HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

School of Information and communications technology

Software Requirement Specification

Version 1.3

AIMS Project

Subject: ITSS Software Development

Group 07:

No	Name	Student ID
1	Trịnh Thị Thuỳ Dương	20226034
2	Nguyễn Thị Thu Huyền	20220073
3	Hoàng Khải Mạnh	20225984
4	Đoàn Thị Thu Quyên	20226063
5	Dương Phương Thảo	20226001

Hanoi, May 11^{st} *2025*

Hanoi, <May, 2025>

<All notations inside the angle bracket are not part of this document, for its purpose is for extra instruction. When using this document, please erase all these notations and/or replace them with corresponding content as instructed>

<This document, written by Prof. NGUYEN Thi Thu Trang, is used as a case study for student with related courses. Any modifications and/or utilization without the consent of the author is strictly forbidden>

Table of Contents

List of Figures

No table of figures entries found.

List of Tables

1 Introduction

<The following subsections of the Software Design Document (SDD) document should provide an overview of the entire SDD.>

1.1 Objective

<Identify the purpose of this SDD and its intended audience. In this subsection, describe the purpose of the SDD and specify the intended audience for the SDD>

1.2 Scope

The software product to be developed is **AIMS (An Internet Media Store)**, a digital platform that enables users to browse, purchase, and download a wide range of media content, including applications, music, videos, and e-books. AIMS serves as a centralized marketplace designed to deliver a seamless and secure media consumption experience across web and mobile platforms. It can handle up to 1,000 customers at once without slowing down much and works for 300 hours without crashing. If something goes wrong, it gets back to normal within 1 hour. It responds in 2 seconds normally or 5 seconds during busy times.

AIMS will allow users to create accounts, search and filter media content, manage their purchases, and access their media library anytime. The platform will also support content uploads and management for verified publishers. Its primary goals include enhancing media accessibility, streamlining digital content distribution, and supporting user-friendly interactions.

AIMS will not handle content creation or moderation beyond basic automated checks; these responsibilities lie with the content providers and external policies. This overview is consistent with the system's Software Requirements Specification (SRS) and outlines the scope, benefits, and intended use of the application without detailing individual requirements.

1.3 Glossary

No	***Term***	***Explanation***	***Example***	***Note***
1	token	A piece of data created by server, and contains the user's information, as well as a special token code that user can pass to the server with every method that supports authentication, instead of passing a username and password directly.	JSON Web Token (JWT)	Compact, URL-safe and usable especially in web browser single sign-on (SSO) context.
2	API (Application Programming Interface)	A set of rules that allows different software applications to communicate with each order	VNPay API	Includes online shopping, digital payments, and logistics
3	E-Commerce (Electronic Commerce)	The buying and selling of goods and services over the internet	Amazon, Shopee	Includes online shopping, digital payment, and logistics

4	VAT (Value Added Tax	Value-added tax, a consumption tax added to goods and services.	10% VAT in Vietnam	Applied at each stage of production and distribution
5	Rush Order	An order that requires expedited processing and delivery.	Express shipping, Same-day delivery.	May involve additional fees for faster service.
6	Payment Gateway	A service that authorizes and processes online payments securely.	VNPay	Ensures secure transactions between customers and merchants.

1.4 References

Centers for Medicare & Medicaid Services. (n.d.). Test Case Specification. Retrieved from Centers for Medicare & Medicaid Services: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/TestCaseSpecification.docx

Centers for Medicare & Medicaid Services. (n.d.). Test Plan. Retrieved from Centers for Medicare & Medicaid Services: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/TestPlan.docx

guru99. (n.d.). Download Sample Test Case Template: Example Excel, Word Formats. Retrieved from guru99: https://www.guru99.com/download-sample-test-case-template-with-explanation-of-important-fields.html

2 Overall Description

<This section describes the principles and strategies to be used as guidelines when designing and implementing the system.>

2.1 General Overview

2.1.1. System Context and Design Philosophy

The design of AIMS (An Internet Media Store) centers around delivering a robust and scalable e-commerce platform for physical media products, tailored to meet both business requirements and user expectations. At its core, the system is built to be reliable, responsive, and easy to use — even under demanding conditions. Key design objectives include:

- **High Performance**: The system is expected to handle user requests swiftly, ensuring responsive interactions even during peak traffic periods.
- **Scalability**: AIMS aims to support up to 1,000 concurrent users, providing a seamless experience regardless of user volume.
- **Reliability**: The system should remain operational for at least 300 hours continuously and be able to recover within 1 hour following any downtime.
- **Security**: Strong security practices are integrated, including encrypted data storage and secure user authentication, to protect user privacy and transaction safety.

2.1.2. System and Software Architecture

- **Client Application**: A modern, web-based interface built using technologies like React, HTML, CSS, and JavaScript. It enables users to browse products, manage shopping carts, and complete purchases with ease.
- **Server Application**: The server-side logic is implemented using the SpringBoot framework, which handles business operations, database management, and integration with third-party services.
- **Database**: A relational database serves as the backbone for storing user accounts, product listings, order details, and transaction history.
- **External Integration**: The system integrates with **VNPay**, a third-party payment gateway, to securely process credit card transactions.

2.2 Assumptions/Constraints/Risks

2.2.1 Assumptions

<Describe any assumptions or dependencies regarding the system, software and its use. These may concern such issues as: related software or hardware, operating systems, end-user characteristics, and possible and/or probable changes in functionality>

- **Platform**: Users will interact with AIMS via up-to-date web browsers such as Chrome, Firefox, Safari, or Edge.

- **User Capabilities**: It is assumed that users have basic computer literacy and stable internet access.

- **Third-party Services**: VNPay's APIs are assumed to be stable and consistently available.

- **Scope**: The system will exclusively handle physical media items; digital goods and subscription services are not included.

2.2.2 Constraints

<Describe any global limitations or constraints that have a significant impact on the design of the system's hardware, software and/or communications, and describe the associated impact. Such constraints may be imposed by any of the following (the list is not exhaustive):

· *Hardware or software environment*

· *End-user environment*

· *Availability or volatility of resources*

· *Standards compliance*

· *Interoperability requirements*

· *Interface/protocol requirements*

· *Licensing requirements*

· *Data repository and distribution requirements*

· *Security requirements (or other such regulations)*

· *Memory or other capacity limitations*

· *Performance requirements*

· *Network communications*

· *Verification and validation requirements (testing)*

· *Other means of addressing quality goals*

· *Other requirements described in the Requirements Document >*

- **Hardware Environment**: The AIMS system must run smoothly on standard consumer computers without requiring high-performance hardware, which necessitates performance optimization and limited resource usage.

- **User Environment**: The user interface must be simple and intuitive for non-technical users, influencing the overall UI/UX design approach.

- **Resource Availability**: The system must support up to 1,000 concurrent users and operate continuously for 300 hours, requiring efficient resource management and quick recovery mechanisms.

- **Performance Requirements**: The application must respond within 2 seconds under normal load and within 5 seconds during peak times, affecting logic design and data handling strategies.

- **Network Communications**: Communication between frontend and backend must be stable and low-latency to support real-time feedback, requiring optimized APIs and asynchronous processing.

- **Standards Compliance & Licensing**: Integration with VNPay requires strict adherence to their API standards, which impacts the modular design of the payment system.

- **Interoperability & Protocol Requirements**: The system must interoperate with external APIs using standard protocols like HTTPS and JSON, affecting how external communications are structured and validated.

- **Security Requirements**: Security of user and payment information is critical, necessitating role-based access control, input validation, and data encryption.

- **Data Repository Constraints**: The system must store detailed logs of products, transactions, and user actions, impacting database schema design and logging mechanisms.

- **Capacity Limitations**: Business rules such as deleting a maximum of 30 products at once or updating prices only twice per day must be strictly enforced through system logic.

- **Verification and Validation Requirements**: All input must be validated with user-friendly error feedback, increasing the need for input checking, error handling, and testing coverage.

- **Scalability and Future Expansion**: The system should be scalable to support new product types and payment methods, requiring modular architecture, SOLID principle adherence, and appropriate design patterns.

2.2.3 Risks

<Describe any risks associated with the system design and proposed mitigation strategies.>

Risk	Description	Mitigation Strategy
Downtime	System outages may affect the shopping experience.	Employ redundancy and automatic failover systems.
Security Vulnerabilities	Risks of data breaches and unauthorized access.	Perform regular security audits, updates, and enforce best practices.
Scalability Limitations	Inability to support increasing traffic.	Optimize queries and backend logic, and implement load balancing.
Payment Integration Issues	Failures in VNPay integration could block payments.	Conduct continuous integration testing and maintain communication with VNPay support.
Performance Bottlenecks	Slow responses under heavy usage.	Conduct load testing, enable caching, and optimize backend processes.
Data Loss	Potential data corruption or loss due to bugs or hardware failure.	Schedule regular backups and implement integrity checks.

3 System Architecture and Architecture Design

Architectural Design Steps

The system architectural design process typically includes the following steps:

1. Analyze System Requirements:

- Gather and analyze functional and non-functional requirements from clients, users, and other stakeholders.

2. Identify Key Components:

- Define the main modules and functional components of the system (e.g., frontend, backend, database, services, etc.).

3. Select a Suitable Architectural Pattern:

- Based on the system's requirements and characteristics, choose appropriate architectural patterns such as Layered, Client-Server, Microservices, etc.

4. Design the Detailed Architecture:

- Describe the components in detail, how they interact with each other, their interfaces (APIs), data flows, security measures, etc.

5. Define Technologies and Tools:

- Select the technologies, frameworks, and libraries that will be used for each component.

6. Document the Architecture:

- Draw overall architectural diagrams, describe the components, data flows, and communication between modules.

3.1 Architectural Patterns

Client-Server Architecture

The Client-Server Architecture is a distributed application structure that partitions tasks or workloads between service providers (servers) and service requesters (clients). In this project, the architecture is divided into two main components:

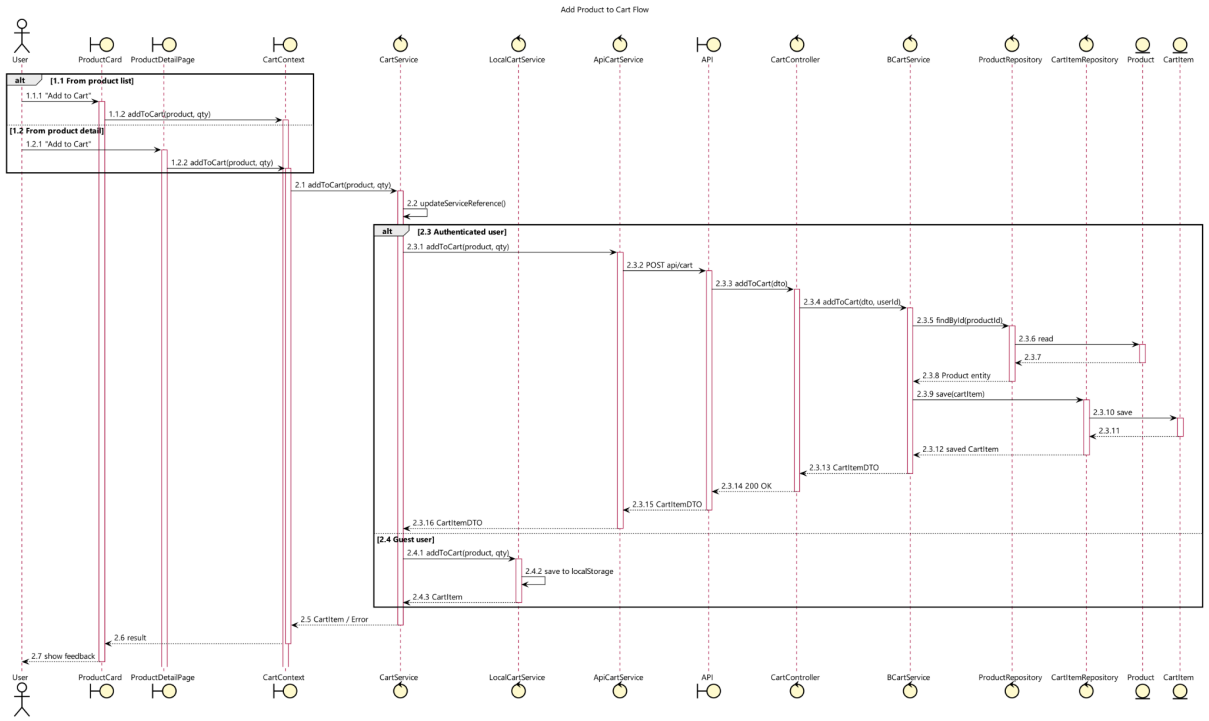
1. Client (Frontend):
    - Implemented using ReactJS (located in the `frontend/` directory).
    - Responsible for presenting the user interface, handling user interactions, and sending requests to the server.
    - Communicates with the backend server via HTTP requests to RESTful APIs.
2. Server (Backend):
    - Implemented using Spring Boot (located in the `backend/` directory).
    - Handles business logic, processes client requests, manages data, and enforces security and validation.
    - Exposes RESTful APIs for the client to interact with and communicates with the database to store and retrieve data.

Reasons for Choosing Client-Server Architecture:

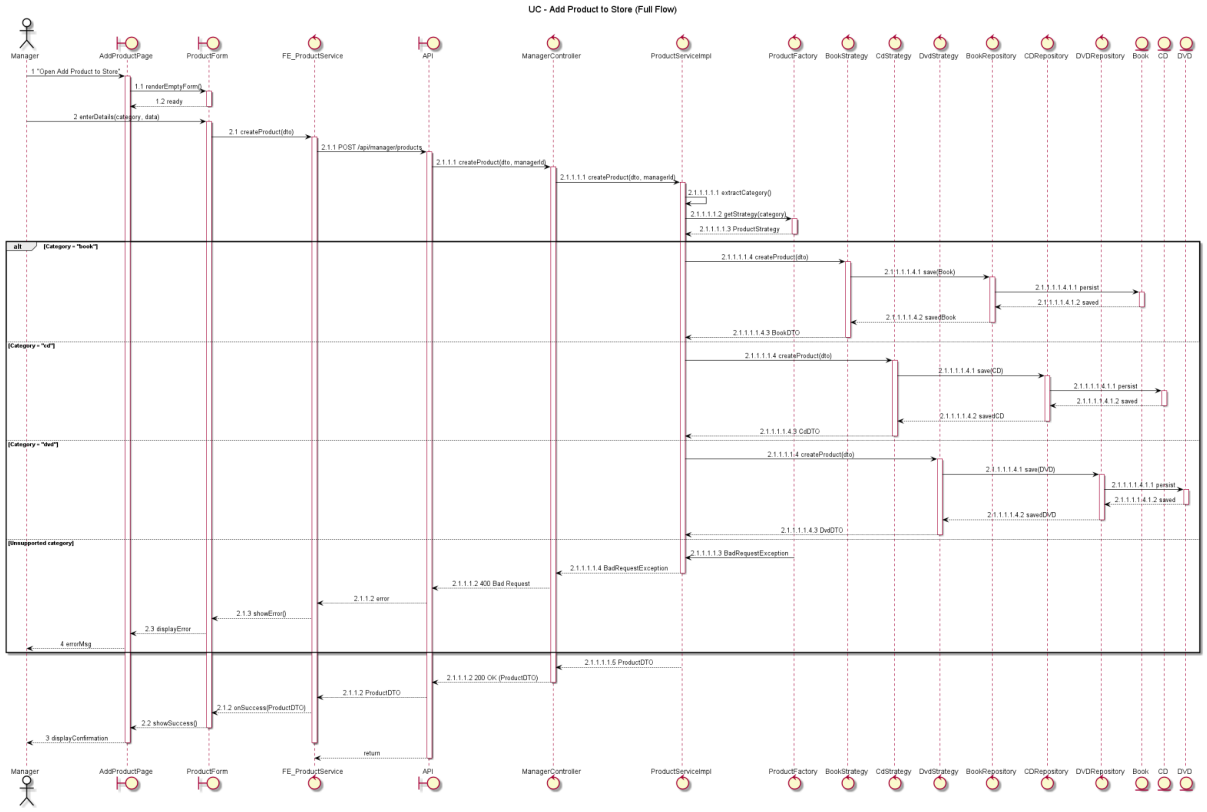
- Separation of Responsibilities: The client focuses on the user experience, while the server handles data processing and business logic.
- Independent Development and Deployment: The frontend and backend can be developed, tested, and deployed independently, allowing for greater flexibility and scalability.
- Scalability: Each component can be scaled separately based on demand (e.g., more frontend instances for more users, more backend instances for heavier processing).
- Security: Sensitive operations and data are handled on the server side, reducing the risk of exposing business logic or data to the client.
- Maintainability: Updates or changes to the frontend or backend can be made independently, improving maintainability and reducing downtime.

