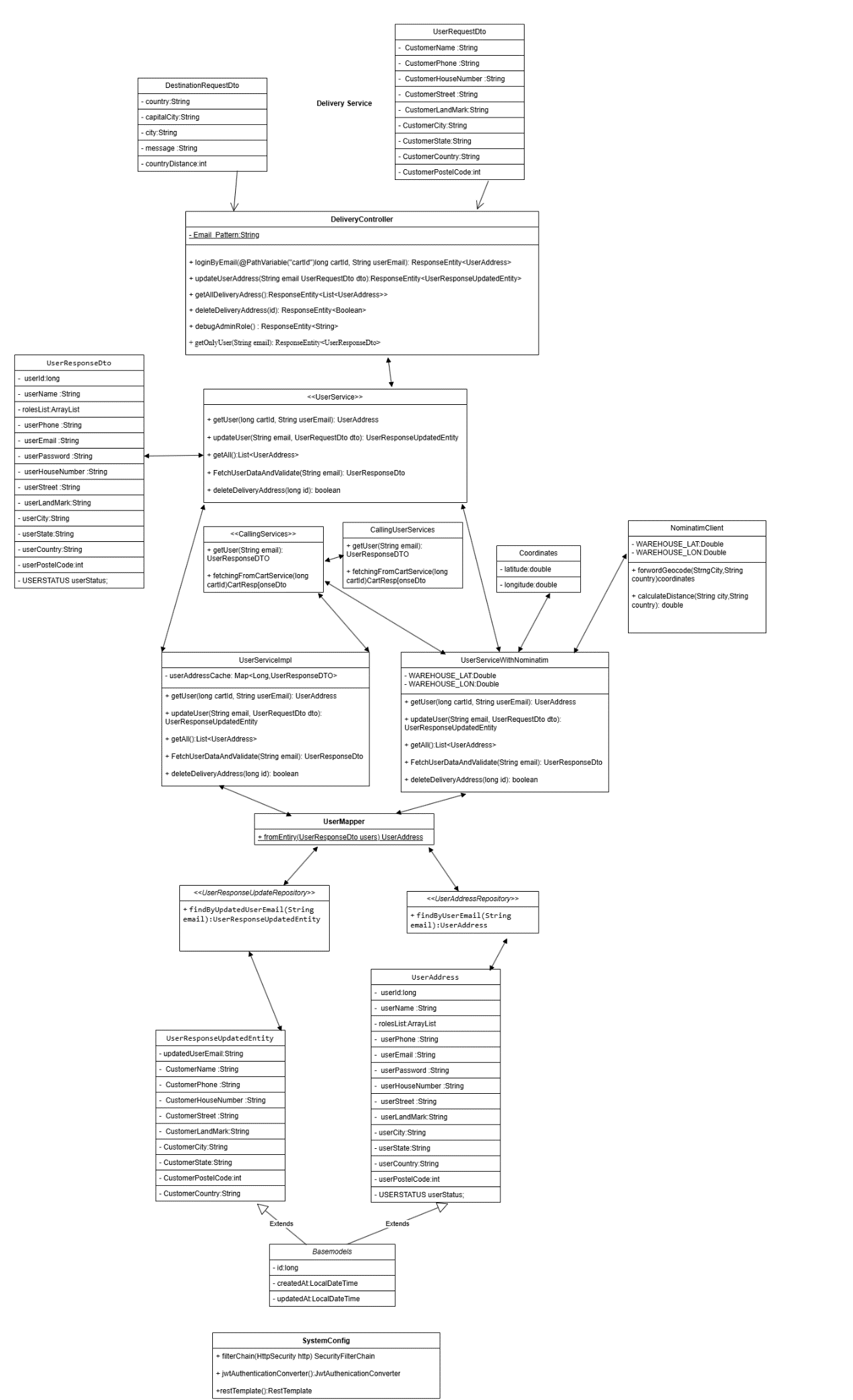
Order service Fig 3.10

## 3.11 Payment service



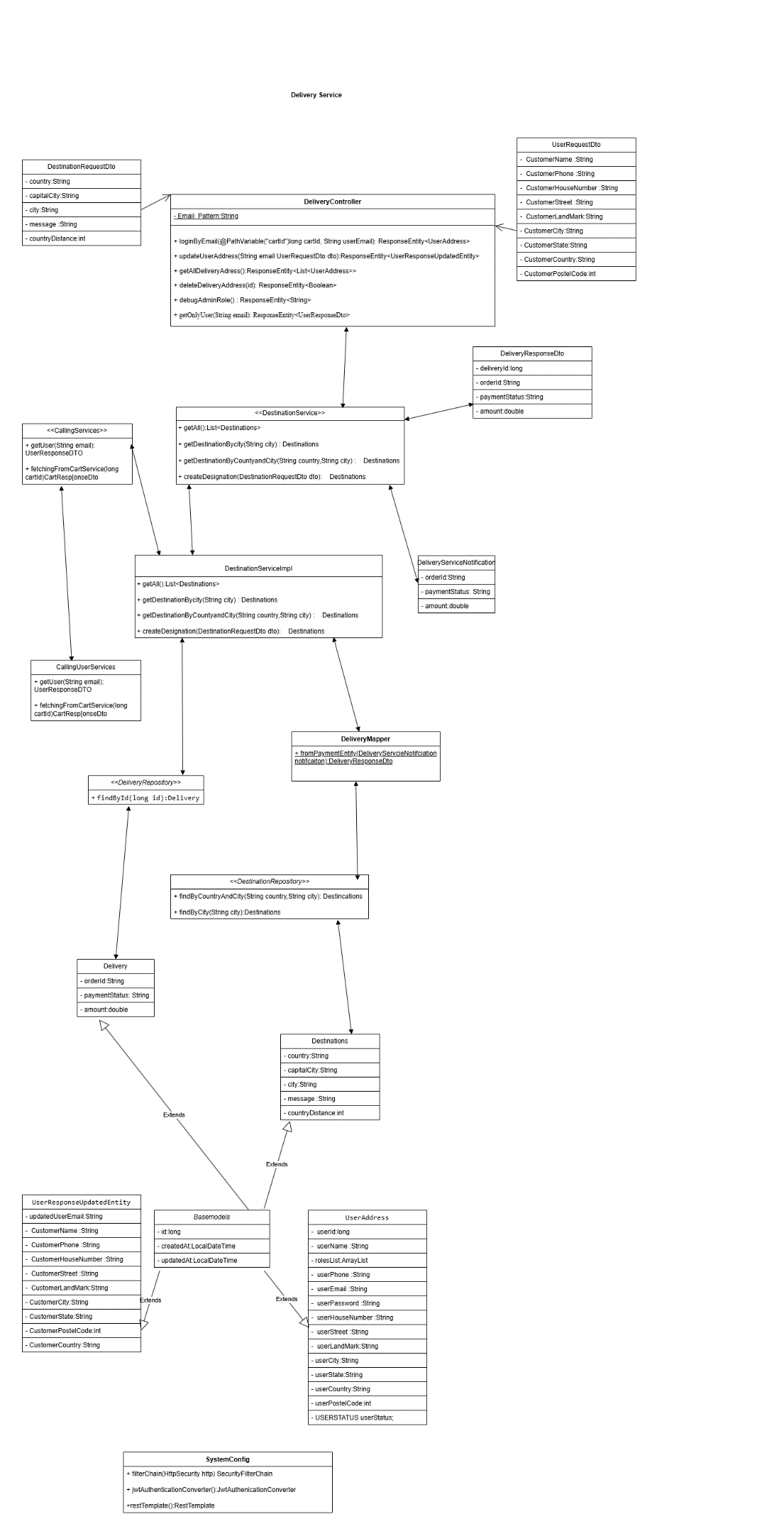
Payment service Fig 3.11

## 3.12 Delivery service



Delivery Service Fig 3.12

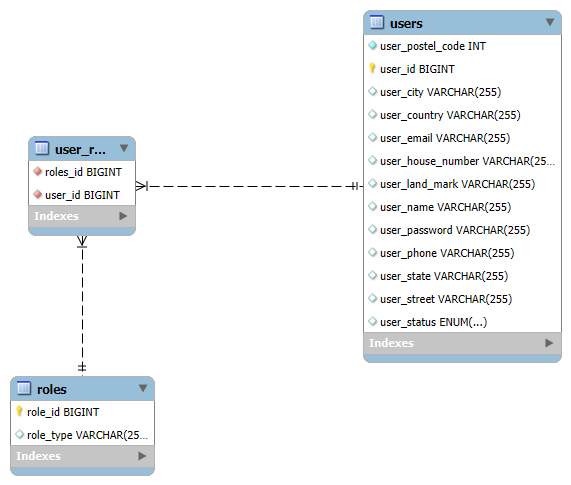
## 3.13 Delivery service (user implementation)



Delivery Service Fig 3.13

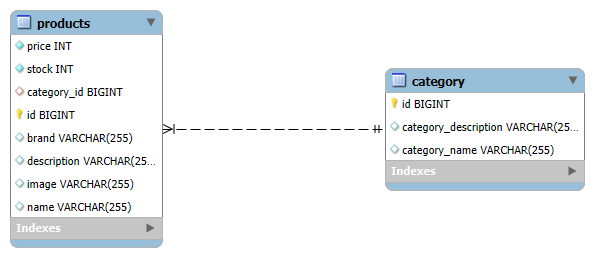
1. Database Schema Design

## 4.1 User service schema diagram



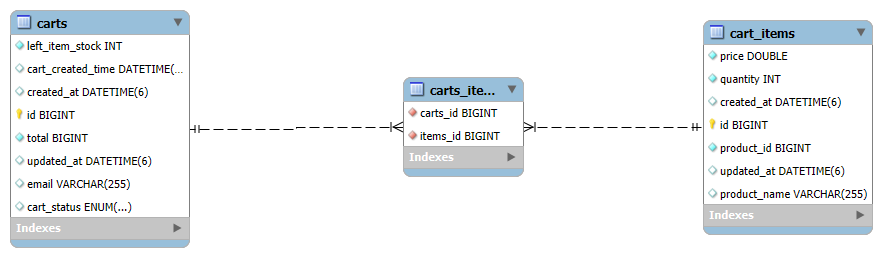
## User service schema diagram Fig 4.1

## 4.2 Product service



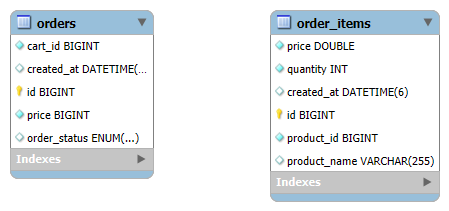
Product service Fig 4.2

## 4.3 Cart service



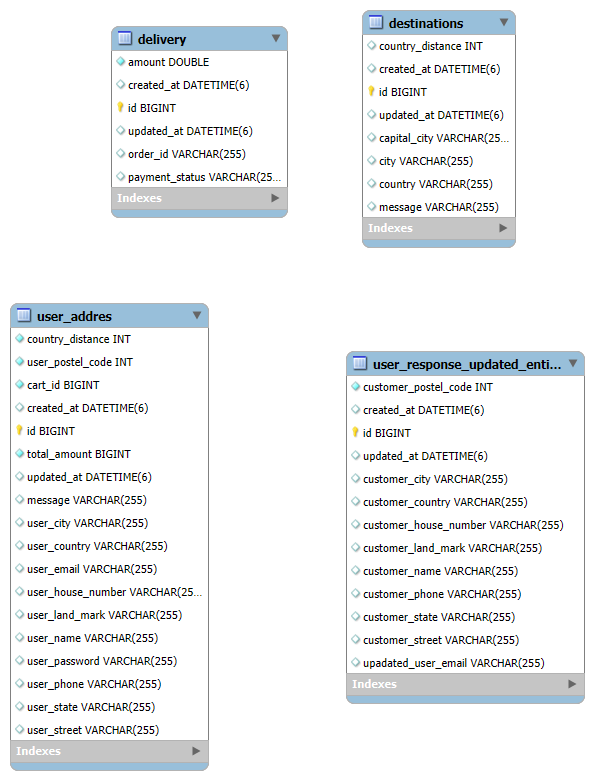
Cart service Fig 4.3

## 4.4 Order service



Order service Fig 4.4

## 4.5 Delivery service



## Delivery service Fig 4.5

# Textual schema design

# User Service textual schema

Roles

* role\_Id (PK, auto\_increment)
* roleType

Users

* user\_Id(PK, auto\_increment)
* user\_Name
* user\_Phone
* user\_Email
* user\_Password
* user\_HouseNumber
* user\_Street
* user\_LandMark
* user\_City
* user\_State
* user\_Country
* user\_PostelCode
* user\_Status (Enum:LOGIN,LOGOUT)