3.2 Interaction Diagrams

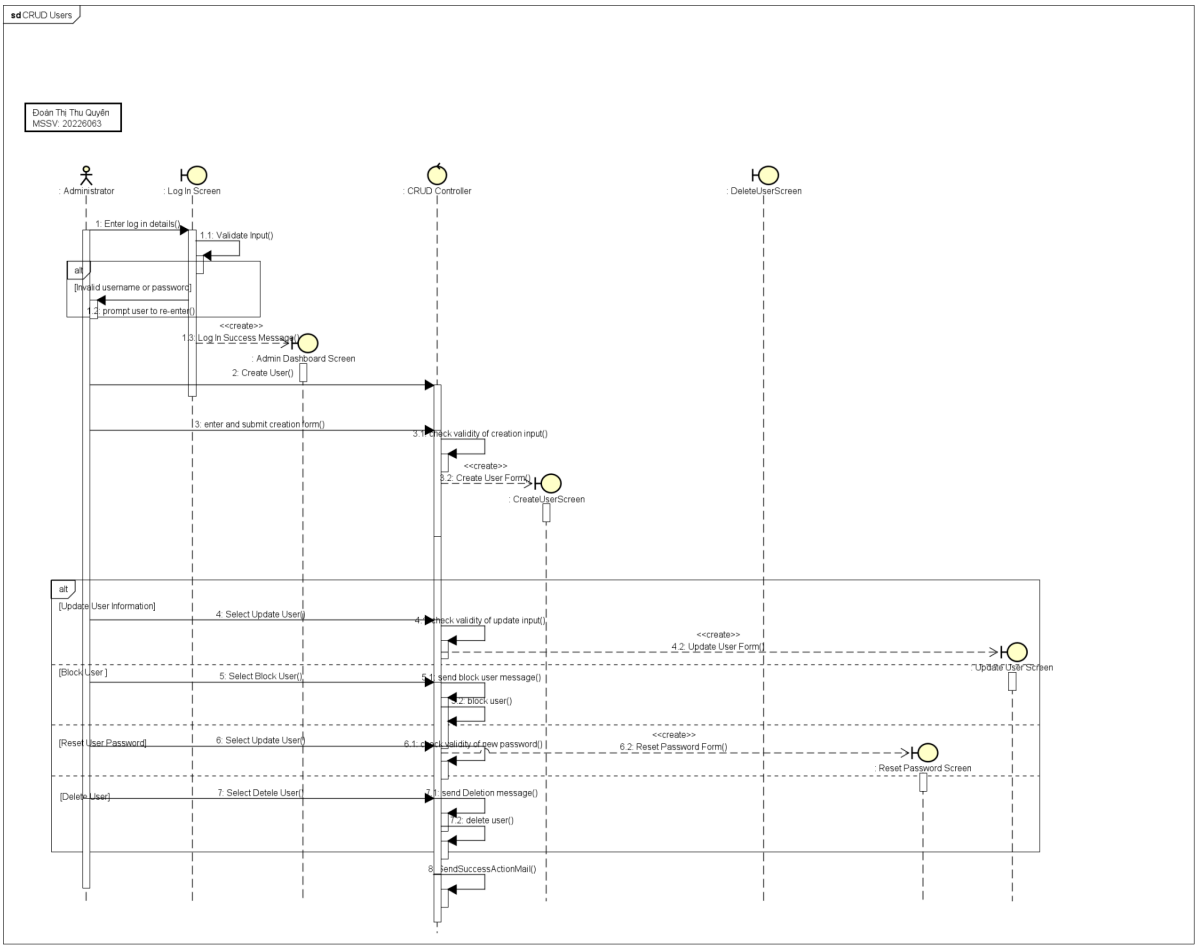
3.2.1. Add Product to Cart



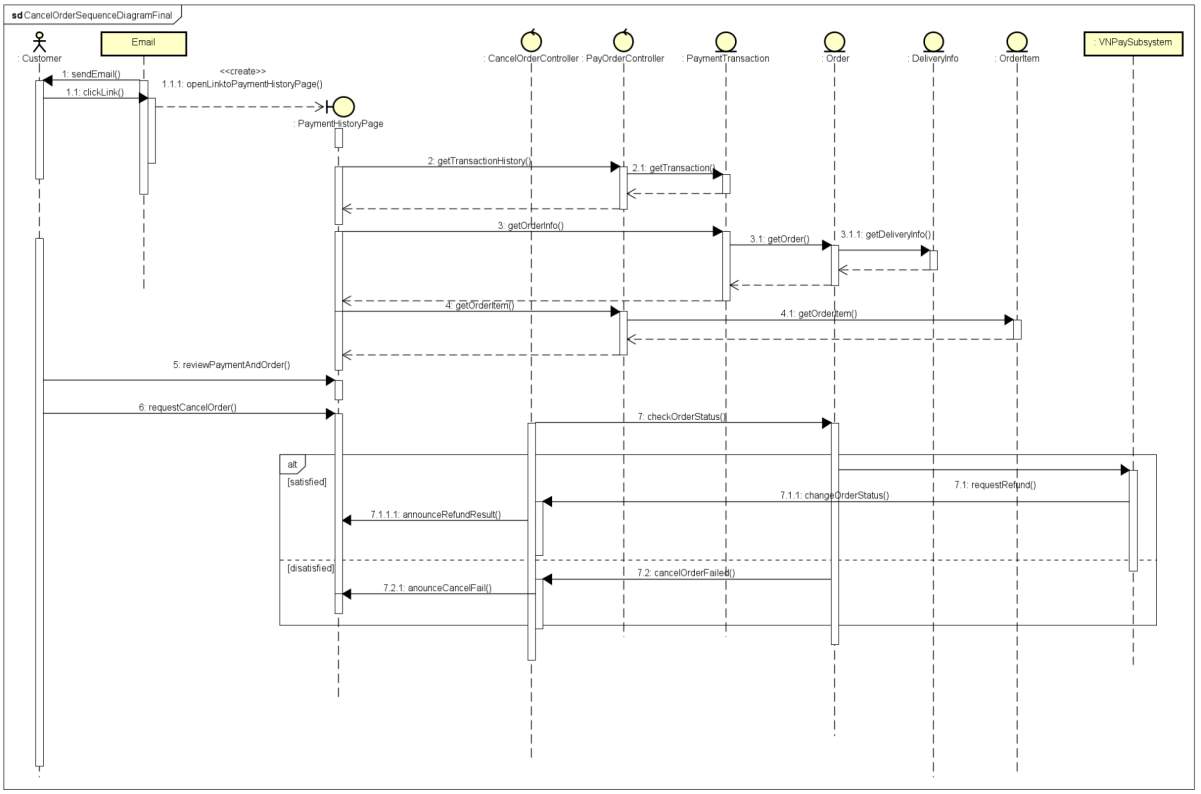
3.2.2. Add Product to Store



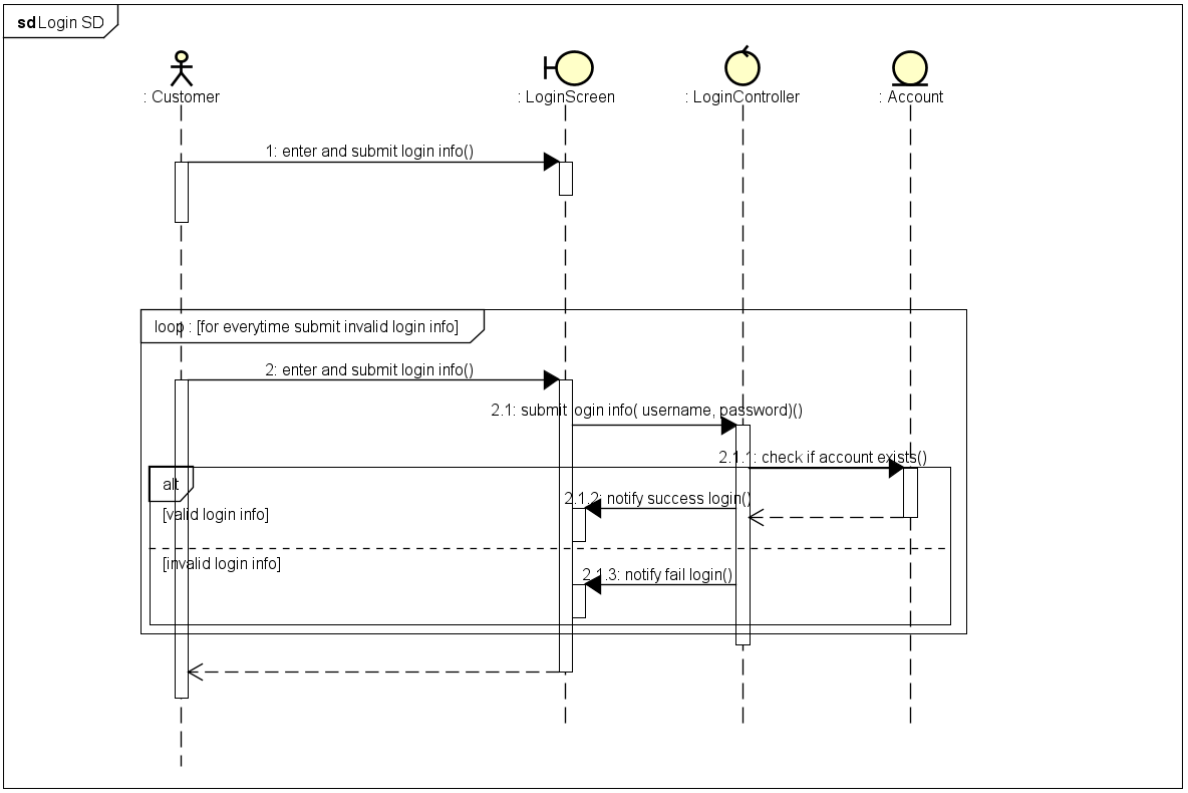
3.2.3. CRUD Users



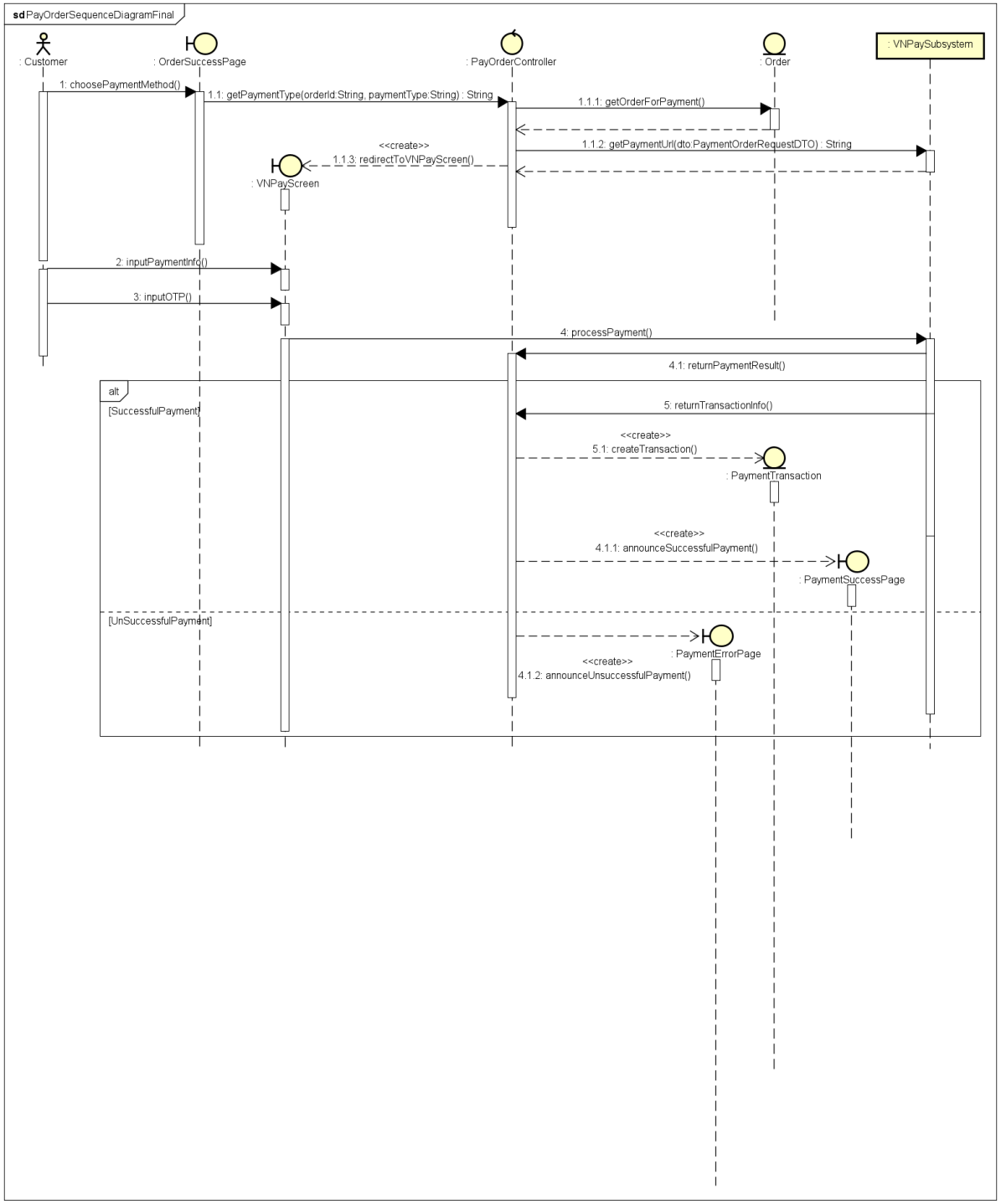
3.2.4. Cancel Order



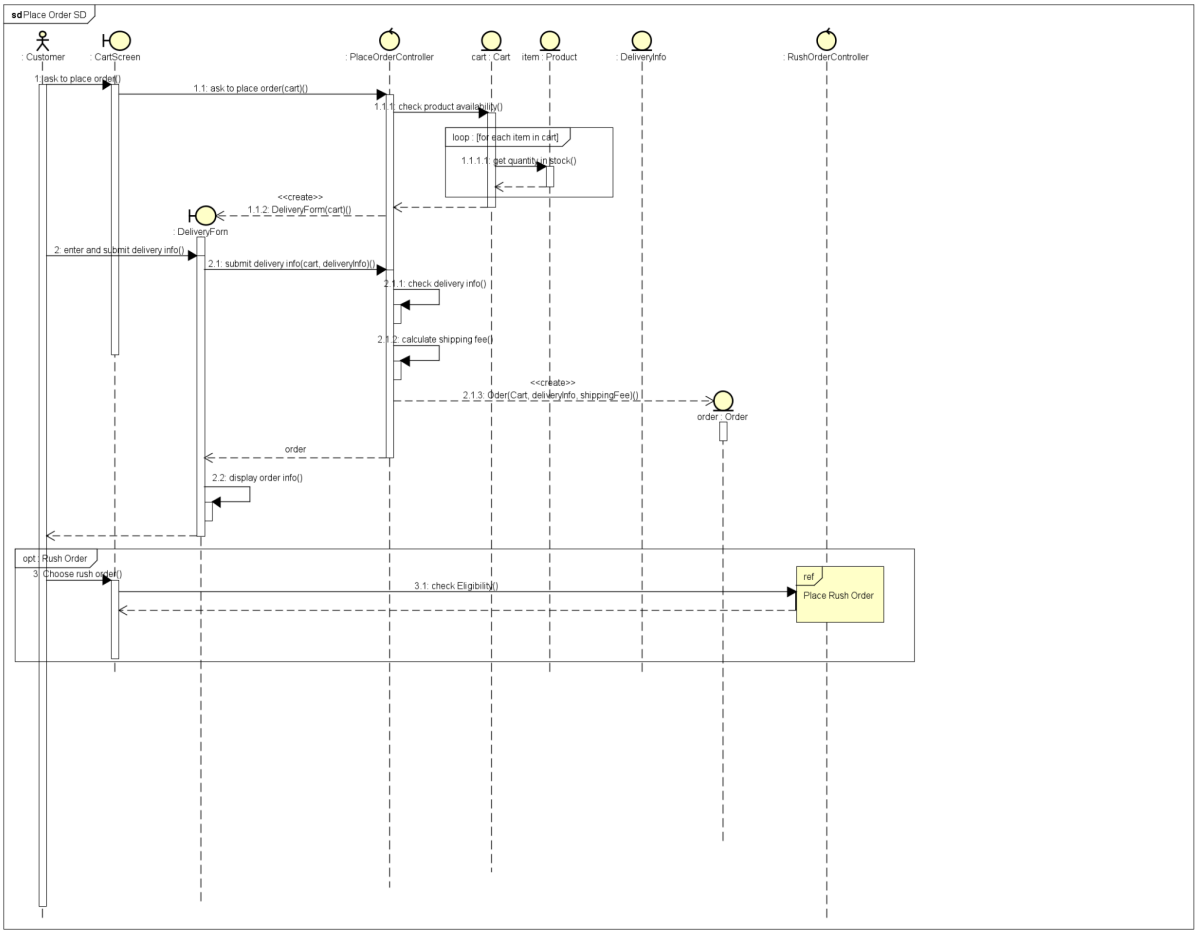
3.2.5. Login



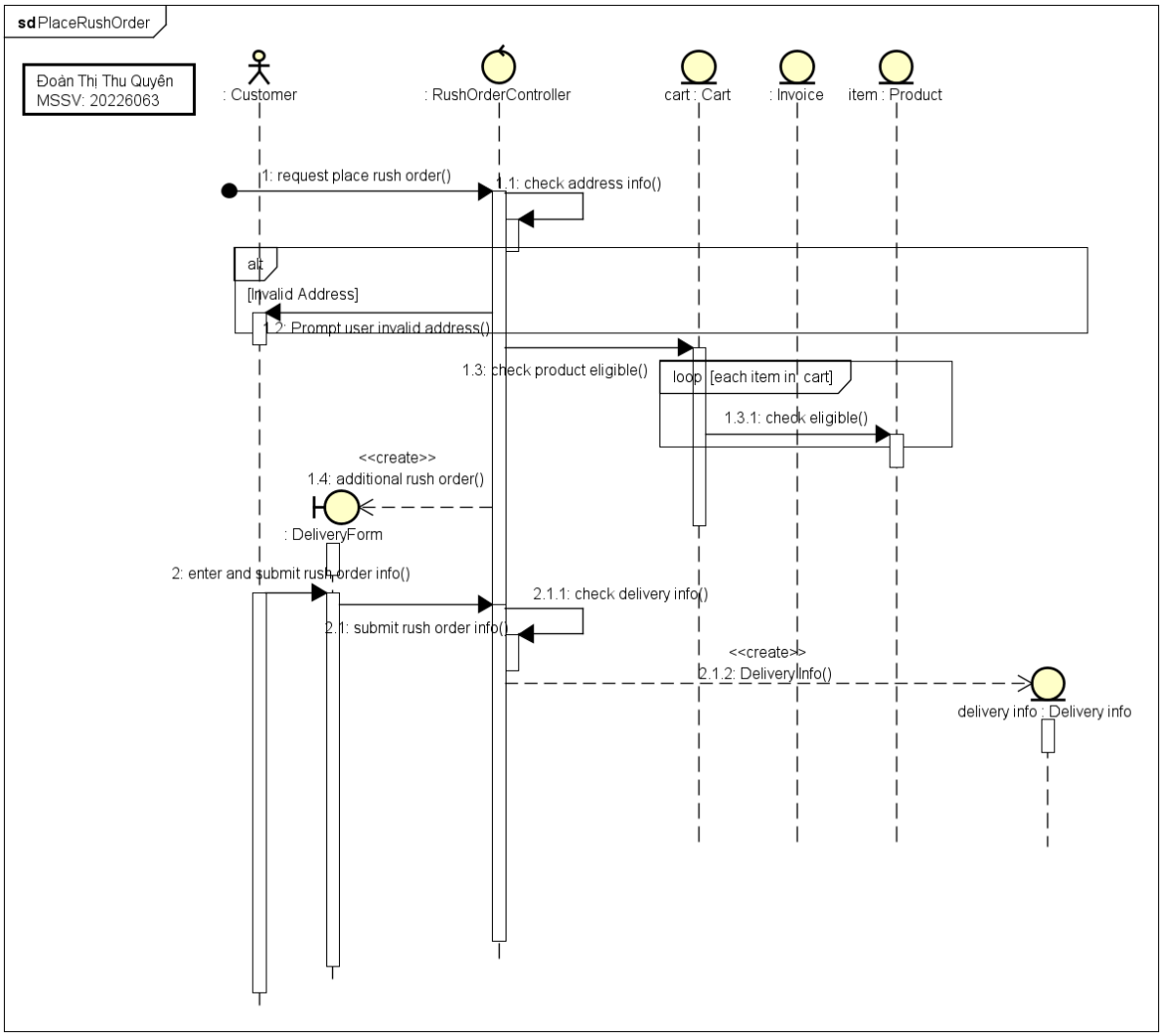
3.2.6. Pay Order



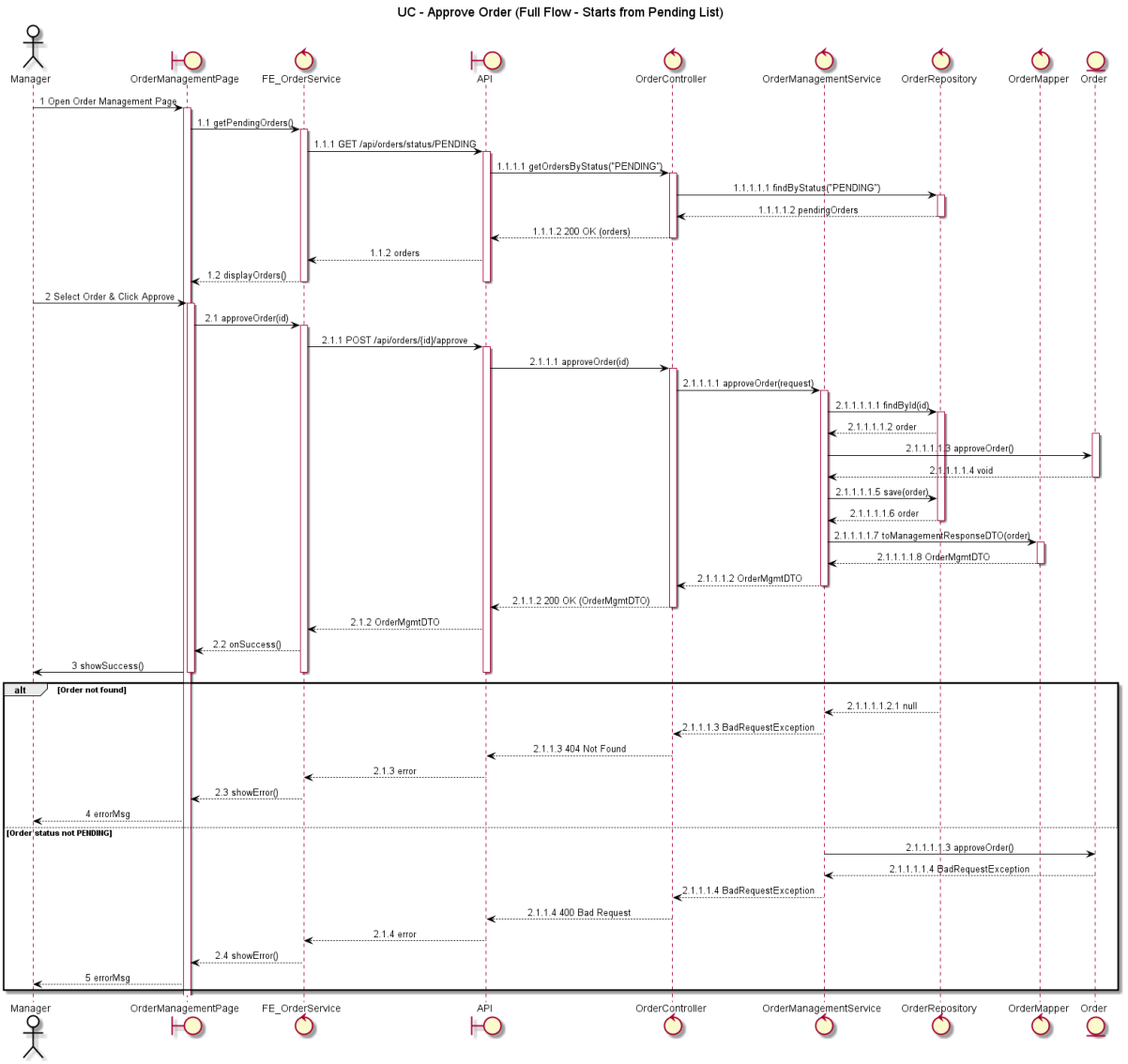
3.2.7. Place Order



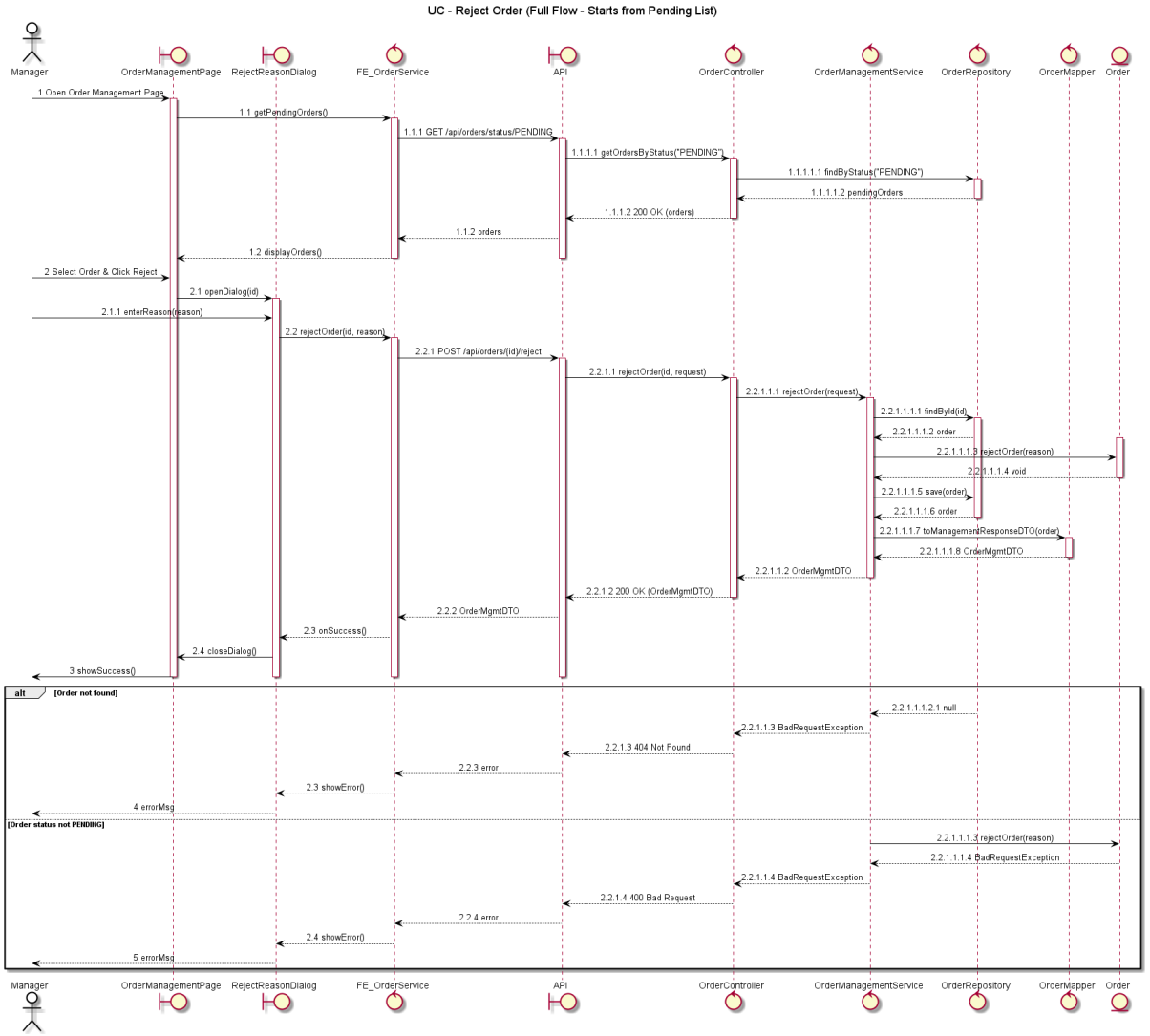
3.2.8. Place Rush Order



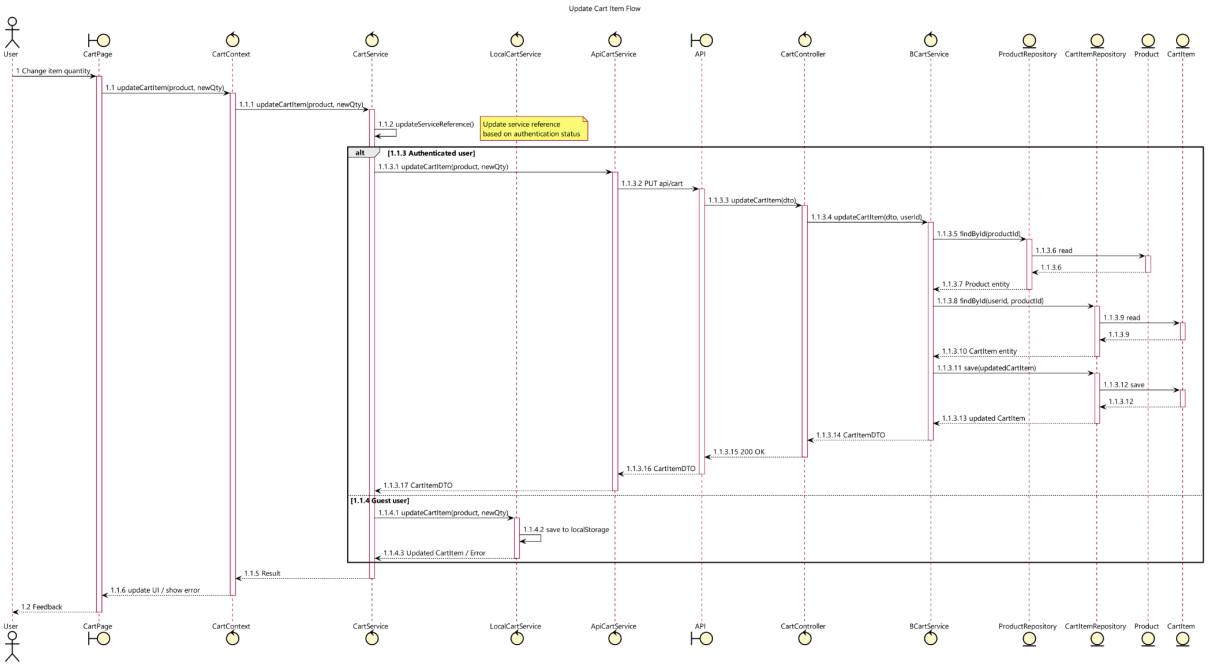
3.2.9. Approve Order



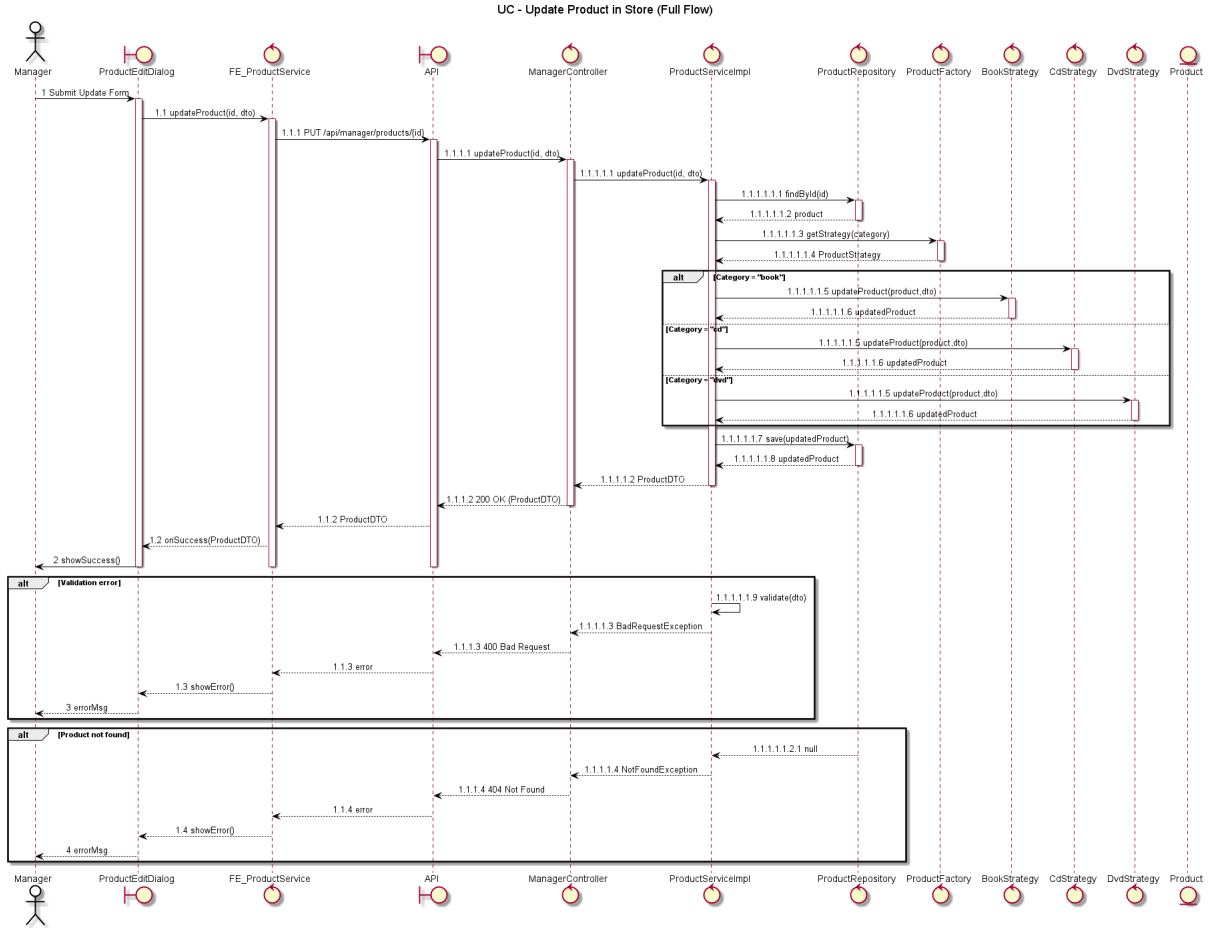
3.2.10. Reject Order



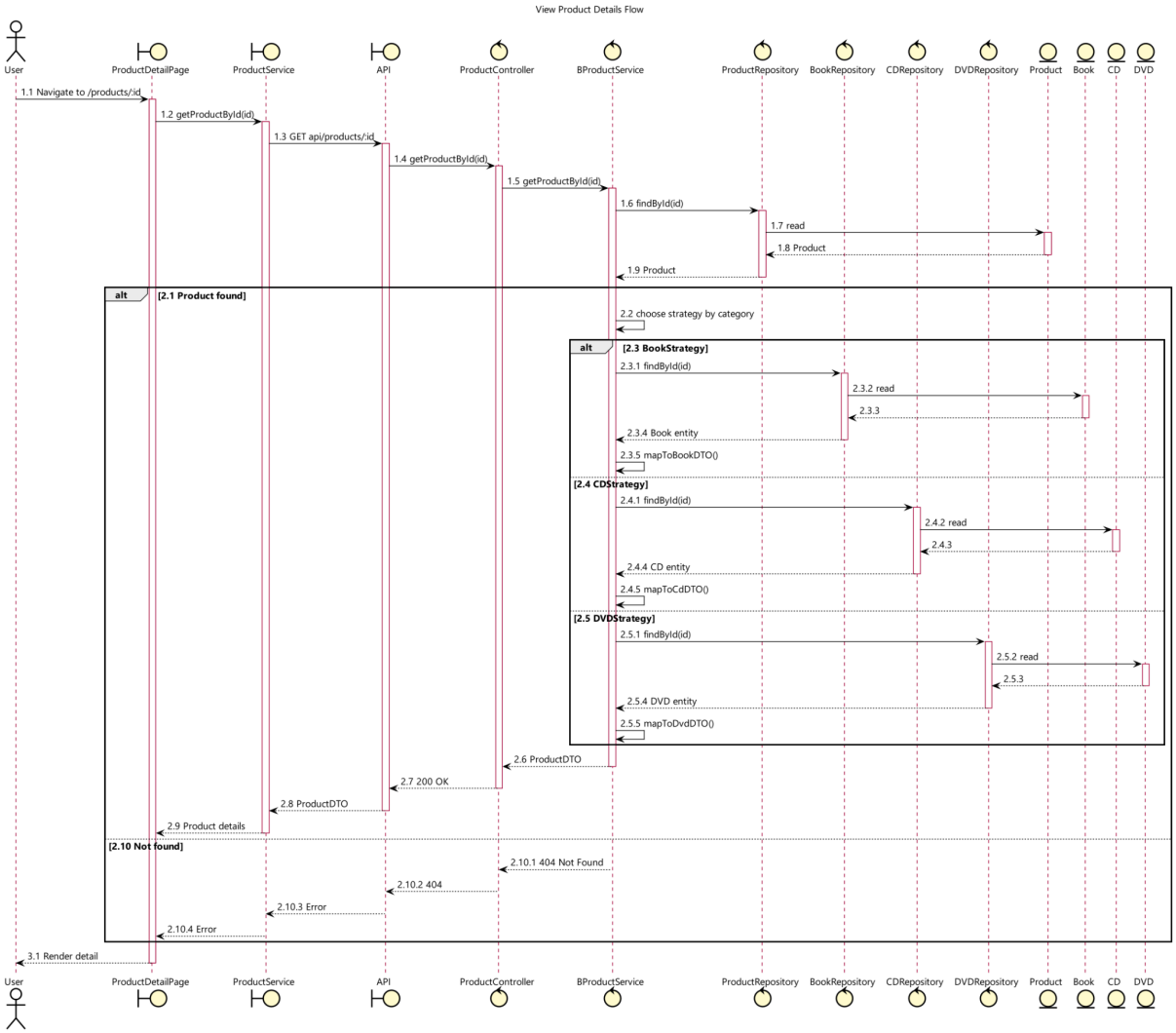
3.2.10. Update Cart



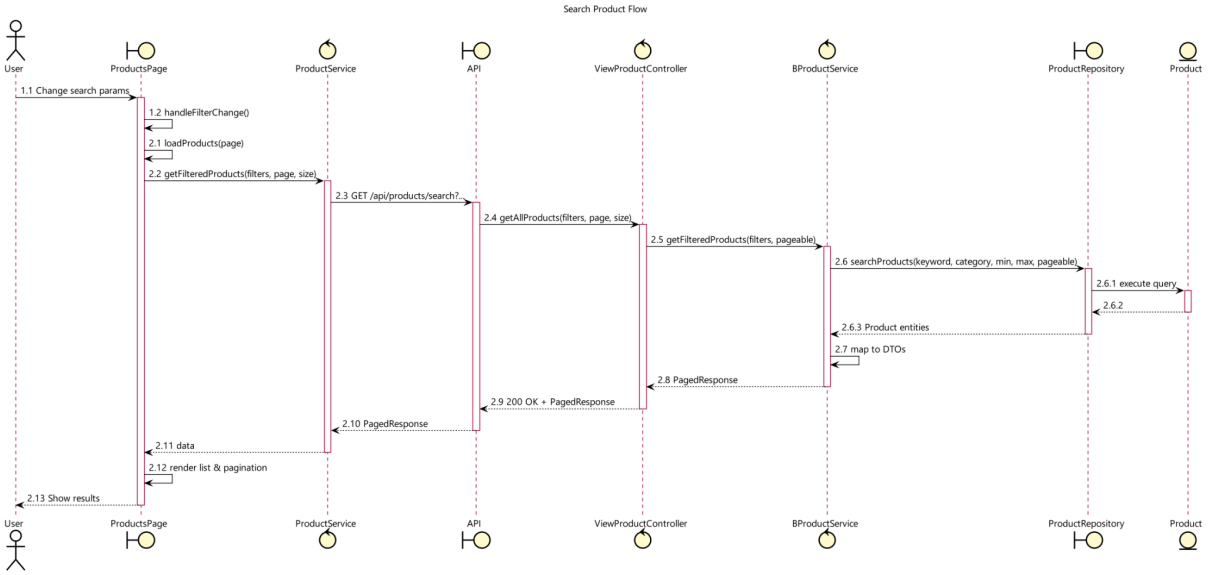
3.2.11. Update Product to Store



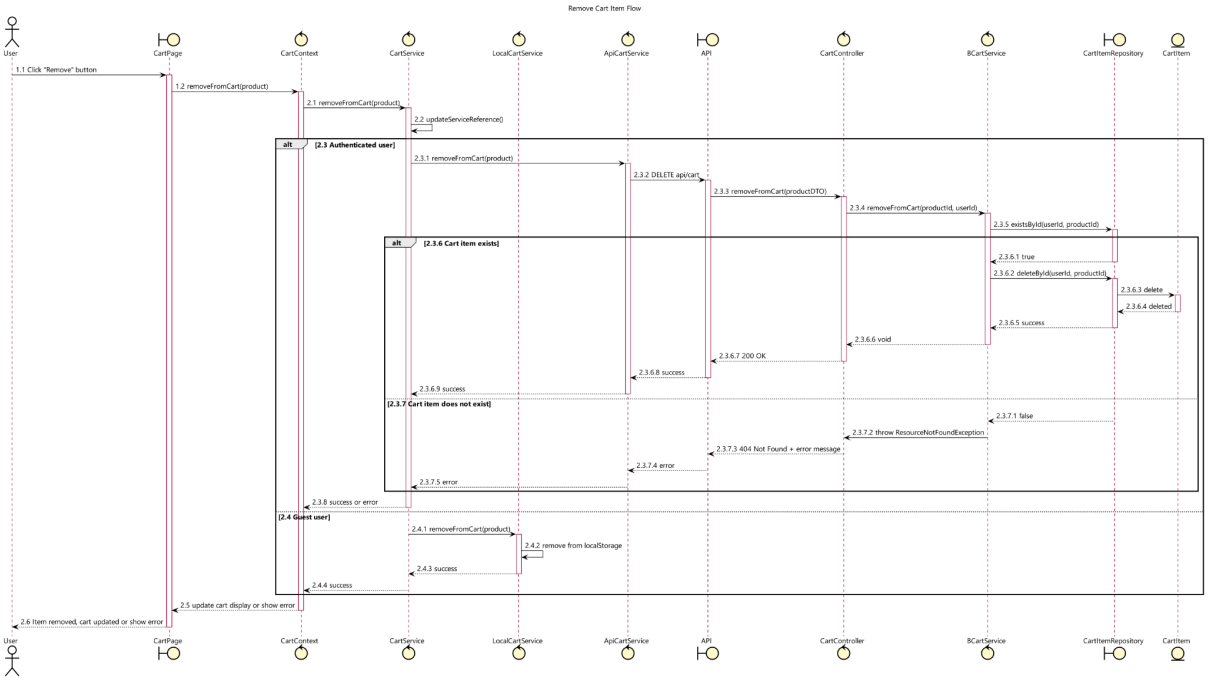
3.2.12. View Product Details



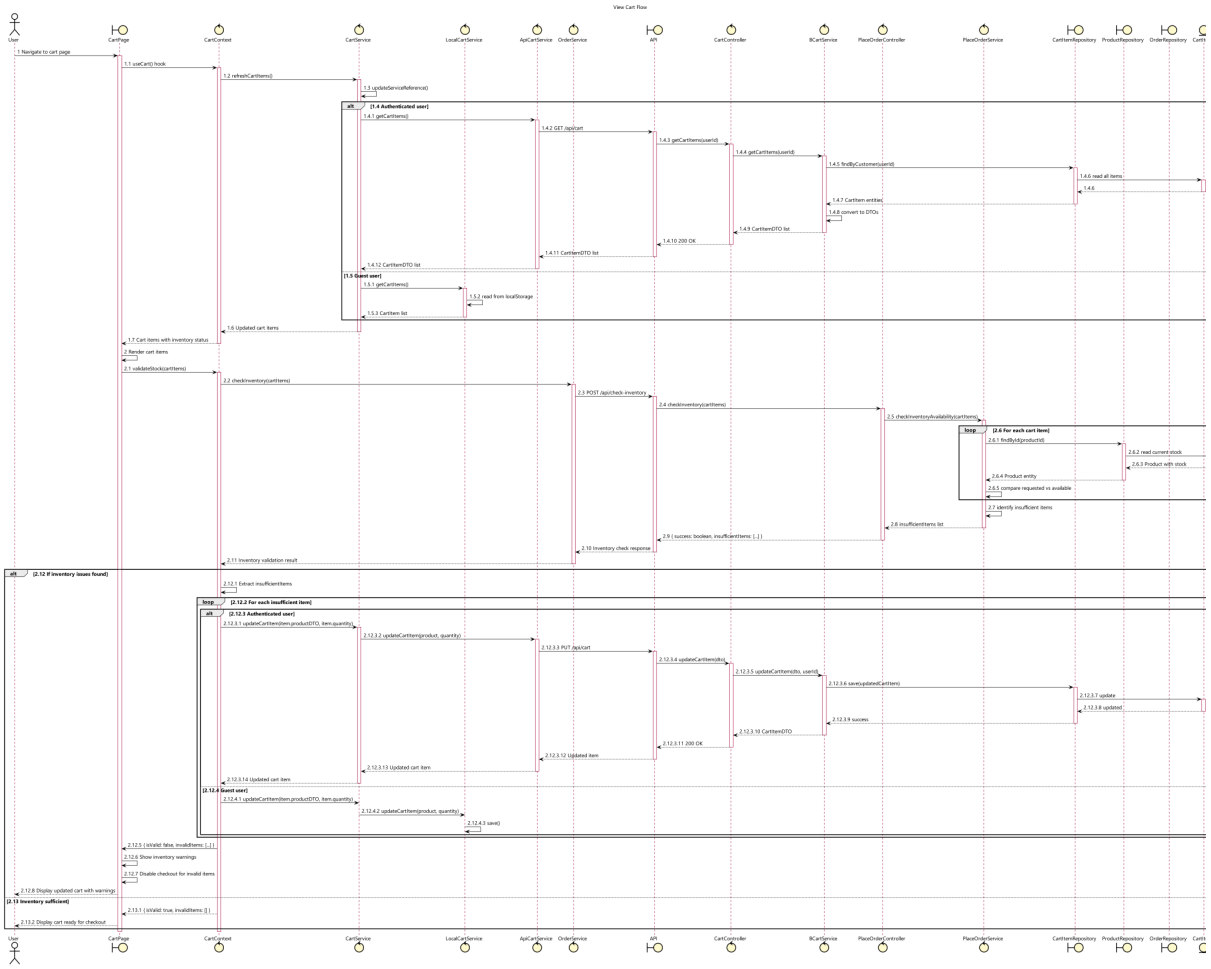
3.2.13. Search Products



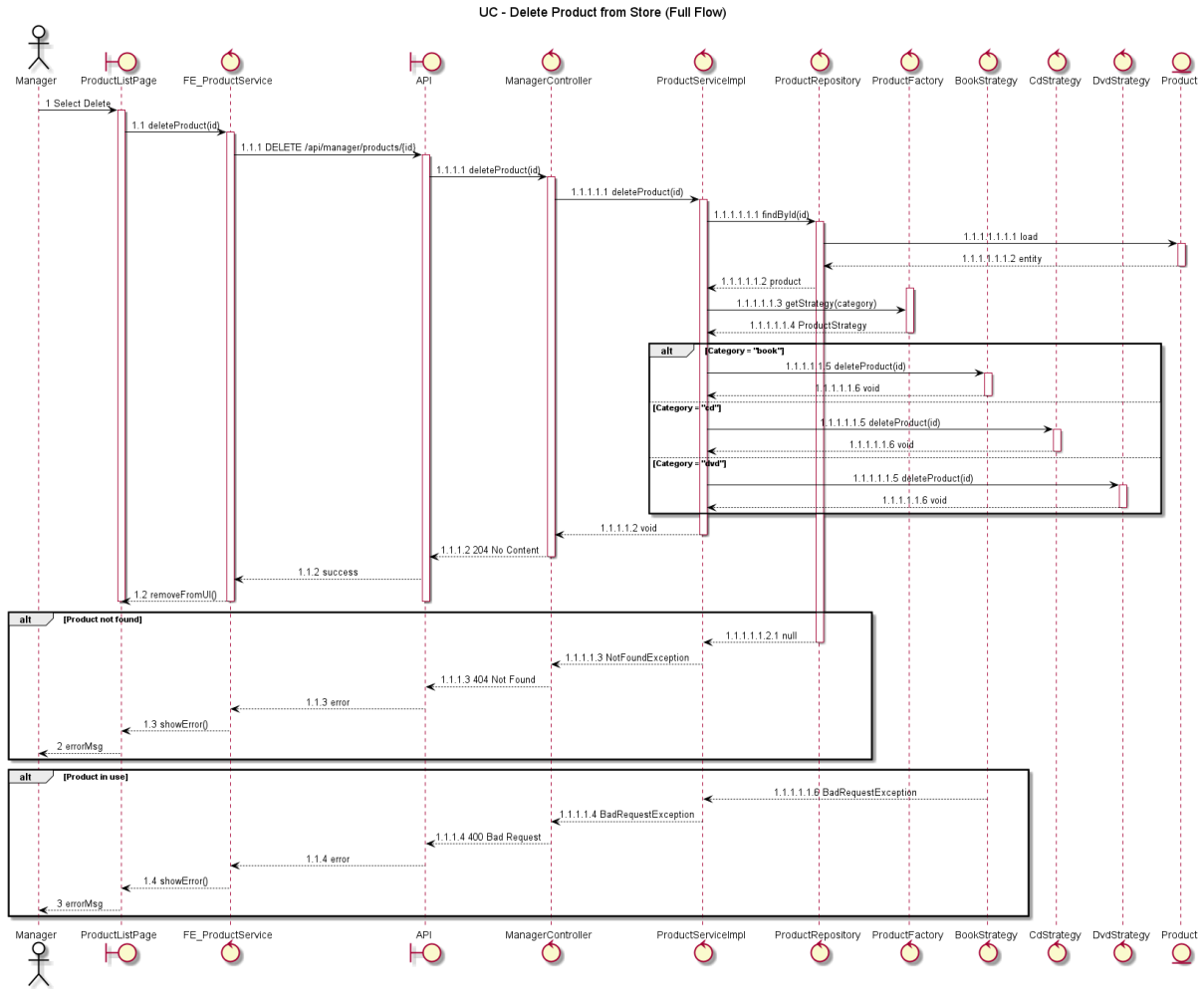
3.2.14. Remove Cart item



3.2.15. View Cart

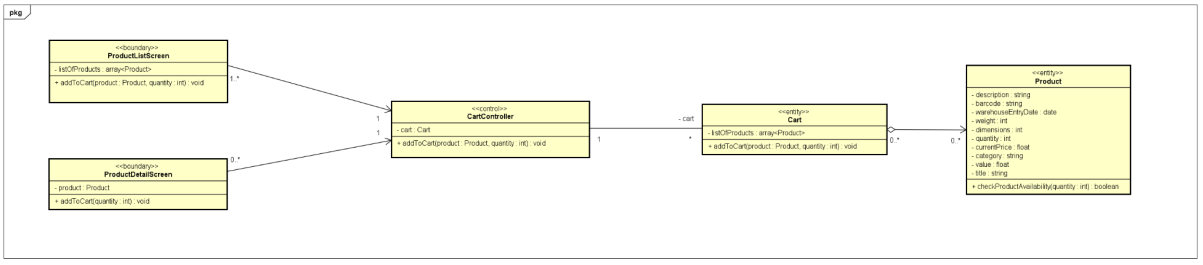


3.2.16. Delete product to store

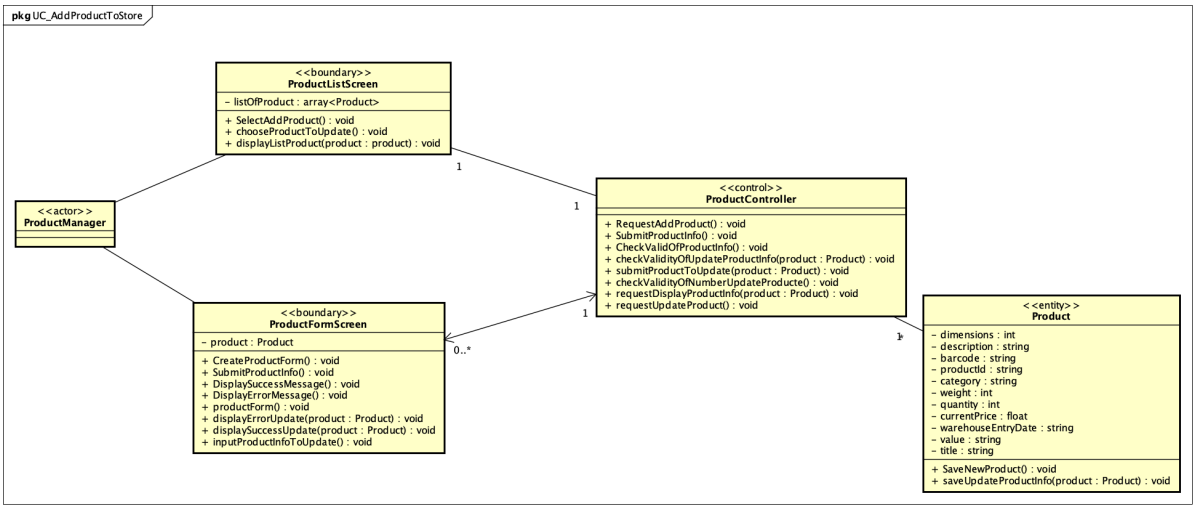


3.3 Analysis Class Diagrams

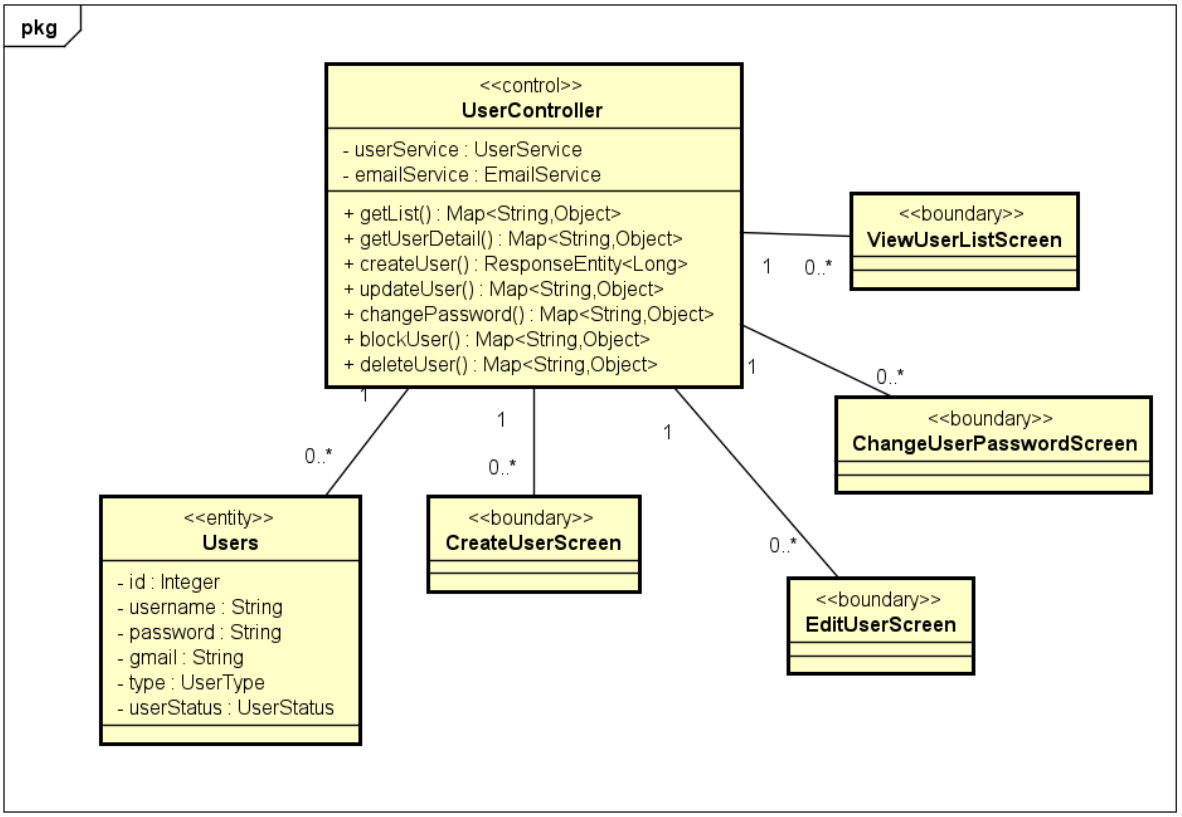
3.3.1. Add Product to Cart



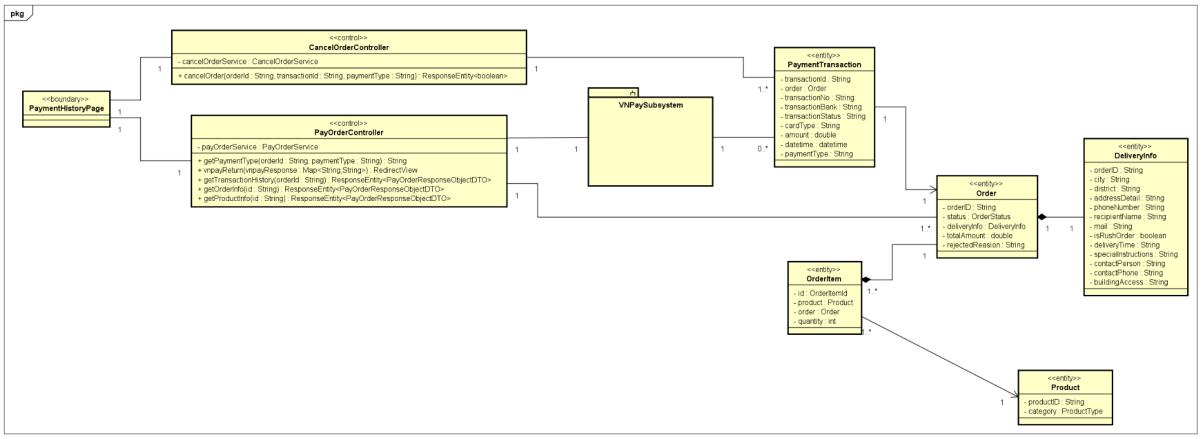
3.3.2. Add Product to Store



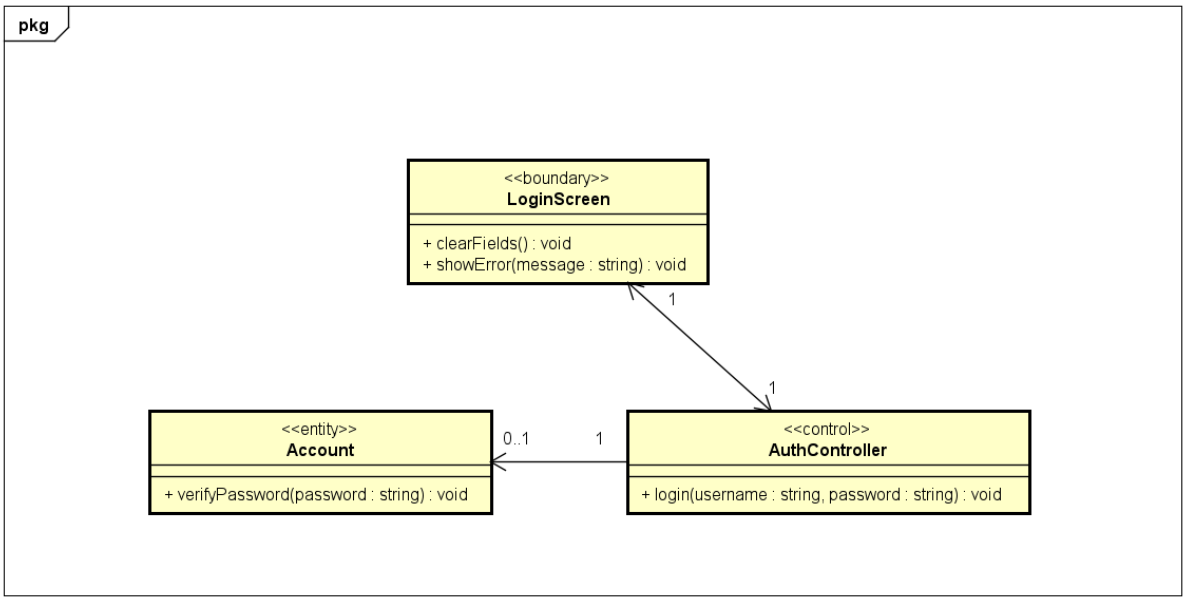
3.3.3. CRUD Users



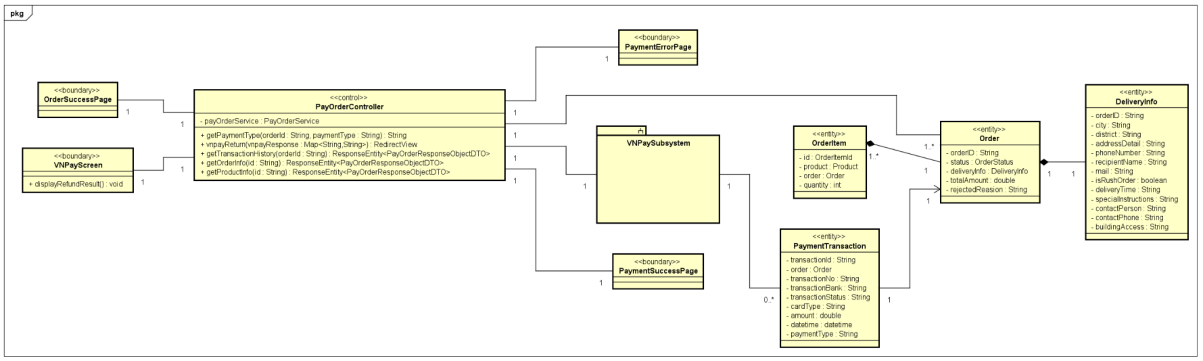
3.3.4. Cancel Order



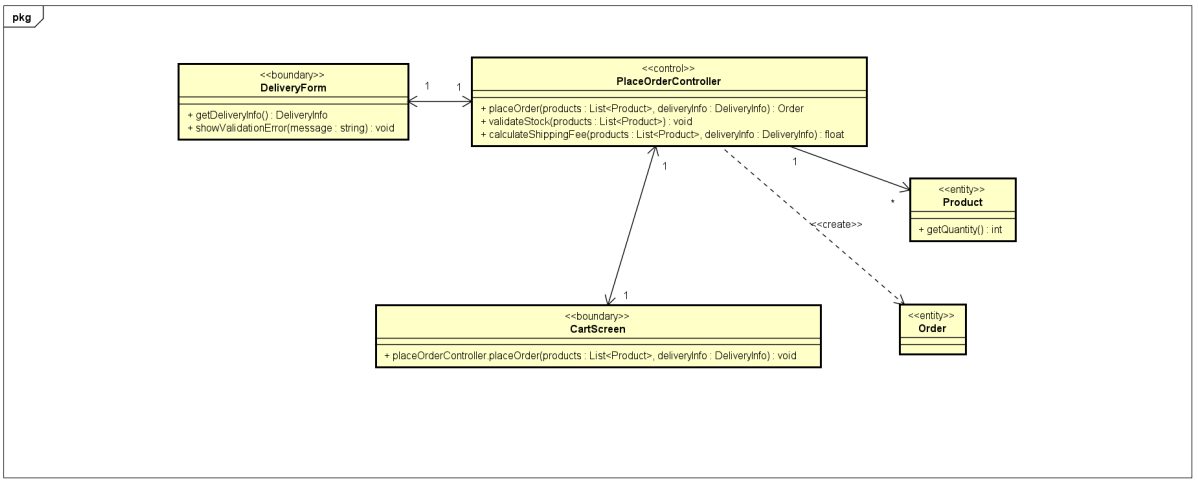
3.3.5. Login



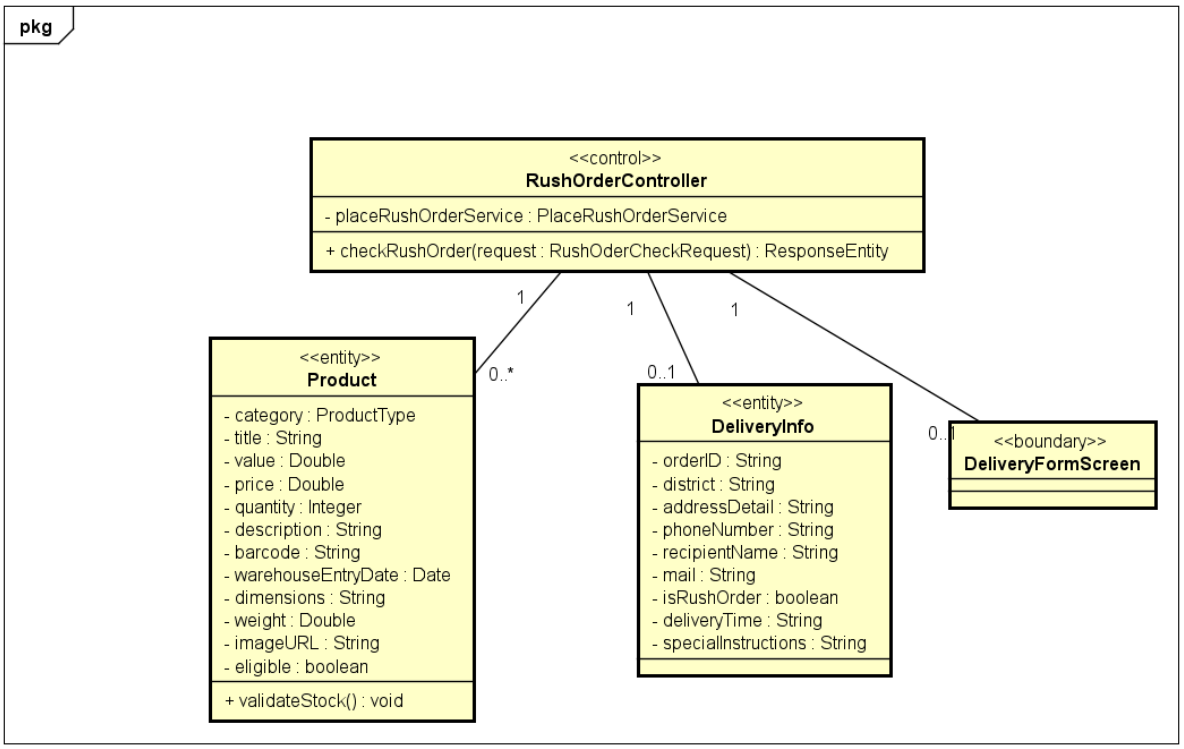
3.3.6. Pay Order



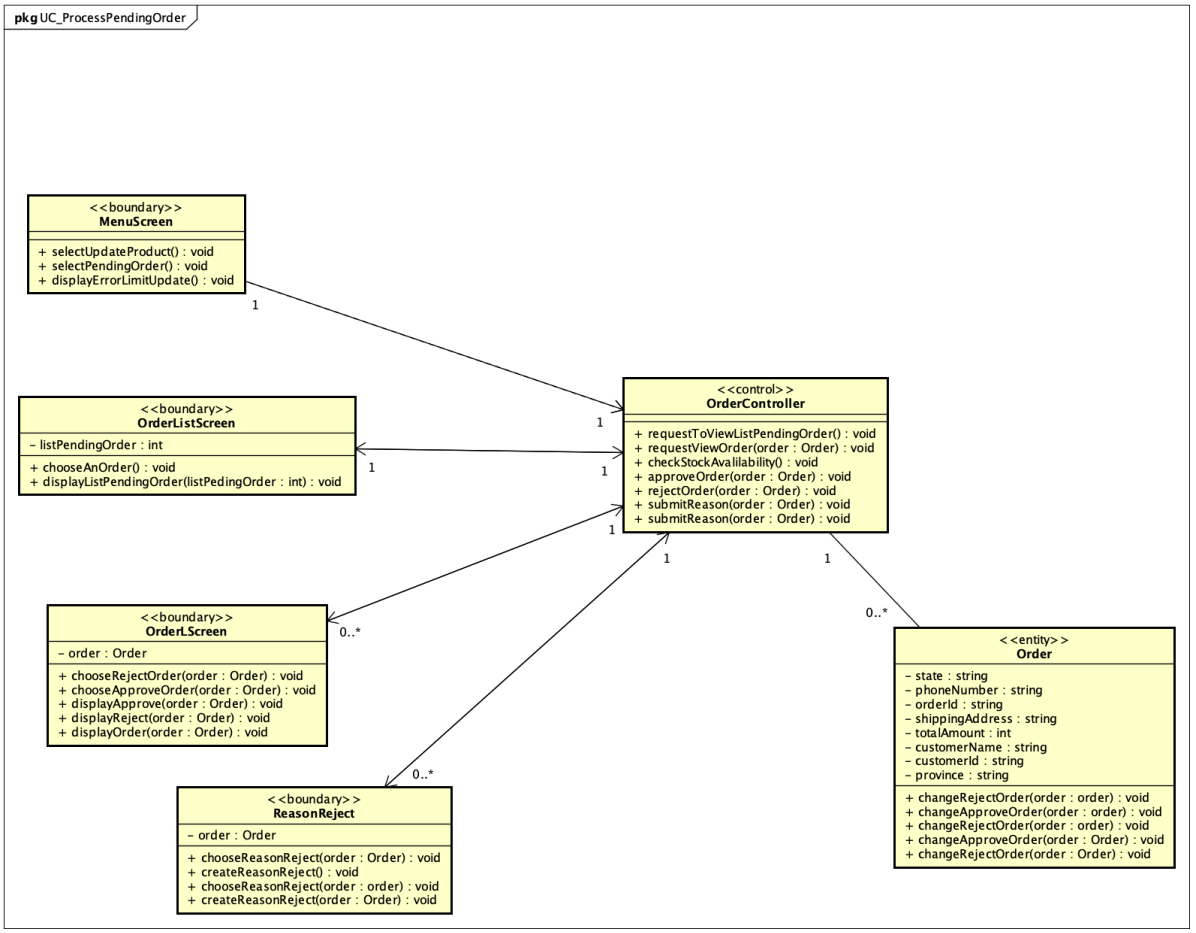
3.3.7. Place Order



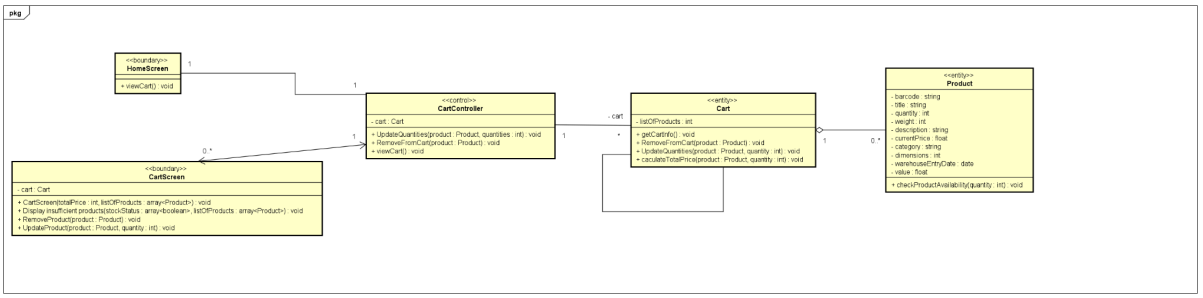
3.3.8. Place Rush Order



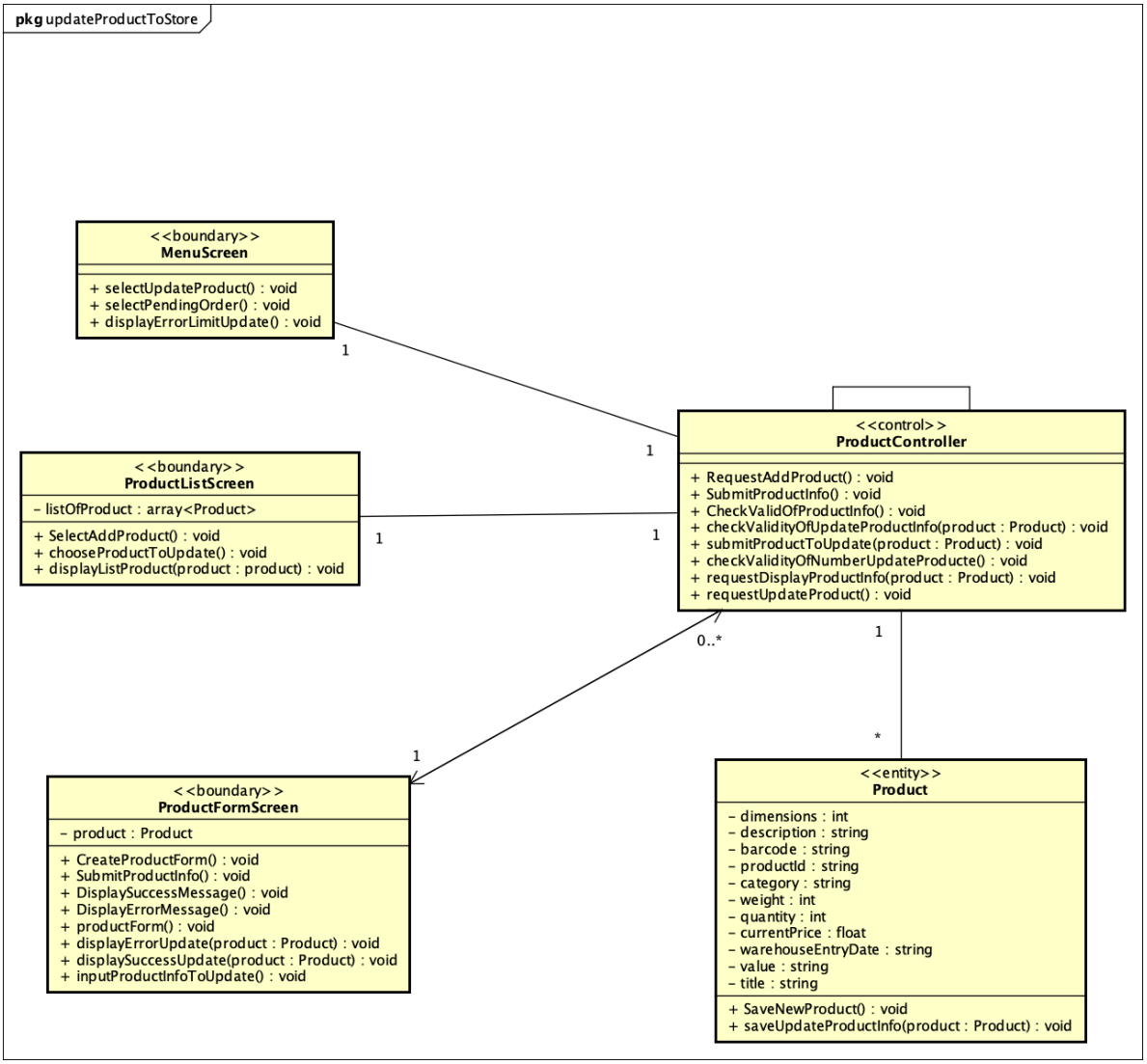
3.3.9. Process Pending Order



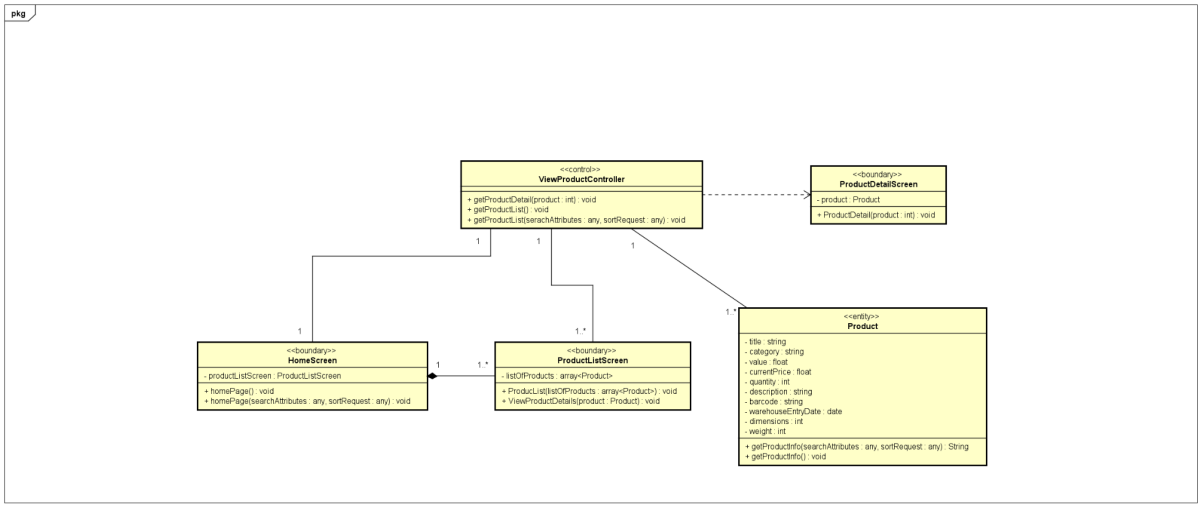
3.3.10. Update Cart



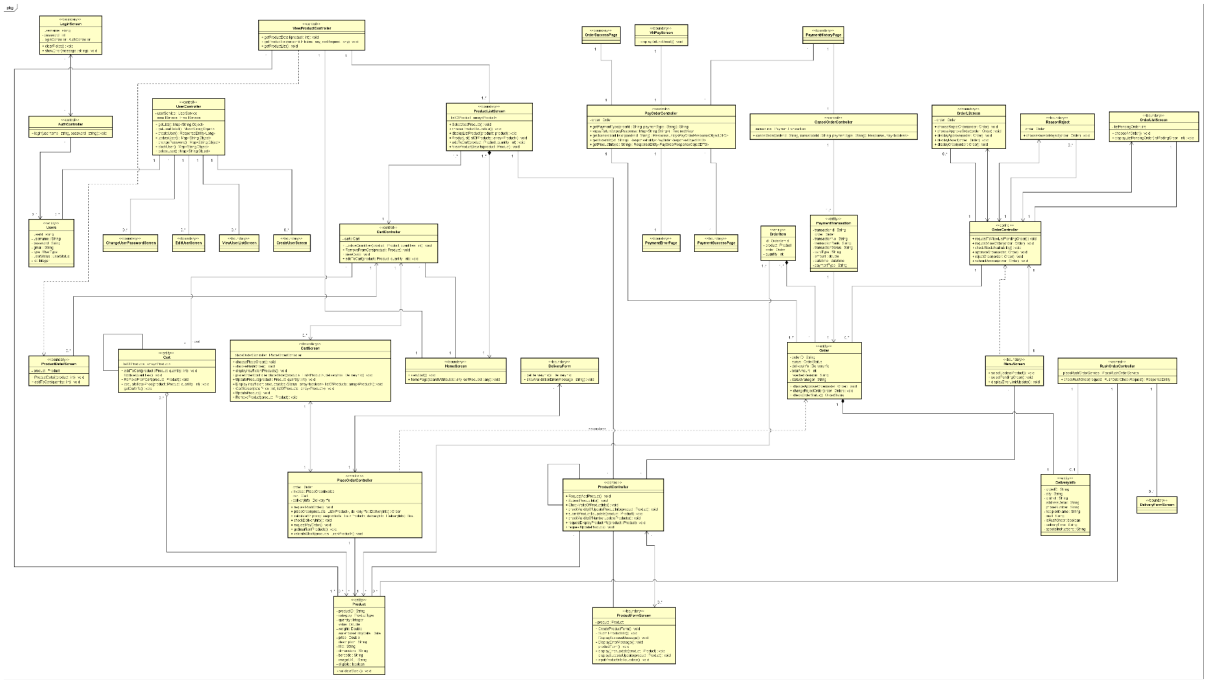
3.3.11. Update Product to Store



3.3.12. View Product Details



3.4 Unified Analysis Class Diagram



3.5 Security Software Architecture

The AIMS system's security architecture is designed to protect user data, ensure secure transactions, and maintain system integrity through a multi-layered approach encompassing authentication, authorization, encryption, and comprehensive logging.3.5.1 Authentication Architecture

The system utilizes a **Token-Based Authentication** mechanism, specifically **JSON Web Tokens (JWT)**, implemented using Spring Security for the backend and integrated into the ReactJS frontend.