user\_role

* user\_id(FK -> Users.user\_id)
* roles\_id(FK->Roles.role\_id)
* Composite Primary Key (user\_id,roles\_id)

**Between users and role : m:n**

# Product Service textual schema

Categoryes

* Id (PK, inherited from Basemodels)
* Category\_name
* Category\_description

Products

* Id (PK,inherited from BaseModels)
* Name
* Description
* Brand
* Price
* Stock
* Image
* Category\_id(FK -> Categoryes.id)

Cardinalities

* Cardinality to Products (1:m)
* Products to categoryes (m:1)

# Cart Service textual schema

CartItems

* Id (pk,Inherited from BaseModels)
* Product\_id
* Product\_name
* Quantity
* Price
* Created\_at (Inherited from BaseModels)
* Updated\_at(Inherited from BaseModels)

Cart

* Id (PK, Inherited from BaseModels)
* Email
* Total
* Cart\_created\_at
* Left\_item\_stock
* Cart\_status(Enum: IN\_PROGRESS, FORWORD\_TO\_ORDER\_SERVICE, DELETED, ACCEPTED, UPDATED)
* created\_at(Inherited from BaseModels)
* updated\_at(Inherited from BaseModels)

Cartitems relationship

Cart to cartItem (1:m)

# Order Service textual schema

OrderItems

* Id (PK, inherited from Basemodel)
* Product\_id
* Product\_name
* Quantity
* Price
* Created\_at

Orders

* Id (PK, inherited from Basemodel)
* cart\_id
* order\_status(Enum: PENDING,SUCCESSFUL, READY\_TO\_PAY )
* Price
* Created\_at

UserDetails

* id (PK, inherited from BaseModels)
* user\_id
* user\_name
* user\_phone
* user\_email
* user\_password
* roles\_list
* user\_city
* user\_state
* user\_country
* user\_postal\_code
* user\_house\_number
* user\_street
* user\_landmark
* created\_at

# Delivery Service textual schema

Delivery

* Id(PK, inherited from Basemodel)
* Order\_id
* Payment\_status
* Amount
* Created\_at
* Updated\_at

Destinations

* id (PK, inherited from BaseModels)
* country
* capital\_city
* city
* message
* country\_distance
* created\_at
* updated\_at

UserAddress

* id (PK, inherited from BaseModels)
* user\_name
* user\_phone
* user\_email
* user\_password
* user\_house\_number
* user\_street
* user\_land\_mark
* user\_city
* user\_state
* user\_country
* user\_postal\_code
* message
* country\_distance
* cart\_id (Reference to the cart – optional FK)

• total\_amount (Amount to be paid)

• created\_at

• updated\_at

UserReponseUpdatedEntity

* id (PK, inherited from BaseModels)
* updated\_user\_email
* customer\_name
* customer\_phone
* customer\_house\_number
* customer\_street
* customer\_land\_mark
* customer\_city
* customer\_state
* customer\_country
* customer\_postal\_code
* created\_at
* updated\_at

Explain the **Low Level Design** of the Project in more detail by providing the **database schema**

**design** description

Provide the schema design textually as well as diagrammatically

Sample Schema Design described textually -

“”

Tables:

Batches

* Batch\_id
* Name
* Start\_month
* Current\_instructor
* Batch\_type\_id
* Primary Key(Batch\_id)

Students

* student\_id
* name
* graduation\_year
* University\_name
* email
* Phone\_number
* batch\_id
* Buddy\_id
* Primary Key(student\_id)

Classes

* Class\_id
* Name
* Date
* Time
* Instructor
* Primary Key(Class\_id)

Mentors

* Mentor\_id
* Name
* Current\_company
* Primary Key(Mentor\_id)

Mentor\_Sessions

* mentor\_session\_id
* time
* Duration
* Student\_id
* Mentor\_id
* Student\_rating
* Mentor\_rating
  + Primary Key(mentor\_session\_id)

Batches\_Classes

* Batch\_id
* Class\_id
* Primary Key(Batch\_id, Class\_id)

Student\_batch\_history

* student\_id
* batch\_id
* Shift\_date
* Primary Key(student\_id, batch\_id)

Batch\_type

* Batch\_type\_id
* Batch\_type
* Primary Key(Batch\_type\_id)

**Foreign Keys:**

* Batches(batch\_type\_id) refers  Batch\_type(batch\_type\_id)
* Students(batch\_id) refers  Batches(batch\_id)
* Mentor\_Sessions(Student\_id) refers Students(Student\_id)
* Mentor\_Sessions(Mentor\_id) refers Mentors(Mentor\_id)
* Batches\_Classes(Batch\_id) refers  Batches(batch\_id)
* Batches\_Classes(student\_id) refers Students(Student\_id)
* Student\_batch\_history(student\_id) refers Students(Student\_id)
* Student\_batch\_history(batch\_id) refers  Batches(batch\_id)

**Cardinality of Relations:**

* Between Batches and Batch\_type -> m:1
* Between Students and Batches -> m:1
* Between Batches and Classes -> m:m

“”

**Textual Schema Design**

**1.** Users Table

This table stores information about the users who interact with the e-commerce system.