- **Components:**
    1. **User Credentials (Frontend):** Users provide their `username` (or `email`) and `password` via the login interface.
    2. **Authentication Controller (Backend - `AuthController`):** Receives login requests.
    3. **Authentication Manager (Spring Security):** Delegates the authentication process to an `UserDetailsService`.
    4. **`UserDetailsService` (Custom Implementation - `UserDetailsServiceImpl`):** Fetches user details from the database based on the provided `username`. It loads `UserDetails` objects which contain user roles and password hashes.
    5. **Password Encoder (Spring Security - `BCryptPasswordEncoder`):** Securely hashes and verifies passwords.
    6. **JWT Provider/Generator:** Upon successful authentication, a JWT is generated. This token contains claims (e.g., user ID, roles, expiration time) and is digitally signed by the server's secret key.
    7. **JWT Filter (Spring Security):** Intercepts incoming requests. It extracts the

JWT from the `Authorization` header, validates its signature, and parses the claims to establish the user's security context for the current request.

- **Authentication Flow:**
    1. A user attempts to log in by sending their credentials to the `/api/auth/login` endpoint.
    2. The `AuthController` passes these credentials to Spring Security's `AuthenticationManager`.
    3. The `UserDetailsService` retrieves the user's hashed password and roles from the `Users` table in PostgreSQL.
    4. The `BCryptPasswordEncoder` compares the provided password with the stored hashed password.
    5. If credentials are valid, the `AuthenticationManager` grants authentication.
    6. A JWT is generated by the server and sent back to the client in the `JwtResponseDTO`.
    7. For subsequent requests, the client includes this JWT in the `Authorization: Bearer <token>` header.
    8. The JWT Filter validates the token, extracts user information, and sets the user's security context, allowing access to protected resources.

3.5.2 Authorization Architecture

Authorization in AIMS is achieved through **Role-Based Access Control (RBAC)**, integrated with Spring Security's annotation-based authorization.

- **Components:**
    - **User Roles (`Users` table):** The `role` column in the `Users` table stores the user's assigned role (e.g., `ADMIN`, `PRODUCTMANAGER`, `CUSTOMER`).
    - **Spring Security `@PreAuthorize` Annotations:** Used on controller methods and service methods to specify which roles are permitted to access certain functionalities.
    - **Security Context:** The authenticated user's roles (extracted from the JWT) are stored in Spring Security's `SecurityContextHolder`.
- **Authorization Flow:**
    - After a user is authenticated, their roles (e.g., from the `jwt.getRoles()`) are populated into the `UserDetailsImpl` object and stored in the `SecurityContextHolder`.
    - When a request hits a protected endpoint (e.g., `/api/manager/products`), Spring Security checks the `@PreAuthorize` annotation on the corresponding controller method (e.g., `@PreAuthorize("hasRole('PRODUCTMANAGER') or hasRole('ADMIN')")`).
    - If the user's roles in the security context match the required roles, access is granted. Otherwise, an `AccessDeniedException` is thrown, resulting in an HTTP 403 Forbidden response.
    - **Examples:**
        - `ManagerController` methods (`createProduct`, `updateProduct`, `deleteProduct`) are protected to allow only `PRODUCTMANAGER` and `ADMIN` roles.
        - `UserController` methods (`getList`, `blockUser`, `deleteUser`) are

restricted to `ADMIN` roles.
    - Customer-specific actions like `addToCart` or `placeOrder` might require the `CUSTOMER` role or be accessible to any authenticated user.

3.5.3 Encryption Protocol

Encryption is applied at multiple layers to protect sensitive data both in transit and at rest.

- **Data in Transit (HTTPS/SSL/TLS):**
    - All communication between the frontend (ReactJS) and the backend (Spring Boot) is secured using **HTTPS (Hypertext Transfer Protocol Secure)**. This relies on **SSL/TLS (Secure Sockets Layer/Transport Layer Security)** certificates.
    - **Reason:** HTTPS encrypts data packets exchanged over the network, preventing eavesdropping, tampering, and message forgery during transmission. This is critical for protecting login credentials, payment information, and personal data.
- **Data at Rest (Database Encryption):**
    - Sensitive information stored in the PostgreSQL database, such as user passwords, is **hashed** using **BCrypt**.
    - **BCrypt:** A strong, adaptive hashing algorithm designed to be computationally intensive, making brute-force attacks and rainbow table attacks impractical. It incorporates a salt to prevent identical passwords from having the same hash.
    - **Reason:** Storing only hashed passwords (instead of plain text) ensures that even if the database is compromised, user passwords cannot be easily recovered.