* **user\_id** (Primary Key): Unique identifier for the user (auto-incremented).
* **username**: Username of the user.
* **password**: Encrypted password for authentication.
* **email**: Email address of the user.
* **first\_name**: First name of the user.
* **last\_name**: Last name of the user.
* **created\_at**: Date when the user was created.
* **updated\_at**: Date when the user information was last updated.

**2.** Products Table

This table stores information about the products available in the e-commerce system.

* **product\_id** (Primary Key): Unique identifier for the product.
* **name**: Name of the product.
* **description**: Description of the product.
* **price**: Price of the product.
* **stock\_quantity**: The number of items available in stock.
* **created\_at**: Date when the product was added.
* **updated\_at**: Date when the product was last updated.

**3.** Categories Table

This table stores information about product categories.

* **category\_id** (Primary Key): Unique identifier for the category.
* **name**: Name of the category.
* **description**: Description of the category.

**4.** Orders Table

This table stores information about orders placed by users.

* **order\_id** (Primary Key): Unique identifier for the order.
* **user\_id** (Foreign Key): References the user who placed the order.
* **order\_date**: Date when the order was placed.
* **status**: Status of the order (e.g., Pending, Shipped, Delivered).

**5. Order\_Items Table**

This table stores information about products in each order. It allows tracking of which products are part of which orders.

* **order\_item\_id** (Primary Key): Unique identifier for the order item.
* **order\_id** (Foreign Key): References the order to which this item belongs.
* **product\_id** (Foreign Key): References the product in the order.
* **quantity**: Number of items of the product ordered.
* **price**: Price of the product at the time of order.

**6. Payments Table**

This table stores payment details for the orders.

* **payment\_id** (Primary Key): Unique identifier for the payment.
* **order\_id** (Foreign Key): References the order associated with the payment.
* **payment\_date**: Date when the payment was made.
* **amount**: Amount paid by the user.
* **payment\_method**: Payment method used (e.g., Credit Card, PayPal).
* **payment\_status**: Status of the payment (e.g., Successful, Failed).

**7. Shipping Table**

This table stores shipping details for orders.

* **shipping\_id** (Primary Key): Unique identifier for the shipping record.
* **order\_id** (Foreign Key): References the order associated with the shipping.
* **address**: Shipping address for the order.
* **shipping\_date**: Date when the order was shipped.
* **delivery\_date**: Estimated or actual delivery date.
* **shipping\_status**: Status of the shipping (e.g., Shipped, In Transit, Delivered).

**Relationships and Foreign Keys:**

* **Users** -> **Orders**: One-to-many (A user can place multiple orders).
* **Orders** -> **Order\_Items**: One-to-many (An order can contain multiple products).
* **Products** -> **Order\_Items**: Many-to-many (Products can appear in multiple orders).
* **Orders** -> **Payments**: One-to-one (Each order has one payment).
* **Orders** -> **Shipping**: One-to-one (Each order has one shipping record).
* **Categories** -> **Products**: One-to-many (A category can have multiple products).

### ****Explanation of the Diagram:****

1. **Users Table**:
   * Contains user information and is related to Orders via the user\_id foreign key.
2. **Orders Table**:
   * Represents an order and links to Order\_Items, Payments, and Shipping tables through foreign keys.
3. **Order\_Items Table**:
   * Links orders to products, with multiple products allowed per order.
4. **Products Table**:
   * Contains product details, linked to categories and order items.
5. **Payments Table**:
   * Stores payment information linked to orders, with attributes like payment\_method and payment\_status.
6. **Shipping Table**:
   * Represents shipping details related to the order, with fields for address, shipping\_date, and shipping\_status.
7. **Categories Table**:
   * Represents product categories, and products belong to these categories through a foreign key reference.

### ****Cardinality and Relationships:****

* **Users** ↔ **Orders**: One-to-many (One user can place many orders).
* **Orders** ↔ **Order\_Items**: One-to-many (An order can contain many items).
* **Products** ↔ **Order\_Items**: Many-to-many (A product can be part of multiple orders).
* **Orders** ↔ **Payments**: One-to-one (Each order has one payment).
* **Orders** ↔ **Shipping**: One-to-one (Each order has one shipping record).
* **Categories** ↔ **Products**: One-to-many (Each category can have multiple products).

# Feature Development Process

topics = "sendemail", groupId = "email-service"

must have javamail app email and password and set it to environments

**Feature Development Process: Implementing the 'Place Order' Feature in an E-Commerce System**

In an **E-Commerce** system, one of the key features is the **"Place Order"** functionality. This feature is critical for processing a customer's request to purchase items, handling payment, updating stock, and preparing for shipping.

{

"userId": 12345,

"orderItems": [

{ "productId": 56789,

"quantity": 2},

{ "productId": 98765,

"quantity": 1}

],

"paymentDetails": {

"paymentMethod": "CreditCard",

"paymentAmount": 150.00

},

"shippingAddress": "1234 Elm Street, City, State, ZIP"

}

* userId: Unique identifier for the user placing the order.
* orderItems: List of products with productId and quantity being purchased.
* paymentDetails: Payment method and amount.
* shippingAddress: The address where the order will be shipped.

## **b. Service Which Picks the Request**

Once the API receives the request, it is passed to the **OrderService** for processing. The service orchestrates the various tasks involved in placing an order, such as:

* Validating product availability.
* Calculating the total price.
* Creating an order record in the database.
* Handling the payment process.
* Managing inventory and reducing stock.
* Creating shipping information.

Pick One key feature - Talk about its development process, implementation and performance optimisation / metric optimisation achieved…

For example, ‘Book a seat’ feature in developing ‘BookMyShow’ app

Elaborate the request flow to backend

1. API Request Payload
2. Service which picks the request
3. Flow of MVC architecture

Explain the performance improvement / metric optimization achieved.

For example,

* Used Cache to reduce API Response time by X seconds…
* Optimized Query Response time by using Indexing…

Benchmarking of response time without the optimisation and post the optimisation

Deployment Flow

The deployment flow of an **E-Commerce** system involves the end-to-end process of releasing the application from the development environment to a production environment. This includes steps for building the application, testing, deploying, and monitoring. The deployment flow is generally automated through a **Continuous Integration (CI)** and **Continuous Deployment (CD)** pipeline.