- **JWT Signature:**
    - JWTs are digitally signed using a **HMAC SHA-256 algorithm** with a secret key known only to the server.
    - **Reason:** This signature ensures the integrity and authenticity of the token. Any tampering with the token's payload can be detected, and only tokens signed by the legitimate server are considered valid.

4 Detailed Design

4.1 User Interface Design

<Suppose that you design a Graphical User Interface (GUI)>

4.1.1 Screen Configuration Standardization

4.1.1.1 Layout and Navigation

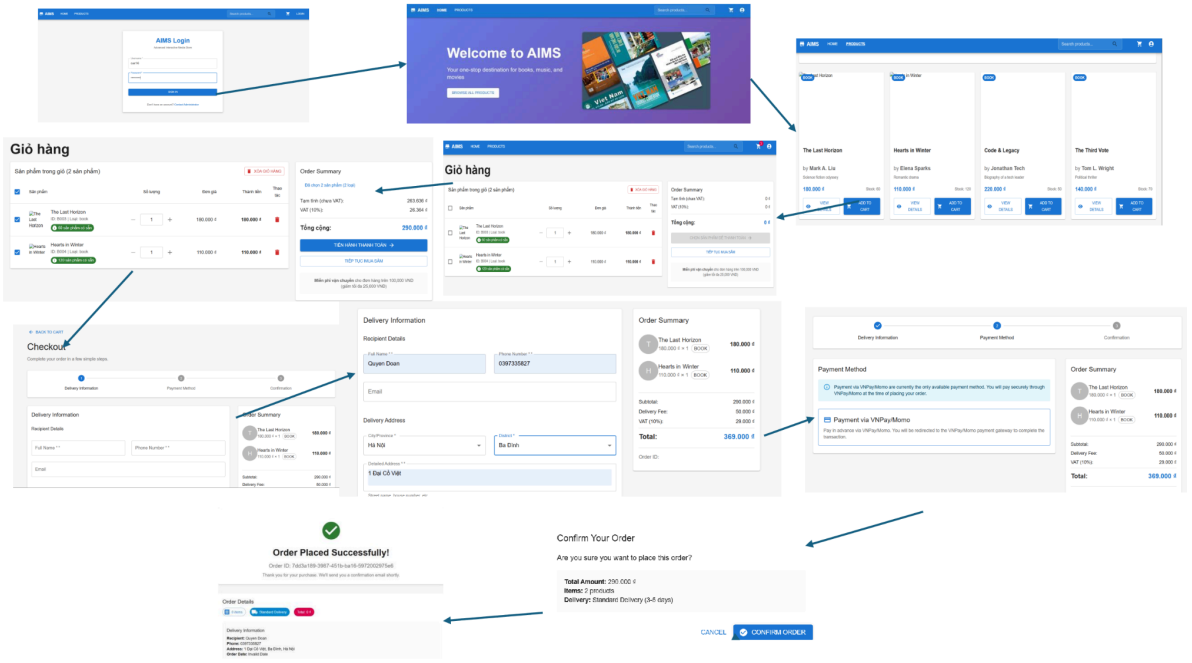
- Header and Navbar
    - The header includes the logo, navigation menu, and user menu (as seen in the layout components).
    - The navigation bar is fixed at the top, ensuring consistent access across all pages.
    - A search bar is present for quick product lookup.
- Main Content Area
    - The main content is wrapped in a `<Container>` with a maximum width of `lg` (typically 1280px or 1560px, depending on the screen).
    - Text alignment defaults to left for readability.
    - Sections such as hero, categories, featured products, and statistics are clearly separated with consistent spacing.

4.1.1.2 Design Standards

- Typography
    - The project uses the "Inter" font family.
    - Normal text uses font weight 400, while headers use font weight 600 or bold for emphasis.
    - Typography components from MUI are used for consistent text styling.
- Color Scheme
    - The color palette includes primary colors (e.g., blue gradients in the hero section) and neutral backgrounds (`grey.50`).
    - Sufficient contrast is maintained for readability (e.g., white text on colored backgrounds).
- Icons and Images
    - SVG icons are used throughout the UI for scalability and consistency.
    - All images include descriptive `alt` text for accessibility (e.g., `alt="AIMS Store"`).
    - Category cards and product images are displayed with proper aspect ratios and rounded corners.
- Spacing and Alignment
    - Consistent padding and margins are applied using MUI's `sx` prop and grid system.
    - The layout uses a responsive grid system (`Grid` component) to align content and ensure adaptability across devices.
    - Cards and sections have uniform spacing, and hover effects are used for interactivity.

4.1.2 Screen Transition Diagrams

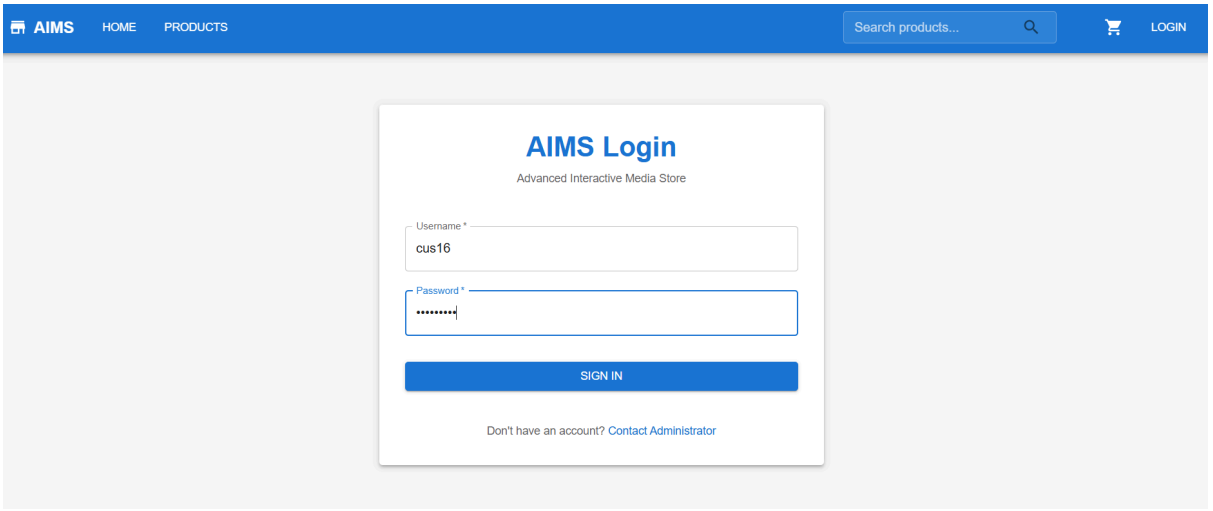
Below is an example of a screen transition diagram from login to placing a successful order.



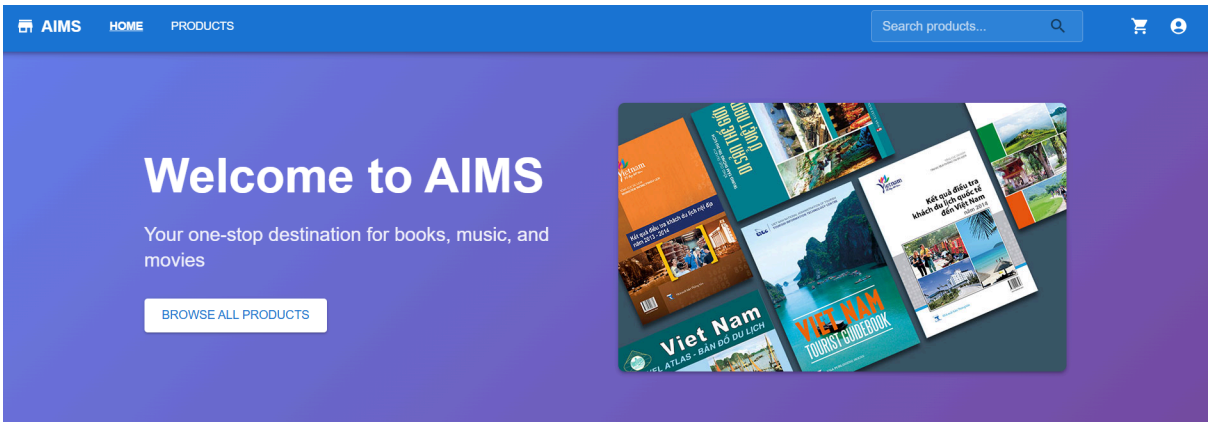
4.1.3 Screen Specifications

<Screen images should be included in the screen specifications>

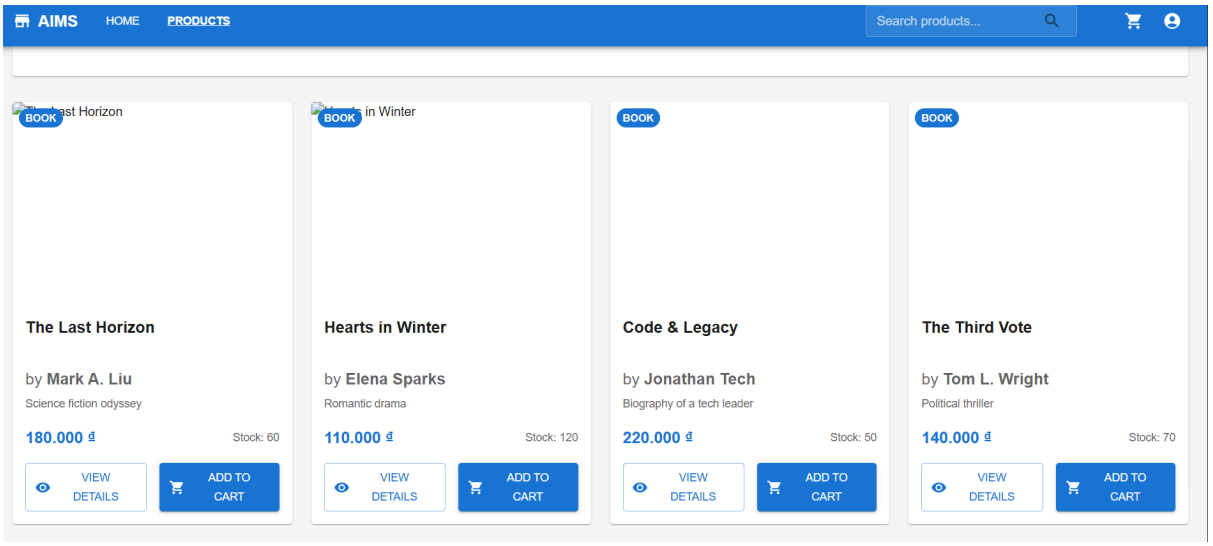
4.1.3.1 Login Screen



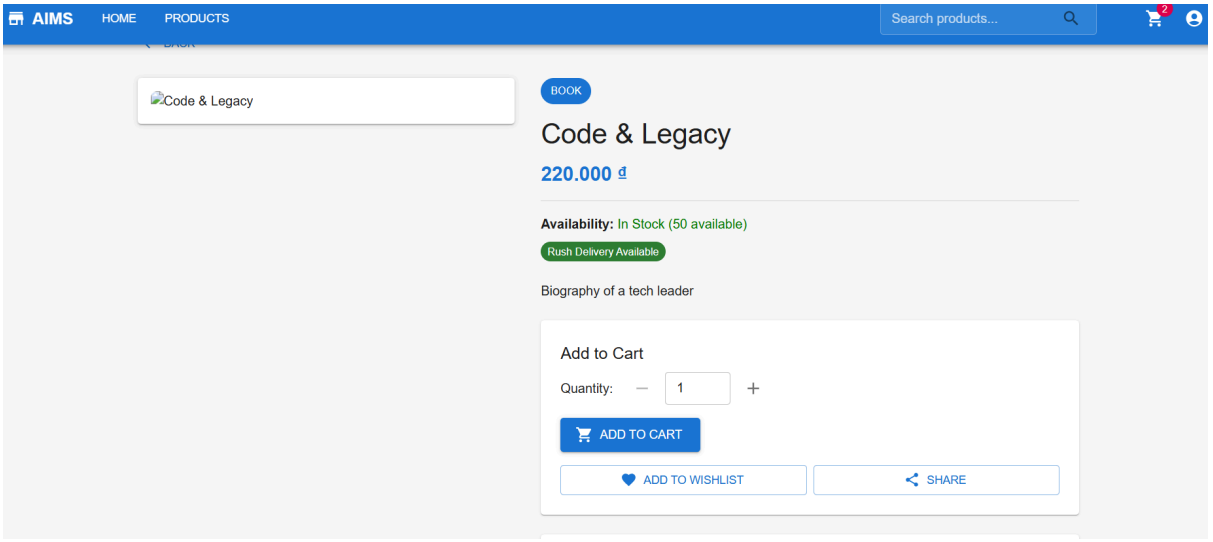
4.1.3.2 Homepage Screen



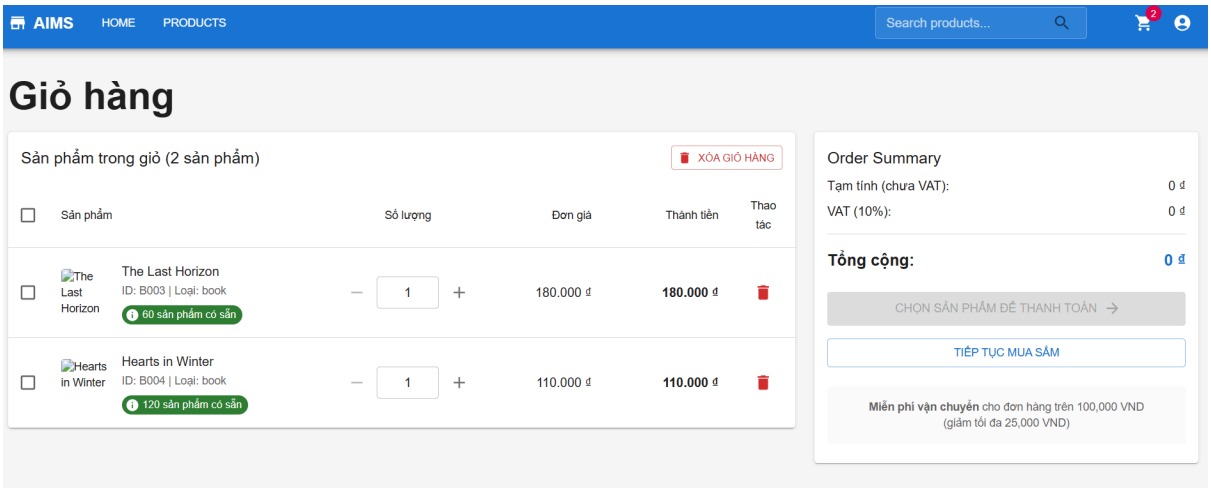
4.1.3.3 Browse Products Screen



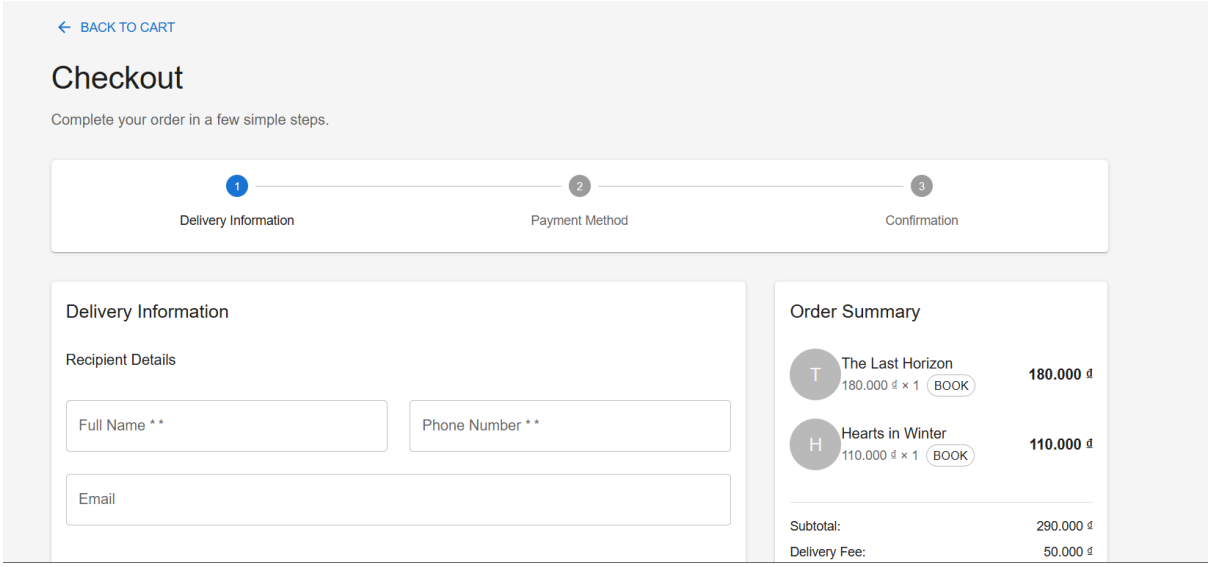
4.1.3.4 View Product Details Screen



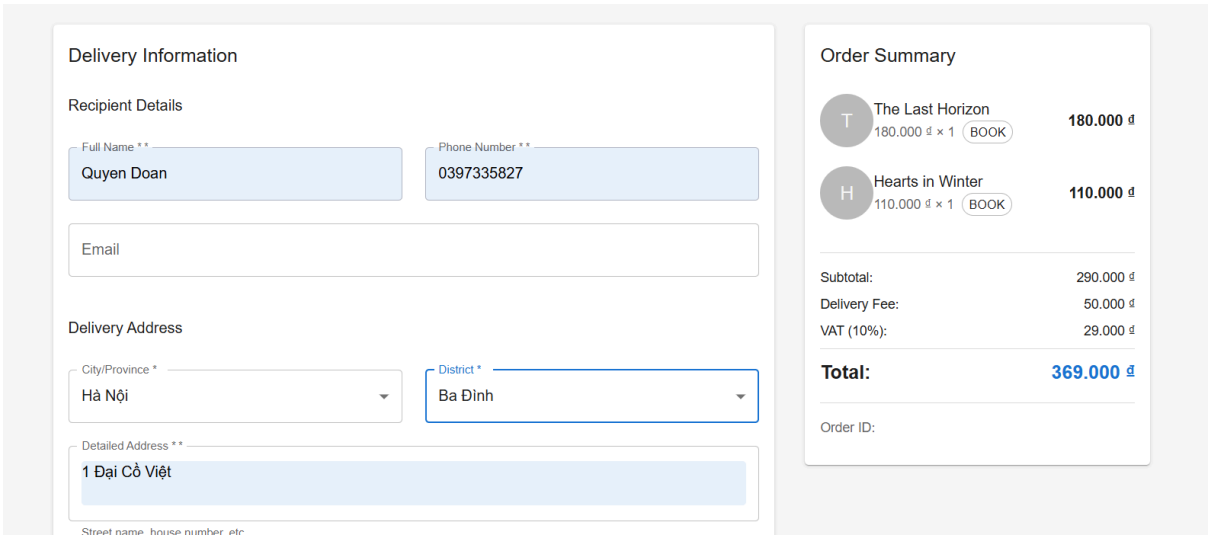
4.1.3.5 Cart Screen



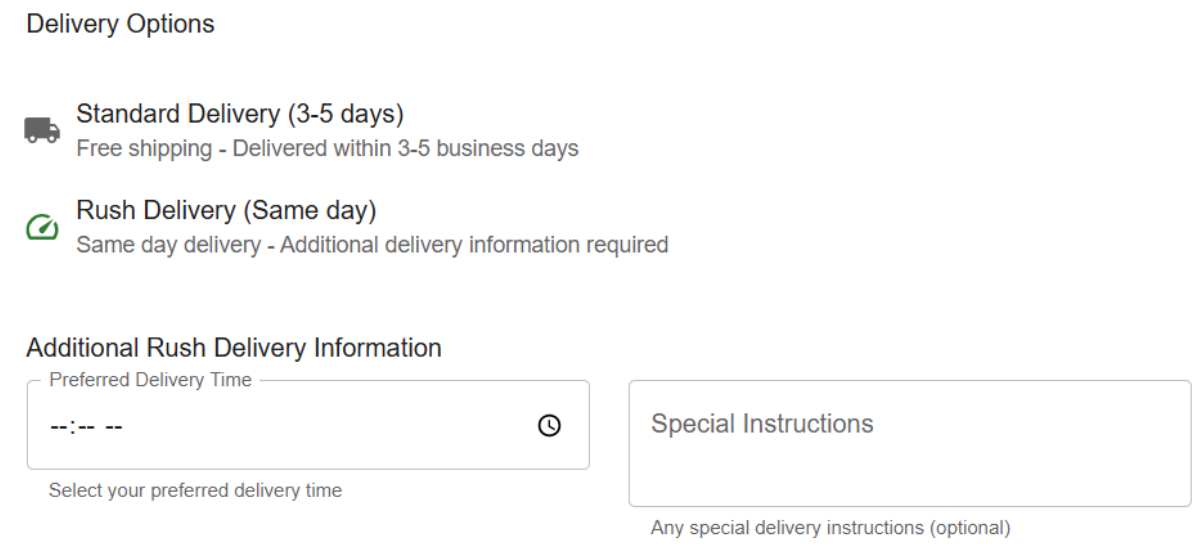
4.1.3.6 CheckOut Screen

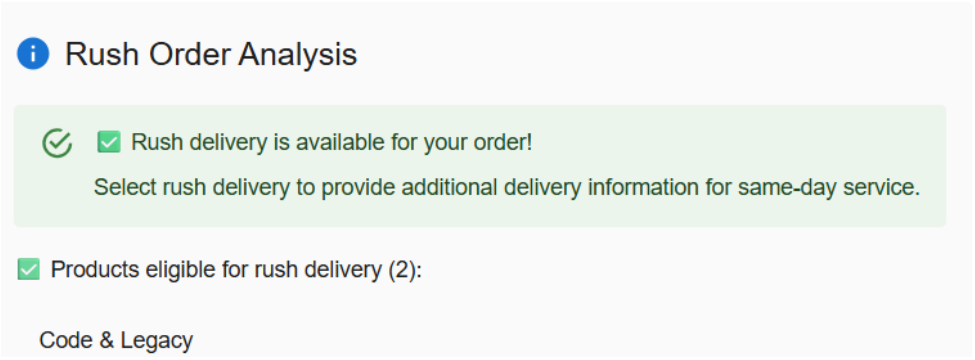


4.1.3.7 Delivery Form Screen

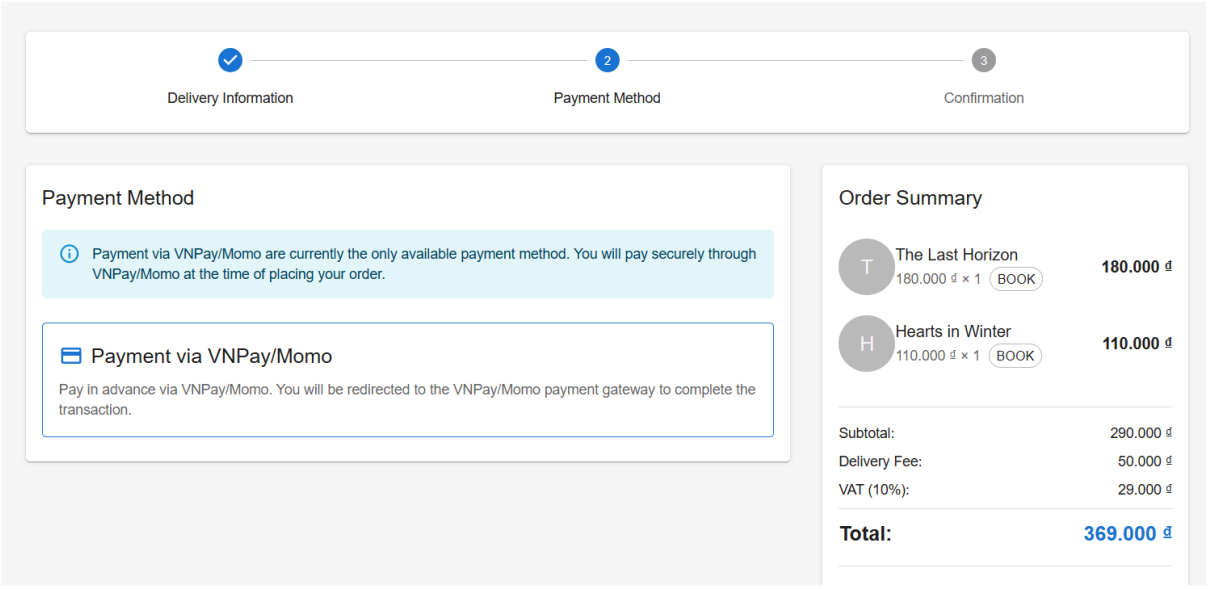


4.1.3.8 Place Rush Order Screen

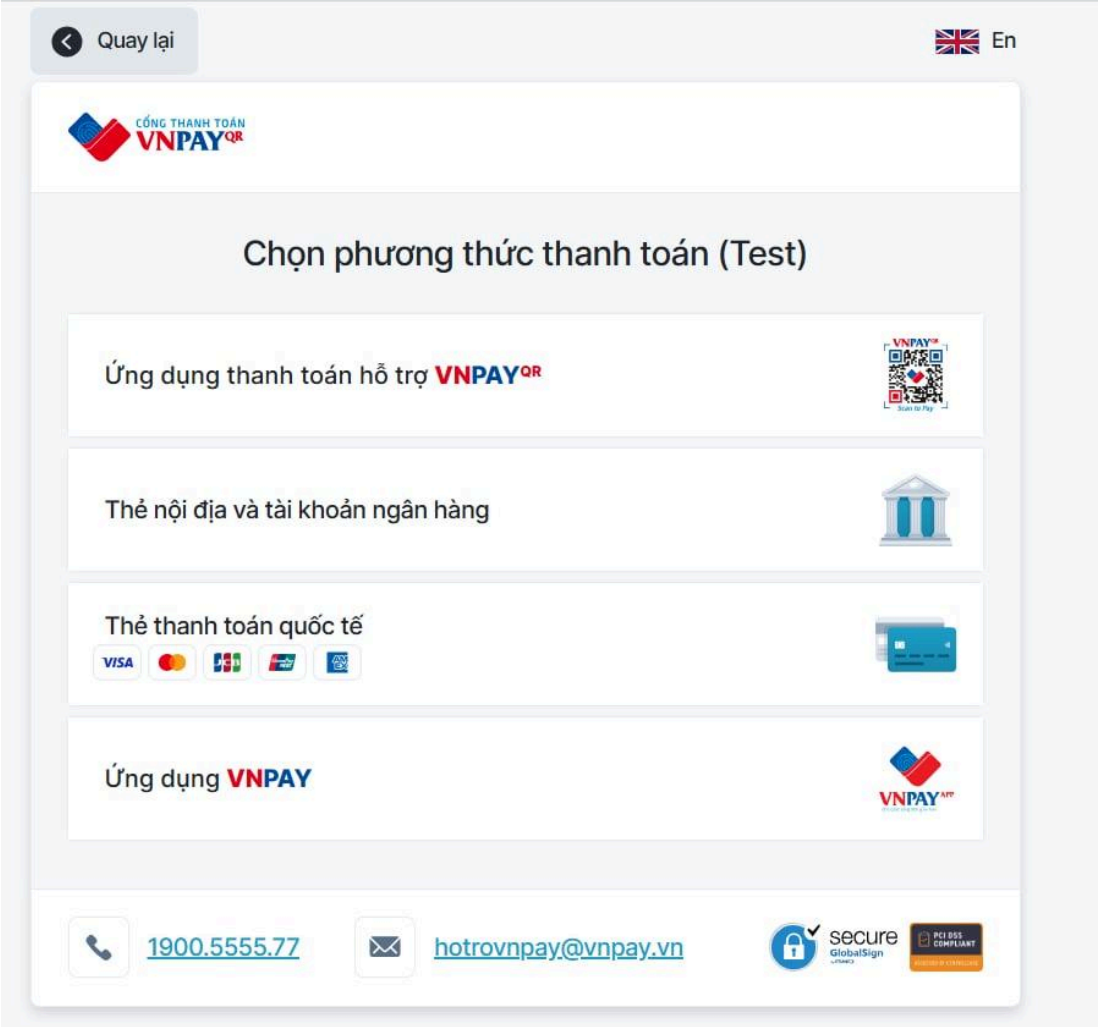




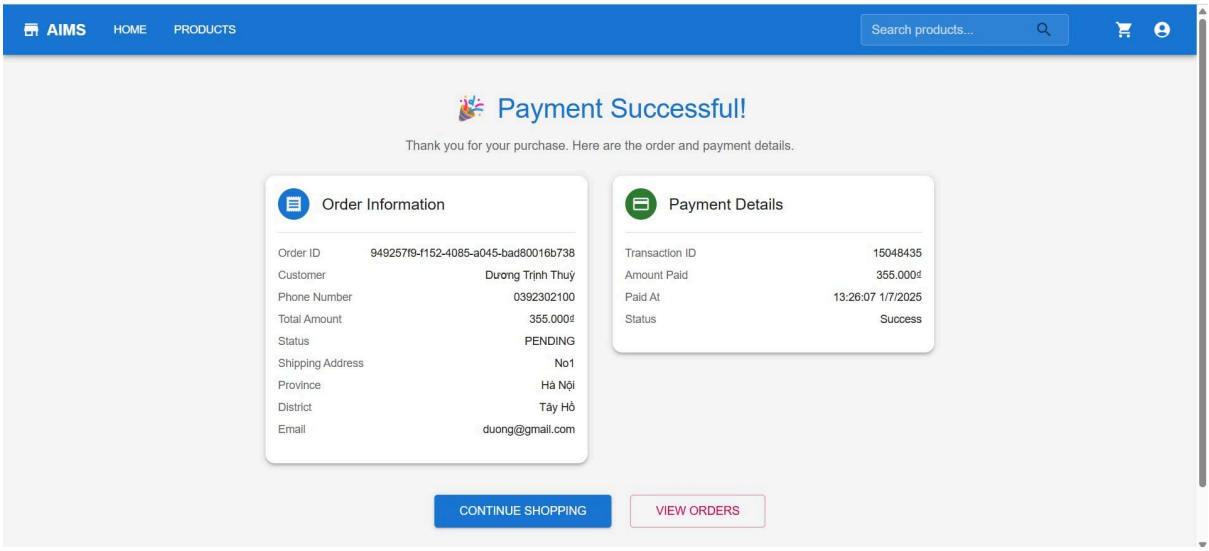
4.1.3.9 Payment Screen



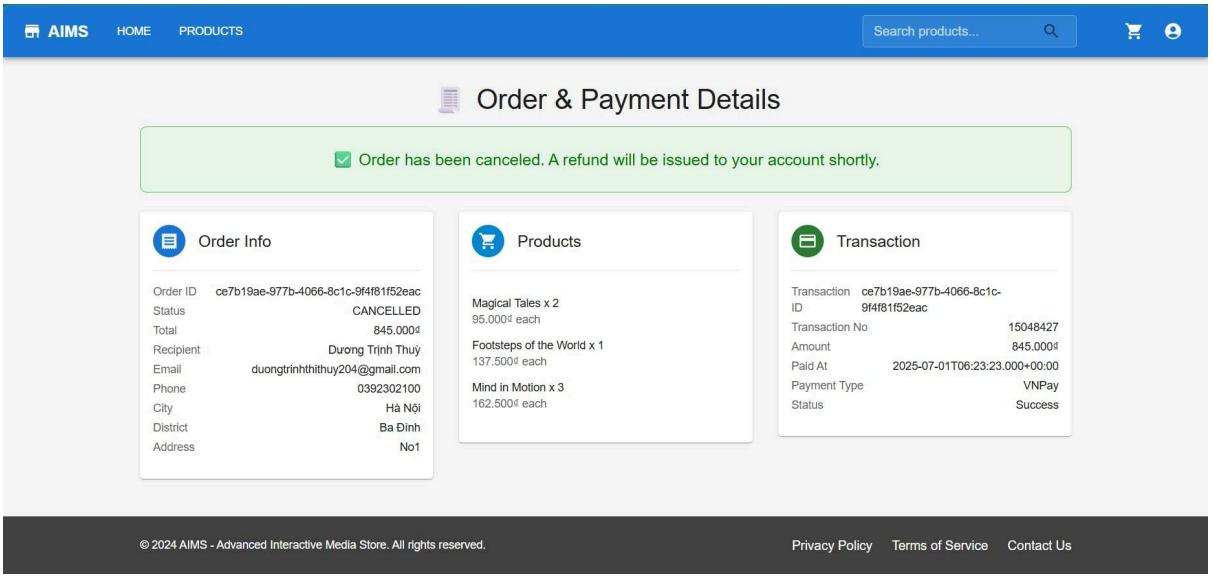
4.1.3.10 Payment Method Screen



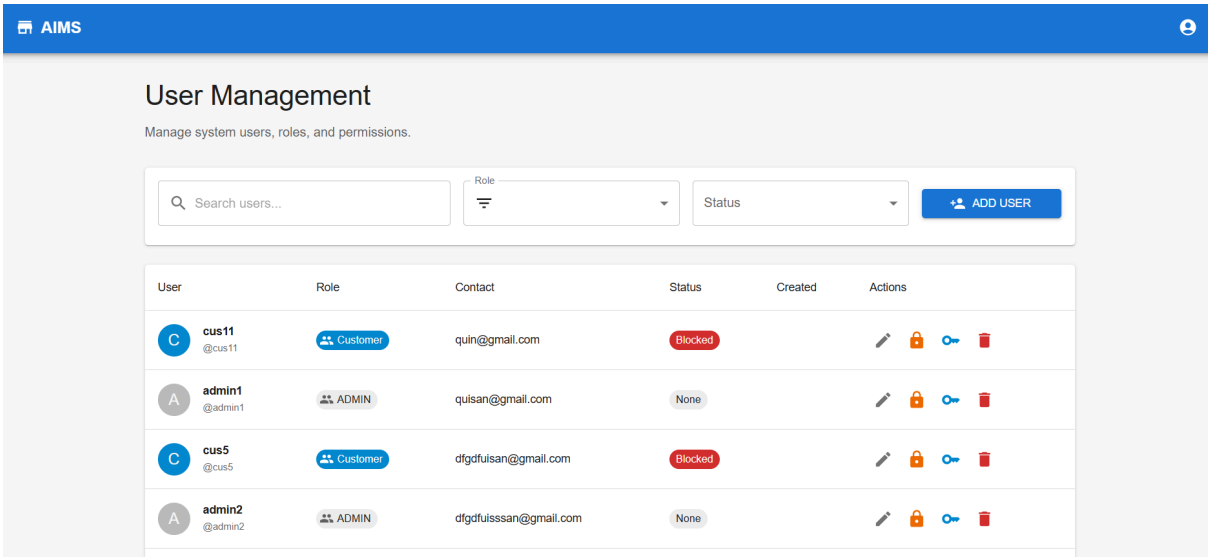
4.1.3.11 Payment Success Screen



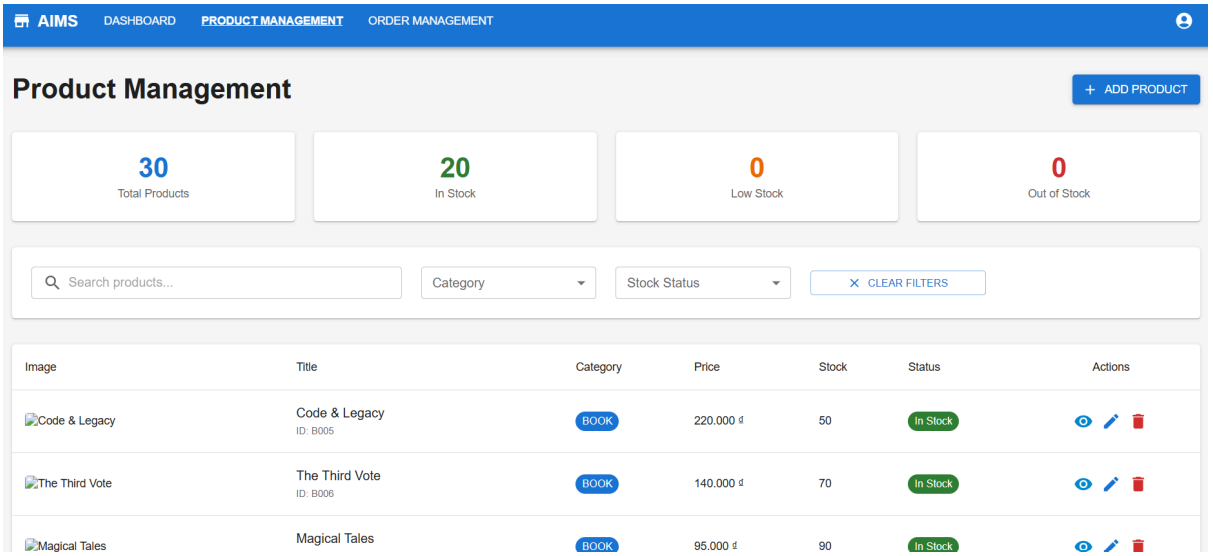
4.1.3.12 Cancel Order Screen



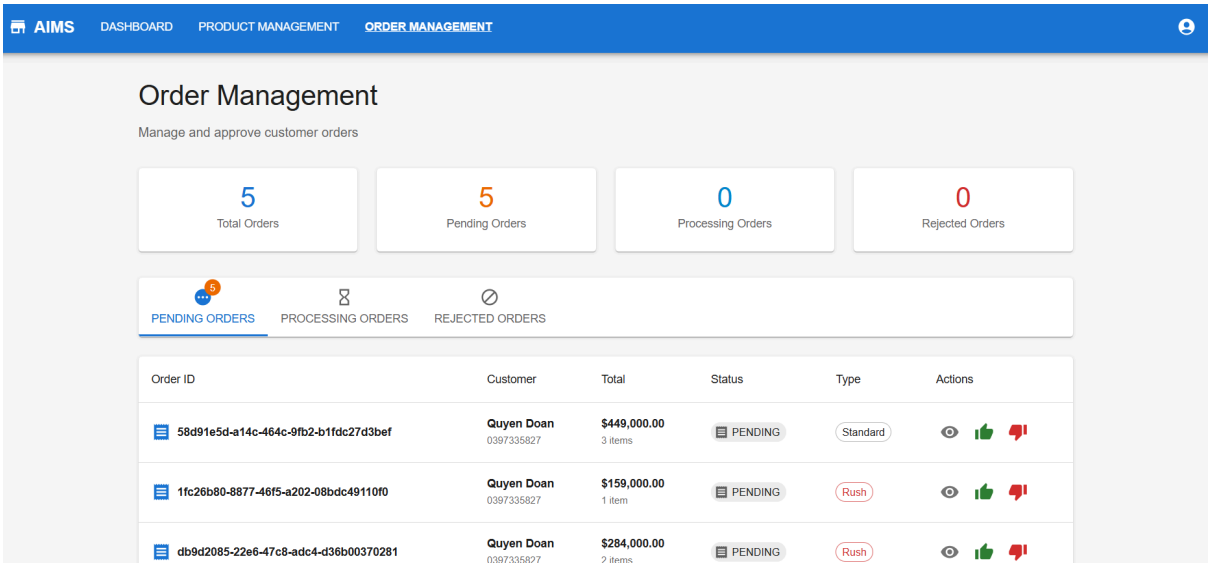
4.1.3.13 Admin User Management Screen



4.1.3.14 Product Management Screen



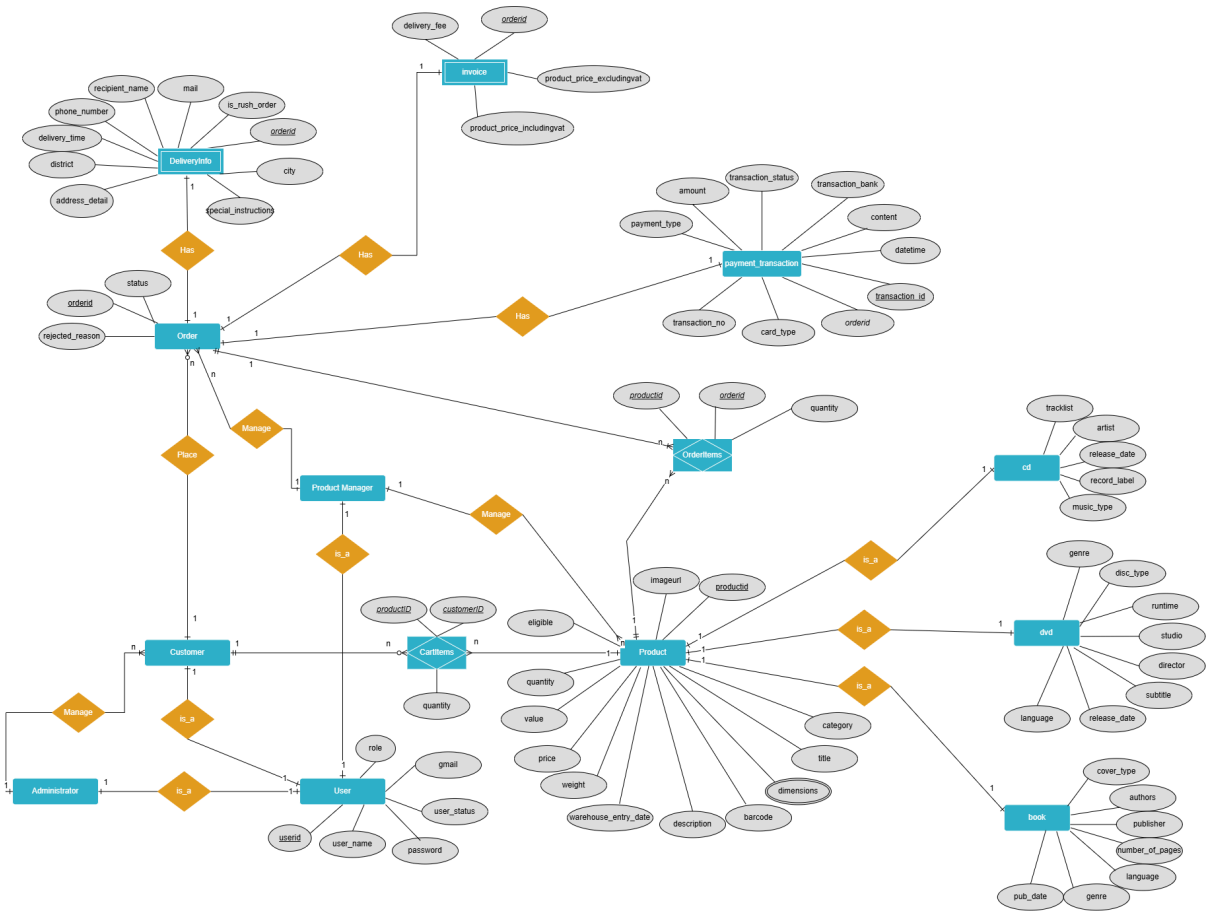
4.1.13.14 Order Management Screen



4.2 Data Modeling

4.2.1 Conceptual Data Modeling

<E-R Diagram image and description of entities and relationships>



Entities (Rectangles):

- **DeliveryInfo:** Contains information related to deliveries.
  - Attributes: recipient_name, mail, id_main_order, phone_number, country, delivery_date, district, address_detail, city, postal_instruction.
- **Order:** Represents an order placed by a customer.
  - Attributes: status, amount, reject_reason, rejected_reason.
- **Payment_transaction:** Details of a payment transaction.
  - Attributes: transaction_status, transaction_item, amount, payment_type, content, datetime, transaction_no, card_type, provider, bank_transaction_id.
- **OrderItems:** Links orders to products.
  - Attributes: productid, orderid, quantity.
- **Product:** Information about a specific product.
  - Attributes: id_product, imgsmall, imgmedium, imglarge, price, weight, warranty_period_days, description, barcode, category, dimension.
- **User:** User account information.
  - Attributes: user_name, password, user_status, email.
- **Administrator:** An administrator user (a role of User).
- **Customer**: A customer user (a role of User).
- **Product Manager:** A product manager user (a role of User).
- **Invoice:** Represents an invoice for an order.
  - Attributes: delivery_fee, product_price_excludingtax, product_price_includingtax.

- **CartItem:** Items in a shopping cart.
    - Attributes: id_a, product_id, quantity.
- **CD:** Specific details for CD products.
    - Attributes: tracklist, edition, release_date, record_label, music_type.
- **DVD:** Specific details for DVD products.
    - Attributes: genres, disc_type, runtime, studio, director, subtitle, language, release_date, cover_type.
- **Book:** Specific details for Book products.
    - Attributes: culture, publisher, number_of_pages, language, genre.

Relationships (Diamonds and Connecting Lines):

- **Has (Order - DeliveryInfo):** An Order "Has" DeliveryInfo.
    - Cardinality: 1 Order to 1 DeliveryInfo.
- **Has (Order - Payment_transaction):** An Order "Has" a Payment_transaction.
    - Cardinality: 1 Order to 1 Payment_transaction.
- **Includes (Order - OrderItems):** An Order "Includes" OrderItems.
    - Cardinality: 1 Order to many OrderItems.
- **Includes (OrderItems - Product):** OrderItems "Includes" a Product.
    - Cardinality: many OrderItems to 1 Product.
- **Manages (Customer - Order):** A Customer "Manages" an Order.
    - Cardinality: 1 Customer to many Orders.
- **Manages (Product Manager - Product):** A Product Manager "Manages" a Product.
    - Cardinality: 1 Product Manager to many Products.
- **Manages (Administrator - User):** An Administrator "Manages" a User.
    - Cardinality: 1 Administrator to many Users.
- **Has (Customer - User):** A Customer "Has" a User account.
    - Cardinality: 1 Customer to 1 User.
- **Has (Customer - CartItem):** A Customer "Has" CartItems.
    - Cardinality: 1 Customer to many CartItems.
- **Has (Product - CartItem):** A Product "Has" CartItems (i.e., a product can be in many cart items).
    - Cardinality: 1 Product to many CartItems.
- **Has (Order - Invoice):** An Order "Has" an Invoice.
    - Cardinality: 1 Order to 1 Invoice.
- **Is_a (CD - Product):** CD "Is_a" Product (Inheritance).
- **Is_a (DVD - Product):** DVD "Is_a" Product (Inheritance).
- **Is_a (Book - Product):** Book "Is_a" Product (Inheritance).

4.2.2 Database Design

4.2.2.1 Database Management System

Here's a revised version focusing on RDBMS in general, with a brief mention of PostgreSQL:

Based on the complex data structure and relationships depicted in the provided Entity-Relationship (E-R) Diagram, the decision to opt for a **Relational Database Management System (RDBMS)** is highly ideal. RDBMSs are powerful, reliable, and

feature-rich systems that are exceptionally well-suited for managing the types of data and structured relationships present in this model.

An RDBMS fundamentally operates on the relational model, where data is organized into tables (relations) with rows and columns, and relationships between these tables are established through keys. This paradigm is renowned for its high reliability, robust data integrity through the enforcement of constraints, strong extensibility, and strict adherence to SQL standards. RDBMSs support a wide variety of data types and offer advanced features essential for complex data management.

An RDBMS is an ideal choice for this system due to its excellent support for the relational model, ensuring robust data integrity through strong SQL constraints. It provides advanced features and extensibility, including support for complex data types, custom functions, and inheritance-like structures, enabling sophisticated business logic implementation. RDBMSs also excel in concurrency handling and ACID-compliant transaction management, guaranteeing data reliability and consistency under any load. With high performance, flexible scaling options (both vertical and horizontal), proven stability, and extensive industry support, an RDBMS offers a reliable and cost-effective solution. **PostgreSQL**, for instance, stands out as a prime example of an open-source RDBMS that embodies these characteristics, making it a very strong candidate for such a system.

4.2.2.2 Database Diagram

a. Process to Design a Database from an E-R Diagram

- **Step 1: Convert Strong Entities into Tables**

- Rule: Each strong entity in the E-R diagram becomes a table in the relational database. The simple attributes of the entity become columns in that table, and the Primary Key (PK) of the entity becomes the primary key of the table.
- Example:
    - The product entity (a strong entity) is directly converted into the product table. Its productid serves as the PK.
    - Similarly, users, orders, delivery_info, payment_transaction, and invoice are strong entities that become their respective tables (users, orders, delivery_info, payment_transaction, invoice) with their corresponding primary keys (id, orderid, orderid, transaction_id, orderid).

- **Step 2: Convert Weak Entities into Tables**

- Rule: Each weak entity becomes a new table. It includes all attributes of the weak entity. The primary key of the new table is a combination of the primary key of its owning strong entity (added as a Foreign Key - FK) and the discriminator attribute of the weak entity (if any).
- Example: The tables like delivery_info, payment_transaction, and invoice have orderid as their PK, FK, implying a strong dependency on the orders table. While not strictly "weak entities" in the E-R diagram, their design reflects this dependency.

- **Step 3: Convert 1:1 (One-to-One) Relationships**

- Rule: There are two common ways to handle this:
    - Merge Tables: If the relationship is mandatory on both sides, attributes and keys of one entity can be merged into the table of the other entity.

    - Add Foreign Key: Place the primary key of one entity as a foreign key (FK) into the table of the other entity. This foreign key can also be unique (UNIQUE) if the relationship is strictly 1:1.
- Example:
    - The relationship between orders and delivery_info is 1:1. The delivery_info table uses orderid as both its PK and FK referencing orders.orderid. This ensures that each order has exactly one delivery information record and vice-versa.