Explain how the deployment will work via AWS (Describe the below) -

* EC2
* VPC
* Security Groups
* RDS
* Cache
* Managed Infra / Elastic Beanstalk

Use diagrams, images to explain better

### ****1. Code Commit and Version Control****

* **Source Control**: Developers commit code to a version control system like **Git** (e.g., GitHub, GitLab, or Bitbucket).
* **Branching**: The deployment process typically works with **feature branches** that are merged into the **main branch** once the feature is complete and tested.

### ****2. Continuous Integration (CI)****

Once the code is pushed to the version control system, the **CI pipeline** is triggered. This step ensures that the code is compiled, tested, and packaged.

#### **CI Steps:**

1. **Build**:
   * The code is checked out, and the **build** process is started using a build tool like **Maven** or **Gradle**.
   * This process compiles the code, resolves dependencies, and creates an executable package (e.g., .jar file for a Spring Boot app).
2. **Unit and Integration Testing**:
   * **Unit tests** and **integration tests** are executed to ensure that individual components and their interactions work as expected.
   * A tool like **JUnit** or **TestNG** is used to run the tests.
   * Code coverage reports are generated to ensure all parts of the application are tested.
3. **Static Code Analysis**:
   * Tools like **SonarQube** are used to analyze the code for quality issues, such as bugs, security vulnerabilities, and code smells.
   * If any issues are found, the build fails, and the developer is required to fix them.
4. **Artifact Creation**:
   * After passing all tests and static code analysis, an artifact (e.g., .jar, .war, .docker image) is generated.
   * This artifact is stored in an artifact repository such as **Nexus** or **Artifactory**.

### ****3. Continuous Deployment (CD)****

The **CD pipeline** automates the deployment process to different environments, from staging to production.

#### **CD Steps:**

1. **Staging Deployment**:
   * The application is deployed to a **staging environment**, which is a replica of the production environment.
   * The **staging environment** uses the same infrastructure as production, including the same database, caching, and messaging systems.
   * Deployment tools like **Kubernetes**, **Docker Swarm**, or **Ansible** are used to automate the deployment process.
2. **Smoke Testing**:
   * Once the application is deployed to staging, **smoke tests** are performed to ensure that the application is running as expected (e.g., testing the basic functionality like logging in, placing an order, etc.).
3. **Performance Testing**:
   * Load testing tools like **JMeter** or **Gatling** are used to simulate real-world traffic and check if the application performs well under load.
   * Performance metrics such as **response time**, **throughput**, and **error rates** are monitored to ensure that the application is scalable.
4. **Approval Gate**:
   * After smoke and performance tests are successful, an approval gate is typically set up for manual review. A developer or QA team member verifies the changes before moving to production.
5. **Production Deployment**:
   * Once the changes are approved, the application is deployed to the **production environment**. This is usually done with **zero-downtime** deployment strategies like **blue/green deployment** or **canary releases** to minimize disruptions to users.
   * The application is deployed, and a **load balancer** ensures traffic is directed to the correct application instance.

### ****4. Post-Deployment****

Once the application is deployed to production, the system needs to be monitored and maintained.

#### **Post-Deployment Steps:**

1. **Monitoring and Logging**:
   * **Application monitoring** tools like **Prometheus**, **Grafana**, or **New Relic** are used to track the health of the application.
   * **Logs** from the application are aggregated using tools like **ELK Stack** (Elasticsearch, Logstash, Kibana) or **Splunk** to check for errors, performance issues, or unusual behavior.
2. **Error Tracking**:
   * **Sentry** or **Rollbar** can be used to automatically track and alert developers of any errors that occur in the production environment.
3. **Auto-Scaling**:
   * If the application is deployed in **Kubernetes** or **AWS**, auto-scaling policies are configured to handle traffic spikes and ensure the system scales up or down depending on demand.
4. **Continuous Monitoring and Feedback**:
   * After deployment, continuous feedback is obtained through **user feedback** or **user experience monitoring** tools (e.g., **Hotjar**, **Google Analytics**) to understand how users interact with the system and identify potential issues.

### ****5. Rollback Strategy****

In case of issues in production, having a **rollback strategy** is crucial.

#### **Rollback Steps:**

1. **Automatic Rollback**:
   * Some CI/CD pipelines are configured to automatically rollback the deployment in case of failure, restoring the previous stable version of the application.
2. **Manual Rollback**:
   * If there are critical issues, the development team can manually trigger a rollback to the previous stable build using **Docker images** or **Kubernetes rollbacks**.

### ****6. Deployment Diagram****

Here’s a **diagrammatic representation** of the deployment flow:

pgsql

CopyEdit

+-----------------------+

| Developer Pushes Code |

+-----------------------+

|

v

+----------------------+

| CI Pipeline Triggered|

+----------------------+

|

v

+--------------------------+

| Build, Test & Package App |

+--------------------------+

|

v

+------------------------+

| Deploy to Staging Env |

+------------------------+

|

v

+-------------------------+

| Smoke & Performance Test |

+-------------------------+

|

v

+--------------------------+

| Approval Gate (Manual) |

+--------------------------+

|

v

+----------------------------+

| Deploy to Production Env |

+----------------------------+

|

v

+--------------------------+

| Monitoring & Logging |

+--------------------------+

|

v

+------------------------+

| Continuous Feedback |

+------------------------+

|

v

+---------------------------+

| Rollback (If Issues Found)|

+---------------------------+

### ****Summary of the Deployment Flow****

1. **CI Pipeline**:
   * Code is committed, built, tested, and packaged into an artifact.
2. **CD Pipeline**:
   * The artifact is deployed to **staging**, undergoes testing, and is then deployed to **production**.
3. **Post-Deployment**:
   * Continuous monitoring ensures that the application remains healthy. Errors are logged, and performance is tracked.
4. **Rollback Strategy**:
   * If issues occur, the system can automatically or manually roll back to a previous stable version.

By automating the deployment and monitoring processes, this flow ensures that new features, updates, and bug fixes are delivered efficiently with minimal downtime or disruption to the user experience.

Technologies Used

Kafka, MySQL, Springboot, Cloud etc…

* For each key technology used in building the project,
  + Detail and describe each of them
  + Elaborate how they can be used in real life
  + Provide example of real-life applications using them

Use diagrams, images to explain better

**Tip** - Use the internet to improve your project but DO NOT PLAGIARIZE - Include proper references if you are quoting articles from the internet

In this project, several technologies were utilized to ensure scalability, performance, and smooth functionality across microservices, databases, and other integrated services. Below, I will provide detailed descriptions of each key technology used, explain their real-life use cases, and provide examples of real-world applications where these technologies are used.

### ****1. Kafka - Distributed Streaming Platform****

#### **Overview**

**Apache Kafka** is an open-source, distributed event streaming platform used to build real-time data pipelines and streaming applications. It’s designed for high throughput, fault tolerance, and scalability.

#### **Use Case in the Project**

In the project, **Kafka** is used for handling high-throughput, real-time event streaming between different microservices, such as processing user actions, order placements, and notifications in the e-commerce system.

#### **Real-Life Use Case**

Kafka is widely used in scenarios where real-time data processing is critical. Some common real-life applications include:

* **Real-Time Analytics**: Companies like **Netflix**, **Uber**, and **LinkedIn** use Kafka for real-time user analytics and recommendation engines.
* **Log Aggregation**: Kafka is used for collecting logs from various services and making them available for real-time processing and monitoring.

#### **Example**

* **Netflix**: Netflix uses Kafka to stream millions of events per second, such as user actions (like play, pause, skip) and service metrics, to monitor service health and personalize recommendations in real-time.

### ****2. MySQL - Relational Database Management System (RDBMS)****

#### **Overview**

**MySQL** is one of the most popular open-source relational database management systems (RDBMS). It stores data in tables and supports SQL queries for data retrieval, modification, and deletion.

#### **Use Case in the Project**

In the project, **MySQL** is used to store structured data such as user profiles, orders, products, and transactions. MySQL ensures consistency and enables complex queries that aggregate data from different tables.

#### **Real-Life Use Case**

MySQL is commonly used in applications where structured data storage is required:

* **E-Commerce Systems**: **Amazon** and **eBay** use MySQL to store transactional data and user profiles.
* **Content Management Systems (CMS)**: Platforms like **WordPress** and **Joomla** use MySQL to store blog posts, user information, and metadata.

#### **Example**

* **Airbnb**: Airbnb uses MySQL as the primary database for storing listings, bookings, user data, and payments, ensuring high availability and consistency across its global platform.

### ****3. Spring Boot - Java Framework for Building Microservices****

#### **Overview**

**Spring Boot** is a popular Java framework used to create stand-alone, production-grade Spring-based applications. It simplifies the development of microservices by providing built-in configuration, embedded servers, and other utilities.

#### **Use Case in the Project**

In the project, **Spring Boot** is used to create various microservices like user service, product service, order service, and payment service. It provides REST APIs, manages dependencies, and simplifies configurations, making it easier to deploy and scale services independently.

#### **Real-Life Use Case**

Spring Boot is used in real-world applications to quickly build scalable and efficient web applications and microservices:

* **E-Commerce**: **Alibaba** and **Zalando** use Spring Boot to develop and scale their microservices.
* **Banking Applications**: **JPMorgan Chase** uses Spring Boot for developing banking applications with highly available APIs.

#### **Example**

* **Spotify**: Spotify uses Spring Boot to build its backend microservices that handle music streaming, user data, playlists, and recommendations.

### ****4. Cloud (AWS, Azure, Google Cloud)****

#### **Overview**

Cloud platforms like **AWS**, **Azure**, and **Google Cloud** provide scalable infrastructure, storage, and services that help deploy and manage applications efficiently. They provide solutions for computing, networking, and data storage.

#### **Use Case in the Project**

In the project, **AWS** (Amazon Web Services) is used for deploying microservices and databases, ensuring high availability, security, and scalability. Key services include EC2 for hosting services, RDS for MySQL databases, and S3 for file storage.

#### **Real-Life Use Case**

Cloud platforms are widely used in various industries to reduce operational costs and improve performance and scalability:

* **Netflix** uses **AWS** to manage its global infrastructure and deliver content to millions of users.
* **Spotify** uses **Google Cloud** for handling the large amount of music data, user interactions, and streaming content.

#### **Example**

* **Airbnb**: Airbnb uses AWS for hosting its website, running microservices, and scaling its infrastructure based on demand.

### ****5. Docker - Containerization****

#### **Overview**

**Docker** is an open-source platform used to automate the deployment of applications inside lightweight, portable containers. Containers allow applications to run in isolated environments, ensuring that dependencies do not conflict across different environments.

#### **Use Case in the Project**

In the project, **Docker** is used to containerize each microservice, making it easy to deploy them across multiple environments (development, staging, production) and ensuring consistency across services.

#### **Real-Life Use Case**

Docker is commonly used in:

* **Microservices Architecture**: Companies like **Uber** and **GitLab** use Docker to containerize their microservices for efficient management and scaling.
* **Continuous Integration**: Docker is used to create isolated testing environments for continuous testing and integration.

#### **Example**

* **Pinterest**: Pinterest uses Docker to containerize microservices, ensuring a consistent deployment process across their infrastructure.

### ****6. Redis - In-Memory Data Store****

#### **Overview**

**Redis** is an open-source, in-memory data store often used for caching and real-time data storage. It supports various data structures like strings, lists, sets, and hashes, providing high-performance read and write operations.

#### **Use Case in the Project**

In the project, **Redis** is used for caching frequently accessed data such as product details and user sessions, reducing the load on the database and improving performance.

#### **Real-Life Use Case**

Redis is used in scenarios that require fast data retrieval:

* **Social Media**: **Twitter** and **Instagram** use Redis to cache feeds and user data for faster load times.
* **E-Commerce**: **eBay** and **Amazon** use Redis for caching product information, user sessions, and cart data.