    - Similarly, payment_transaction (though transaction_id is its PK, orderid is FK, implying one order can have one main transaction or multiple attempts, which deviates slightly from strict 1:1 if orderid isn't unique in payment_transaction as a whole), and invoice (with orderid as PK, FK) also demonstrate 1:1 relationships with the orders table.

- **Step 4: Convert 1:N (One-to-Many) Relationships**

- Rule: Add the primary key of the entity on the "one" side as a foreign key (FK) into the table of the entity on the "many" side.
- Example:
    - The relationship between product (one product) and order_items (many order items can contain that product): The order_items table has a productid column which is part of its composite PK and also an FK referencing product.productid.

- **Step 5: Convert N:M (Many-to-Many) Relationships**

- Rule: Create a new table (an intersection table) to represent this relationship. This new table will include the primary keys of both related entities as foreign keys. The primary key of the intersection table is often a composite of these two foreign keys. If the relationship has its own attributes, those attributes will become columns in the intersection table.
- Example:
    - The relationship between orders (many) and product (many) is handled by the order_items table. order_items has a composite primary key made of orderid (FK from orders) and productid (FK from product). It also includes the quantity attribute, which is specific to this relationship.
    - Similarly, the cart_items table handles the N:M relationship between users (customers) and products. It has a composite primary key of customerid (FK from users) and productid (FK from product), along with the quantity attribute.

- **Step 6: Convert Multivalued Attributes**

- Rule: Create a new table for the multivalued attribute. This new table will have a column for the multivalued attribute and a foreign key referencing the primary key of the entity that contains that attribute. The primary key of the new table is often a composite of the foreign key and the multivalued attribute.

- **Step 7: Convert Composite Attributes**

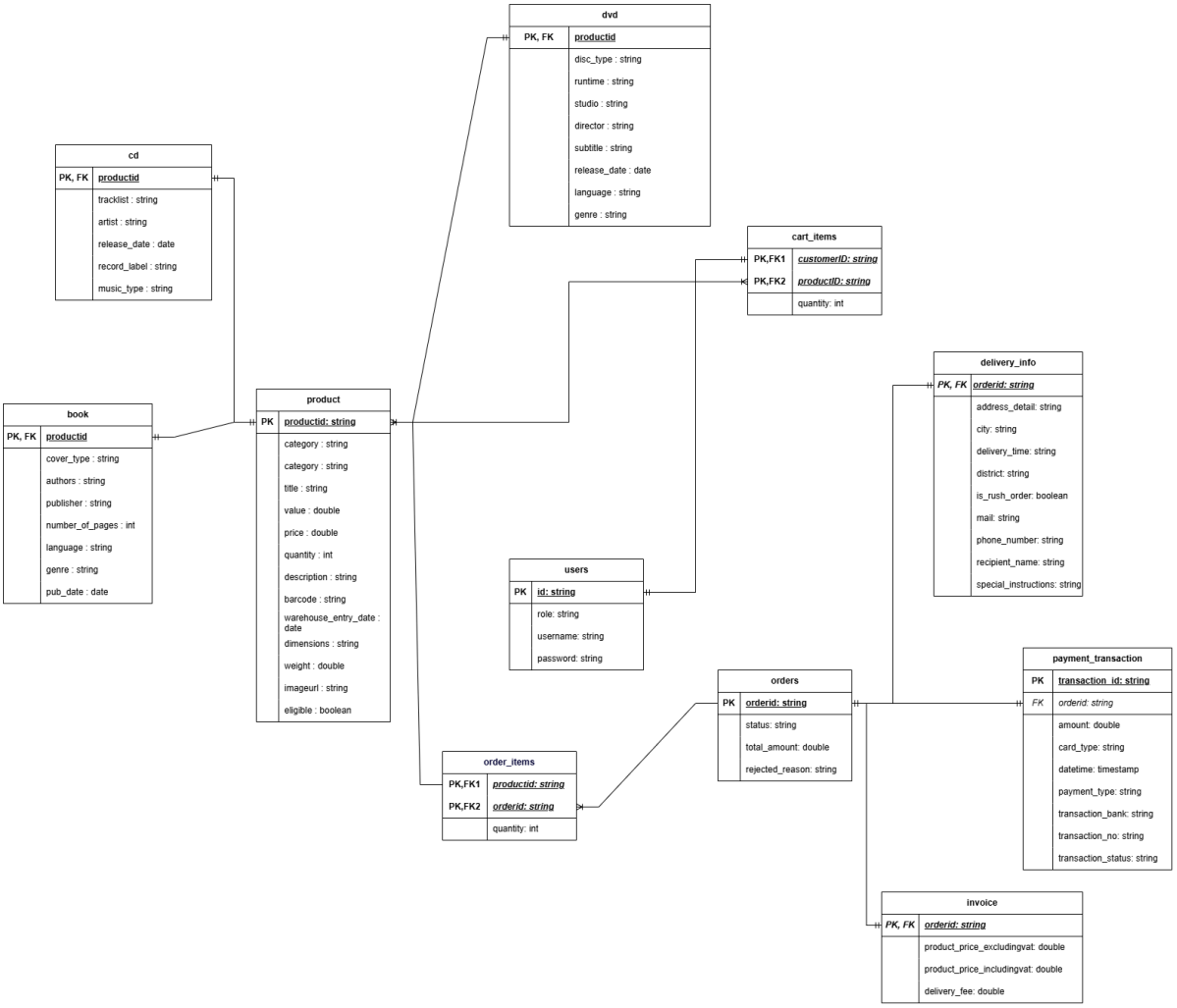
- Rule: Decompose the composite attribute into its individual component attributes, which then become separate columns in the table.

- **Step 8: Convert Generalization/Specialization ("Is_a" relationships)**

- Rule: The most common and effective approach is to create a table for the superclass and separate tables for each subclass. Each subclass table has its primary key being a foreign key referencing the primary key of the superclass table.
- Example:

- The product table serves as the superclass, containing common attributes.
- The cd, dvd, and book tables are subclasses. Each of these tables uses productid as its primary key (PK) and also as a foreign key (FK) referencing product.productid. This design correctly implements the "Is_a" relationship, allowing each specific product type to have its unique attributes while being linked to the general product information.

b. **Relational Database Schema Diagram**



4.2.2.3 Database Detail Design

Table 1. Users table

#	PK	FK	Column name	Data type	Default value	Mandatory	Description

1	x		userid	integer	nextval('users_userid_seq'::regclass)	Yes	Unique identifier for each user (Primary Key)
2			gmail	character varying(255)		Yes	User's Gmail address (must be unique).
3			password	character varying(255)		Yes	User's password (must be unique — unusual, might need review).
4			role	character varying(255)			Role of the user (ADMIN, PRODUCTMANAGER, CUSTOMER)
5			user_status	character varying(255)			Status of the user (NONE, BLOCKED)
6			user_name	character varying(255)		Yes	Username (must be unique).

Table 2. Product table

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		productid	character varying(255)		Yes	Unique identifier for the product (Primary Key).
2			barcode	character varying(255)			Product barcode used for identification/scanning.

3			category	character varying(255)			Category of the product: book, cd, or dvd. Enforced by a CHECK
4			description	character varying(255)			Description of the product.
5			dimensions	character varying(255)			Physical dimensions of the product (e.g., height x width x depth).
6			eligible	boolean			Whether the product is eligible (e.g., for promotion or delivery)
7			imageurl	character varying(255)			URL to the product image.
8			price	double precision			Selling price of the product.
9			quantity	integer			Quantity of the product in stock.
10			title	character varying(255)			Title or name of the product.
11			value	double precision			Estimated or declared value of the product.

12			warehouse_entry_date	date			Date the product entered the warehouse.
13			weight	double precision			Weight of the product.

Table 3. book table

#	PK	FK	Column name	Data type	Default value	Mandatory	Description
1			authors	character varying(255)			Name(s) of the book's author(s).
2			cover_type	character varying(255)			Type of book cover (e.g., hardcover, paperback).
3			genre	character varying(255)			Genre or category of the book (e.g., fiction, non-fiction).
4			language	character varying(255)			Language in which the book is written.
5			number_of_pages	integer			Total number of pages in the book.
6			pub_date	date			Publication date of the book.
7			publisher	character varying(255)			Publisher of the book.

8	x	x	productid	character varying(255)		Yes	Unique identifier linking to the product table (Primary Key and Foreign Key).

Table 4. cd table

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1			artist	character varying(255)			Name of the artist or band.
2			music_type	character varying(255)			Genre or type of music (e.g., pop, rock, classical).
3			record_label	character varying(255)			Record label that published the CD.
4			release_date	date			Official release date of the CD.
5			tracklist	character varying(255)			A list of music tracks on the CD (often as a single string).
6	x	x	productid	character varying(255)		Yes	Unique identifier linking to the product table (Primary Key and Foreign Key).

Table 5. dvd table

#	PK	FK	Column name	Data type	Default value	Mandatory	Description
1			director	character varying(255)			Name of the DVD's director.
2			disc_type	character varying(255)			Type of disc (e.g., single-layer, dual-layer, Blu-ray).
3			genre	character varying(255)			Genre of the DVD (e.g., action, drama, documentary).
4			language	character varying(255)			Language of the movie or video content.
5			release_date	date			Release date of the DVD.
6			runtime	character varying(255)			Duration of the video content (e.g., 120 minutes).
7			studio	character varying(255)			Studio or company that produced the DVD.
8			subtitlr	character varying(255)			Subtitle language(s) available on the DVD.
9	x	x	productid	character varying(255)		Yes	Unique identifier linking to the product table (Primary Key and Foreign Key).

Table 6. orders table

#	PK	FK	Column name	Data type	Default value	Mandatory	Description
1	x		orderid	character varying(255)		x	Unique identifier for the order (Primary Key).
2			rejected_reason	character varying(255)			Reason why the order was rejected (if applicable).
3			status	character varying(255)			Current status of the order. Must be one of the predefined values.
4			total_amount	double precision			Total monetary amount of the order.

Table 7. order_items table

#	PK	FK	Column name	Data type	Default value	Mandatory	Description
1	x	x	orderid	character varying(255)		x	Identifier of the associated order (Primary Key & Foreign Key to orders)
2	x	x	productid	character varying(255)		x	Identifier of the associated product (Primary Key & Foreign Key to product).

3			quantity	integer			Quantity of the specified product in the order.

Table 8. delivery_info table

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		orderid	character varying(255)		x	Unique identifier linking to an order (Primary Key — possibly also FK).
2			address_detail	character varying(255)			Detailed address of the recipient.
3			city	character varying(255)			City of the delivery address.
4			delivery_time	character varying(255)			Preferred or estimated delivery time.
5			district	character varying(255)			District of the delivery address.
6			is_rush_order	boolean			Indicates if the order is a rush order (true/false).
7			mail	character varying(255)			Email address of the recipient.
8			phone_numbe r	character varying(255)			Phone number of the recipient.