#### **Example**

* **GitHub**: GitHub uses Redis for caching data like user information and code repositories, ensuring faster page load times for millions of users.

### ****7. Kubernetes - Container Orchestration****

#### **Overview**

**Kubernetes** is an open-source system for automating the deployment, scaling, and management of containerized applications. It helps to manage the lifecycle of containers across clusters of machines, ensuring high availability and scalability.

#### **Use Case in the Project**

In the project, **Kubernetes** is used to orchestrate the deployment of microservices, manage load balancing, and enable auto-scaling based on demand.

#### **Real-Life Use Case**

Kubernetes is used for large-scale containerized application management:

* **Airbnb** uses Kubernetes to manage and scale its microservices across multiple cloud environments.
* **Google** uses Kubernetes internally for managing containerized applications and services at scale.

#### **Example**

* **Dropbox**: Dropbox uses Kubernetes to scale their infrastructure, manage containerized applications, and ensure high availability.

### ****Conclusion****

The technologies mentioned above play crucial roles in ensuring the **scalability**, **efficiency**, and **performance** of modern software systems. By combining tools like **Kafka**, **MySQL**, **Spring Boot**, **Docker**, and **Kubernetes**, organizations can build reliable, distributed systems that can handle millions of requests with minimal latency. These tools are widely adopted across industries, including e-commerce, social media, banking, and entertainment, demonstrating their importance in real-world applications.

By understanding and leveraging these technologies, businesses can optimize their development process, improve system performance, and deliver better user experiences.

Conclusion

The Conclusion should include some key points as elaborated below -

* Key Takeaways: Highlight the important concepts and technologies learned from doing the Project
* Practical Applications : Significance of technologies with their real-world applications
* Limitations : Limitations of the technologies, cost implications and suggestions for improvement

### 

### ****Conclusion****

In this project, a variety of advanced technologies and methodologies were utilized to build a robust, scalable, and efficient microservices-based system. Below are some of the key takeaways, practical applications, and limitations of the technologies used in the project.

### ****Key Takeaways****

1. **Microservices Architecture**:
   * This project reinforced the importance of using a **microservices architecture** for large-scale applications. Each service is independent, which promotes flexibility, scalability, and ease of maintenance.
2. **Event Streaming with Kafka**:
   * The use of **Kafka** for event streaming demonstrated its ability to handle high-throughput and real-time data processing. Kafka allows microservices to communicate asynchronously, improving system responsiveness and fault tolerance.
3. **Relational Databases (MySQL)**:
   * **MySQL**'s use in managing structured data such as user details, transactions, and product information provided valuable experience with relational databases and their ability to enforce data integrity through ACID transactions.
4. **Cloud Infrastructure (AWS)**:
   * Utilizing **AWS** for deploying services and databases highlighted the power of cloud infrastructure. It also showcased the importance of cloud-native services in scaling applications as traffic increases.
5. **Containerization with Docker**:
   * **Docker** played a pivotal role in containerizing microservices, ensuring consistency across development, staging, and production environments, and simplifying the deployment pipeline.
6. **Performance Optimization**:
   * The project highlighted the importance of **caching** with **Redis** to optimize performance and reduce response time. By caching frequently accessed data, significant improvements in system efficiency were achieved.
7. **Orchestration with Kubernetes**:
   * **Kubernetes** allowed for the seamless orchestration and scaling of containerized microservices. It provided high availability and fault tolerance, ensuring that the system could scale as demand grew.

### ****Practical Applications****

1. **Real-Time Data Processing**:
   * Kafka, Redis, and MySQL are critical for building **real-time applications**, such as **e-commerce platforms**, **banking systems**, and **social media applications**, where quick data processing and low latency are essential.
2. **Scalable Web Applications**:
   * Technologies like **Spring Boot**, **Docker**, and **Kubernetes** are perfect for building **scalable web applications** that can grow as traffic increases. Platforms like **Netflix** and **Spotify** rely on such technologies to serve millions of concurrent users.
3. **Cloud-Native Applications**:
   * Using **AWS** for deploying services makes it easier to build **cloud-native applications** that can scale dynamically based on demand. This is crucial for companies such as **Airbnb** and **Uber**, where resource demand fluctuates throughout the day.
4. **Event-Driven Architectures**:
   * Kafka’s real-time event-driven architecture is used by companies like **LinkedIn** and **Uber** to process large volumes of events, such as user actions, in real-time.

### ****Limitations****

1. **Complexity of Microservices**:
   * While microservices provide great flexibility and scalability, managing multiple services can become complex. **Service discovery**, **communication between services**, and **data consistency** can be challenging, especially in a large-scale system.

**Suggestion**: To mitigate this, tools like **Spring Cloud** for service discovery and **Hystrix** for fault tolerance can be used to simplify the management of microservices.

1. **Cost Implications**:
   * Cloud-based services like **AWS** and **Kubernetes** incur ongoing costs based on resource consumption (compute, storage, data transfer). **MySQL** database clusters on the cloud can also become expensive at scale.

**Suggestion**: Optimizing resource usage, using cost-effective cloud services, and periodically reviewing the infrastructure costs can help reduce expenses.

1. **Caching Challenges**:
   * While **Redis** provides significant performance improvements, ensuring data consistency between the cache and the database can be complex. Outdated or inconsistent data in the cache can cause issues, especially in a highly dynamic system.

**Suggestion**: Using a cache expiration strategy and combining **Redis** with a **write-through or write-back cache** strategy could ensure data consistency while still benefiting from caching.

1. **Scalability of Kafka**:
   * Kafka can become challenging to scale beyond a certain point, particularly in environments with many topics and partitions. Kafka's management, such as balancing partitions and ensuring message delivery guarantees, becomes more complicated as the system grows.

**Suggestion**: **Monitoring Kafka’s performance** and using tools like **Kafka Streams** to manage real-time data more effectively can improve its scalability and performance.

### ****Final Thoughts****

This project provided valuable experience in integrating multiple advanced technologies like **Kafka**, **MySQL**, **Spring Boot**, **Docker**, and **Kubernetes** into a cohesive microservices-based architecture. The practical applications and limitations outlined above are critical for designing real-world systems. While these technologies offer substantial benefits in terms of scalability, performance, and flexibility, challenges such as complexity, cost, and data consistency need to be addressed effectively for optimal outcomes.

By carefully balancing the advantages and limitations of these technologies, businesses can build robust, scalable systems that meet the demands of modern, real-time, and distributed applications.

References

Include the websites or works or the list of works referred to in a text or consulted by you for writing this report

1. Name of the Website, Date and time of referring to the Website, Name of the Author, Title/Topic
2. Author Name, Title / Topic, Research Paper Name / Book Name, Year of Publication

First

Customer can check the category and product with any signup

1. Browse Categories

* EndPoint: GET <http://localhost:8080/category/>

Retrive available product categories and choose your preferred category from display list

1. Search products by category

* Endpoint: GET <http://localhost:8080/category/searchByCategoryName/>

Example:search for products in “PHONES” category.

Alternative you can fetch all products using endpoint

* Endpoint: GET http://localhost:8080/product/

1. Add Products to CART

* Endpoint: POST <http://localhost:8085/cart/add>

Add products to your cart by specifying the product ID and quantity

Example Request:

{

"item": [

{

"productId": 352,

"quantity": 5

},

{

"productId": 355,

"quantity": 4

}

]

}

Example response will be :

{

"userId": 2,

"items": [

{

"cartId": 3,

"productId": 352,

"productName": "IPHONE",

"quantity": 5,

"price": 1000.0

},

{

"cartId": 4,

"productId": 355,

"productName": "IPHONE",

"quantity": 4,

"price": 1800.0

}

],

"total": 12200,

"cartCreatedTime": "2024-12-27T19:12:51.1442883"

}

Above response will include products details and cartId, and in cart it will store total price and current time to request.

1. Confirm Cart:

* Endpoint : GET http://localhost:8085/cart/getCartById/{userId}

Confirm the cart details and proceed to OrderService. IT WILL SAVE AND CONFIRM IN DATABASE

1. Place an Order

* Endpoint: GET http://localhost:8086/order/getCartById/{cartId}

Now place an order using cartId;

Example response:

{

"orderId": 1,

"cartId": 1,

"orderStatus": "SUCCESSFUL",

"price": 1800

}

1. Make payment

* Endpoint : GET [http://localhost:9095/pay/{orderId}](http://localhost:9095/pay/%7borderId%7d)

Pay for the order.for making payment a URL will be generated.

Example response:

{

"status": "SUCCESSFUL",

"message": "HERE IS YOUR LINK TO PAY",

"lineItems": [

{

"adjustableQuantity": {

"enabled": true,

"maximum": 10,

"minimum": 1

},

"price": "price\_1QeJcjFKjxizuB9sqD6XOUIp",

"quantity": 1

}

],

"url": "https://buy.stripe.com/test\_9AQ3d85pY4Sm4X68xU"

}

Action : Open the url in browser to complete the payment process

1. User service

* Register as a new User

Endpoint: POST <http://localhost:8090/user/signup>

User will be informed when Signedup by sending email using kafka notification service ;

* Example payload:

{

"userName": "",

"userPhone": "",

"userPassword": "",

"userEmail": "",

"roles": "",

"userHouseNumber": "",

"userStreet": "",

"userLandMark": "",

"city": "",

"state": "",

"country": "",

"postalCode": ""

}

1. Delivery service

* Endpoint: POST [http://localhost:8091/deliveryUser/{cartId}/{email}](http://localhost:8091/deliveryUser/%7bcartId%7d/%7bemail%7d)

Use this endpoint to finalize the delivery after payment.

* Example endpoint :

{

"id": 752,

"createdAt": "2025-01-06T18:58:42.4352385",

"updatedAt": "2025-01-06T18:58:42.4352385",

"userName": "XXXXXX",

"userPhone": "050505050",

"userEmail": "XXXXXX@gmail.com",

"userPassword": "NOT VISIBLE BECAUSE OF PRIVACY REASONS",

"userHouseNumber": "XX",

"userStreet": "XXXXXXX",

"userLandMark": "XXXXXX",

"userCity": "XXXXXXX",

"userState": "XXXXXXX",

"userCountry": "XXXXXXX",

"userPostalCode": XXXXX,

"message": "PARCEL WILL DELIVER IN MAXIMUM 2 DAYS",

"countryDistance": 260, KILOMETERS

"cartId": 1,

"totalAmount": 1800 USD

}

If user cannot find country of city then can check with this api if country or city exists in database or not

Endpoint GET <http://localhost:8091/Destination/>

* 1. Spring Oauth2 authorization server used for security (role based access as ADMIN,USER)
  2. Error handling done by each service.
  3. Kafka used for notification services

Note :

Delivery service used to find the estimate days of delivery which is hardcoded calculation to measure the distance warehouse and destination based on the country ,state, city and postal code.

Format Guidelines

1. Detailed and Elaborate report of 40 pages at least is expected
2. Margins - Every page of your document must meet the margin requirements of 1.25 inches on the left and right, and 1 inch on the top and bottom.
3. Font:
   1. Style: Times New Roman,
   2. Font Size:14 (For Headings), 12 (For body text) in black colored text.
   3. All text must be the same justification, like left justified or fully justified.
4. Line Spacing:
   1. Body of the text: 1.5
   2. List of Tables/graphs/charts/bibliography: single line.
5. Alignment:
   1. Title page: Centre
   2. Chapter Heading: Centre
   3. Subheading: Left
   4. Body of the text: Justify
6. Titles: All titles and subtitles should be in bold. All tables/graphs/charts/figures should have appropriate titles.
7. Numbering of the tables, charts, graphs should be in the following fashion: Second table/graph/chart in the second chapter should be numbered as Table/graph/chart no. 2.02; where the first digit stands for chapter no. and digits after (.) stands for number of table/graph/charts in that chapter. Same numbering should be followed for all other chapters.