9			recipient_name	character varying(255)			Full name of the recipient.
10			special_instructions	character varying(255)			Any additional delivery instructions.

Table 9. invoice table

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x	x	orderid	character varying(255)		x	Unique identifier linking to an order (Primary Key — possibly also FK).
2			delivery_fee	double precision			Cost of delivering the order.
3			product_price_excludingvat	double precision			Total price of products excluding VAT.
4			product_price_includingvat	double precision			Total price of products including VAT.

Table 10. cart_items table

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x	x	productid	character varying(255)		x	Identifier of the product (Primary Key and Foreign Key to product).

2	x	x	userid	integer		x	Identifier of the user (Primary Key and Foreign Key to users).
3			quantity	integer			Quantity of the product in the user's cart.

Table 11. payment_transaction table

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		transaction_id	character varying(255)		x	Unique identifier for the payment transaction (Primary Key).
2			amount	double precision		x	Total amount of the transaction.
3			card_type	character varying(255)			Type of card used for payment (e.g., VISA, MasterCard).
4			datetime	timestamp(6) without time zone		x	Date and time when the transaction occurred.
5			payment_type	character varying(255)			Type of payment (e.g., credit card, VNPay, bank transfer).
6			transaction_bank	character varying(255)			Bank involved in the transaction.

7			transaction_no	character varying(255)			External transaction number provided by the payment gateway/bank.
8			transaction_status	character varying(255)			Status of the transaction (e.g., success, failed, pending).
9		x	orderid	character varying(255)			Associated order ID (Foreign Key to orders). Must be unique.

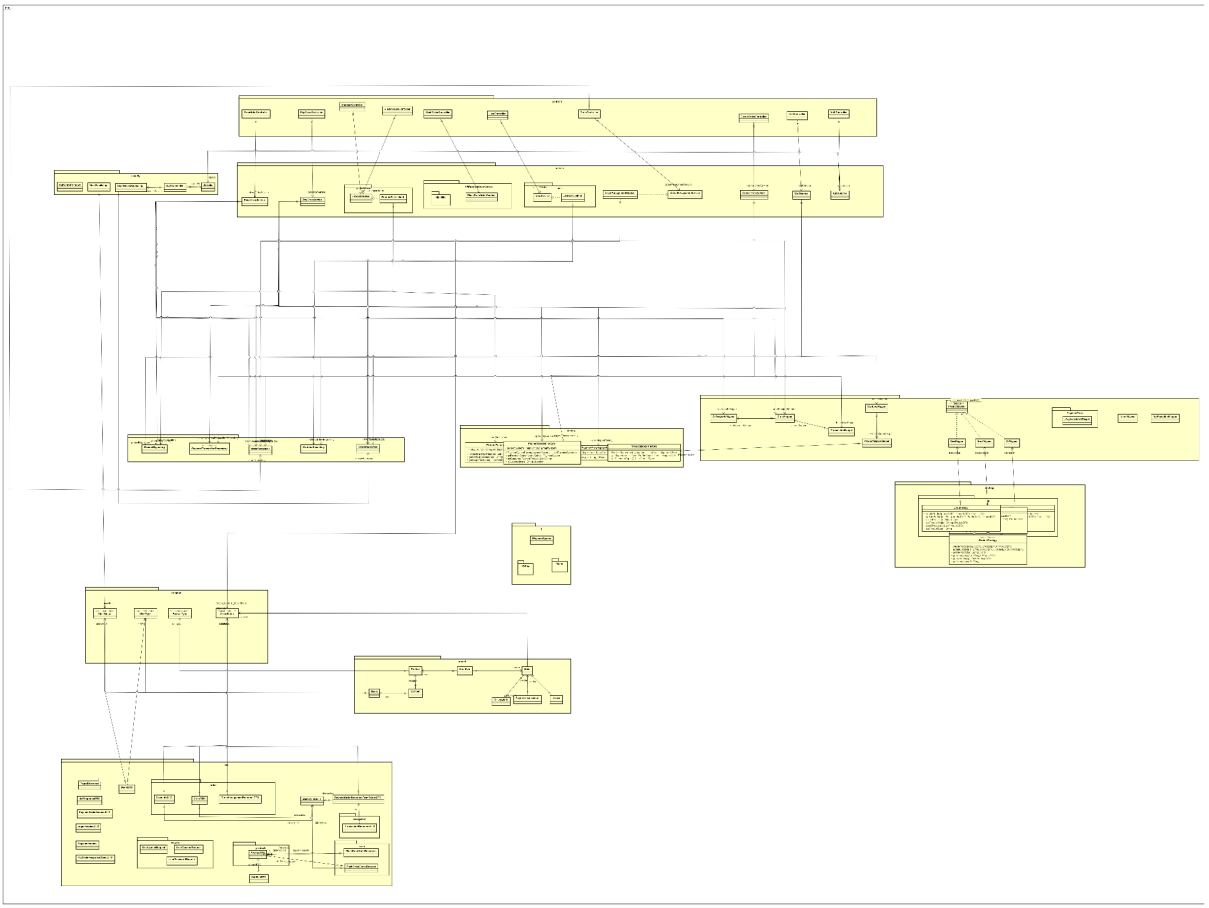
4.3 Non-Database Management System Files

<Provide the detailed description of all non-DBMS files if any and include a narrative description of the usage of each file that identifies if the file is used for input, output, or both, and if the file is a temporary file. Also provide an indication of which modules read and write the file and include file structures (refer to the data dictionary). As appropriate, the file structure information should include the following:

- *Record structures, record keys or indexes, and data elements referenced within the records*
- *Record length (fixed or maximum variable length) and blocking factors*
- *Access method (e.g., index sequential, virtual sequential, random access, etc.)*
- *Estimate of the file size or volume of data within the file, including overhead resulting from file access methods*
- *Definition of the update frequency of the file (If the file is part of an online transaction-based system, provide the estimated number of transactions per unit of time, and the statistical mean, mode, and distribution of those transactions.)*
- *Backup and recovery specifications>*

4.4 Class Design

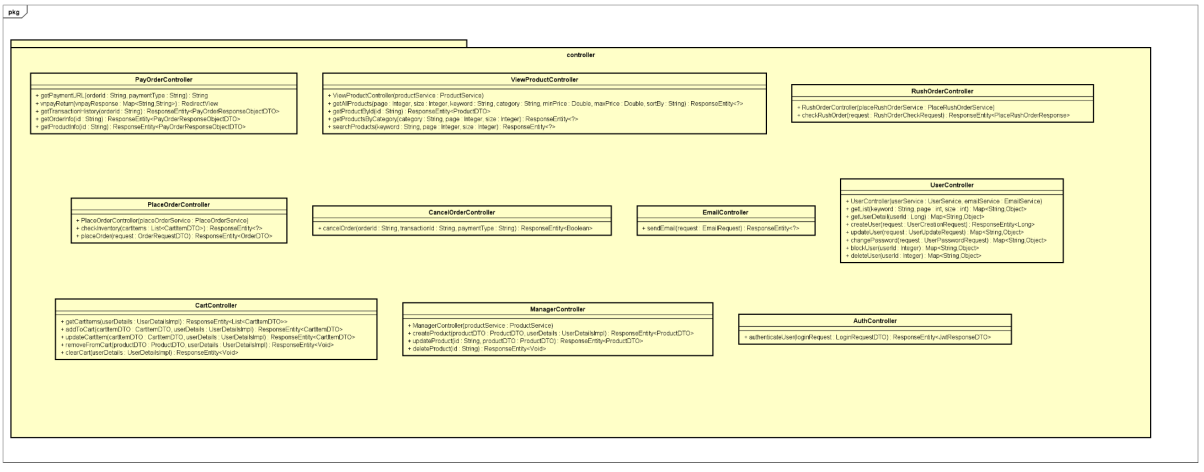
4.4.1 General Class Diagram



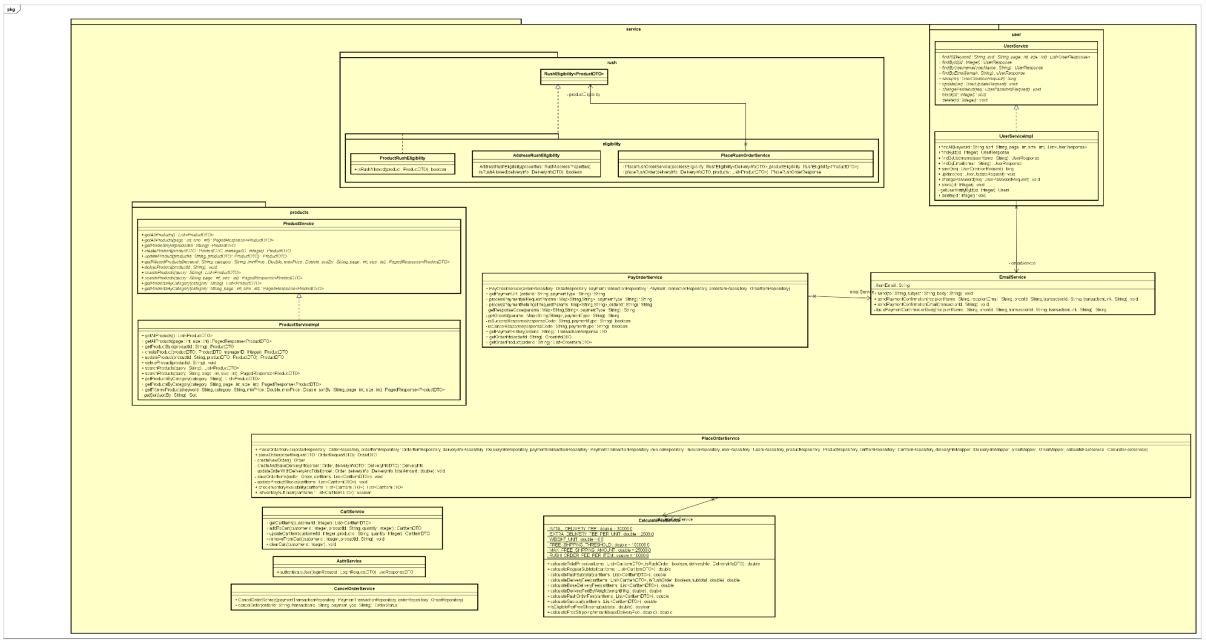
4.4.2 Class Diagrams

<Detail class diagram with full attributes and operations>

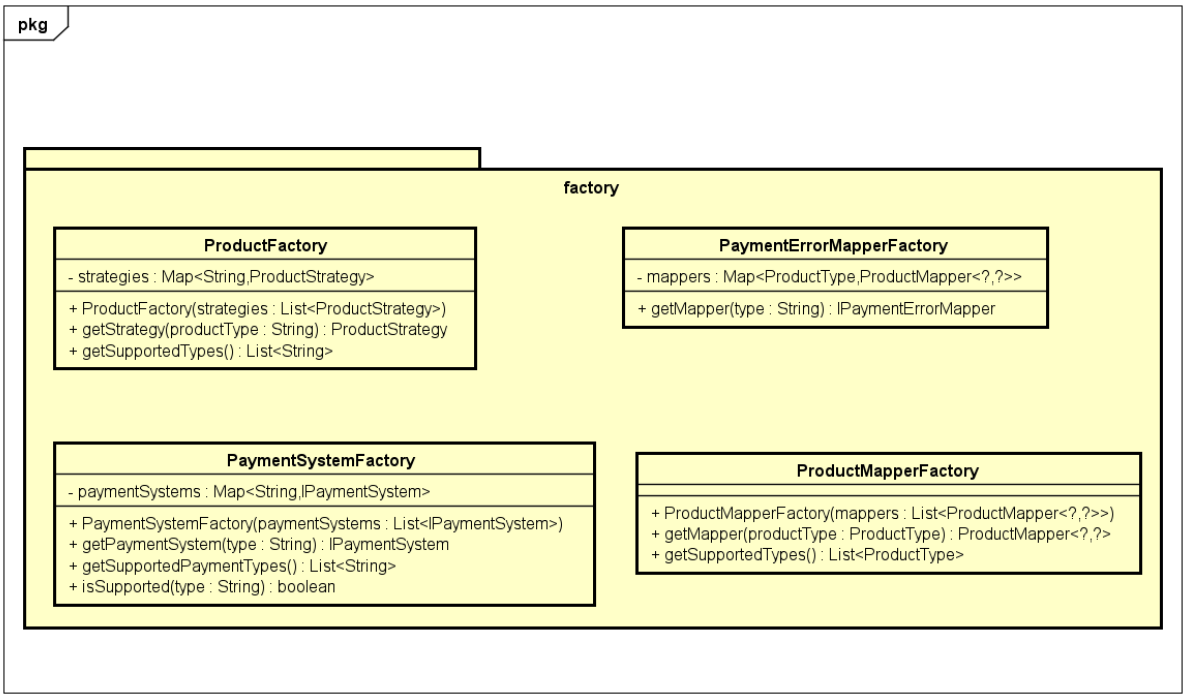
4.4.2.1 Class Diagram for Package Controller



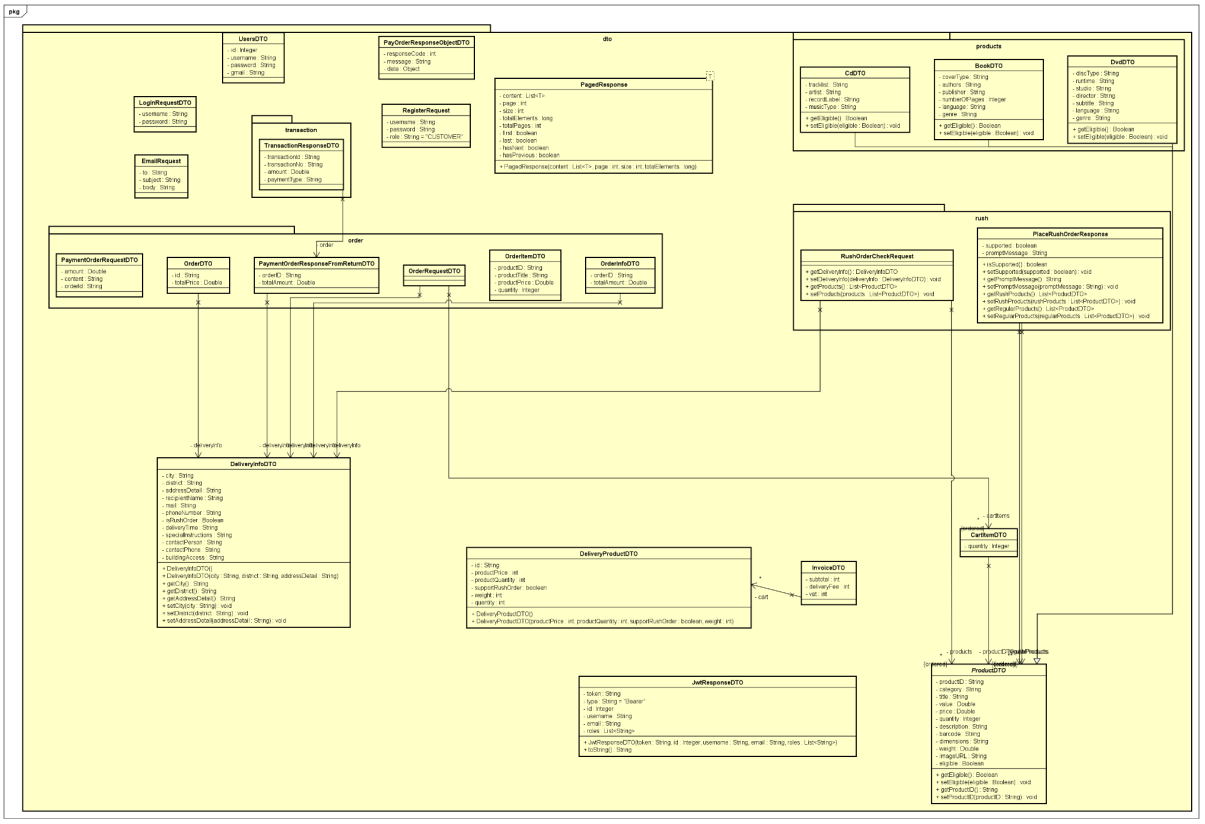
4.4.2.2 Class Diagram for Package Service



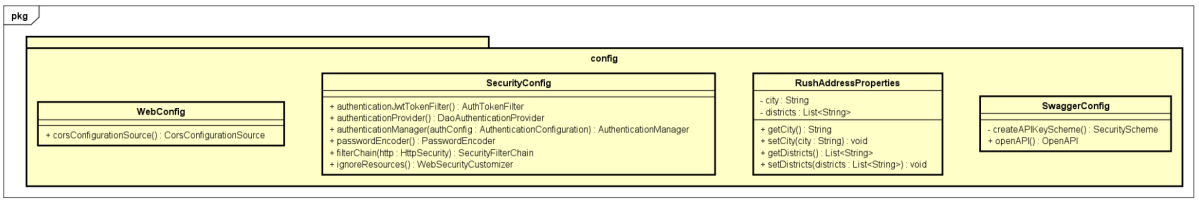
4.4.2.3 Class Diagram for Package Factory



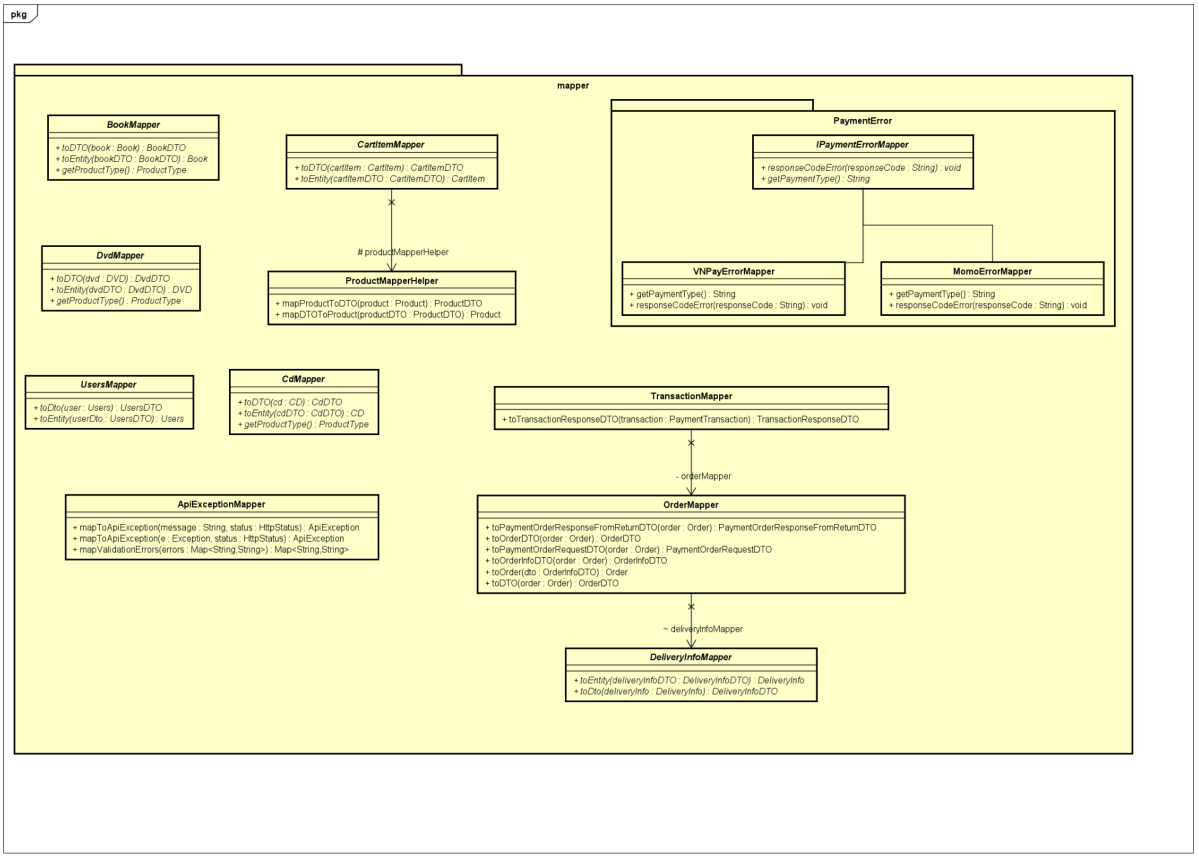
4.4.2.4 Class Diagram for Package DTO



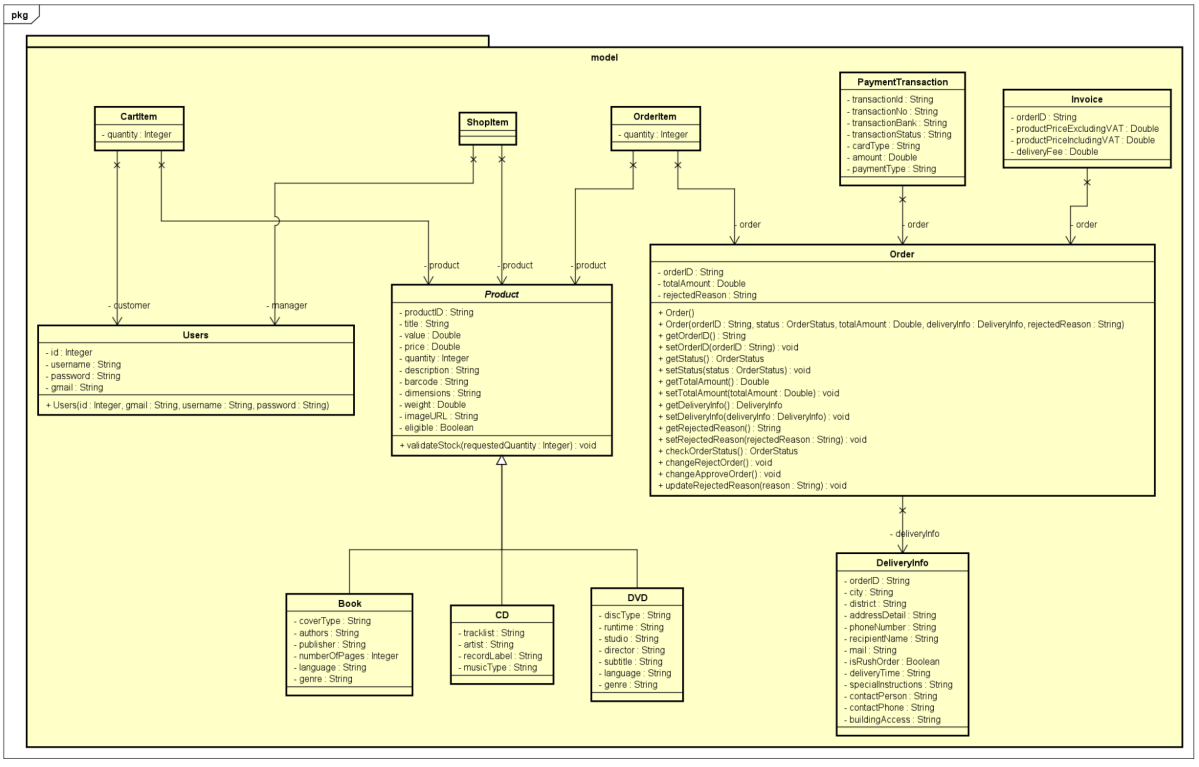
4.4.2.5 Class Diagram for Package Config



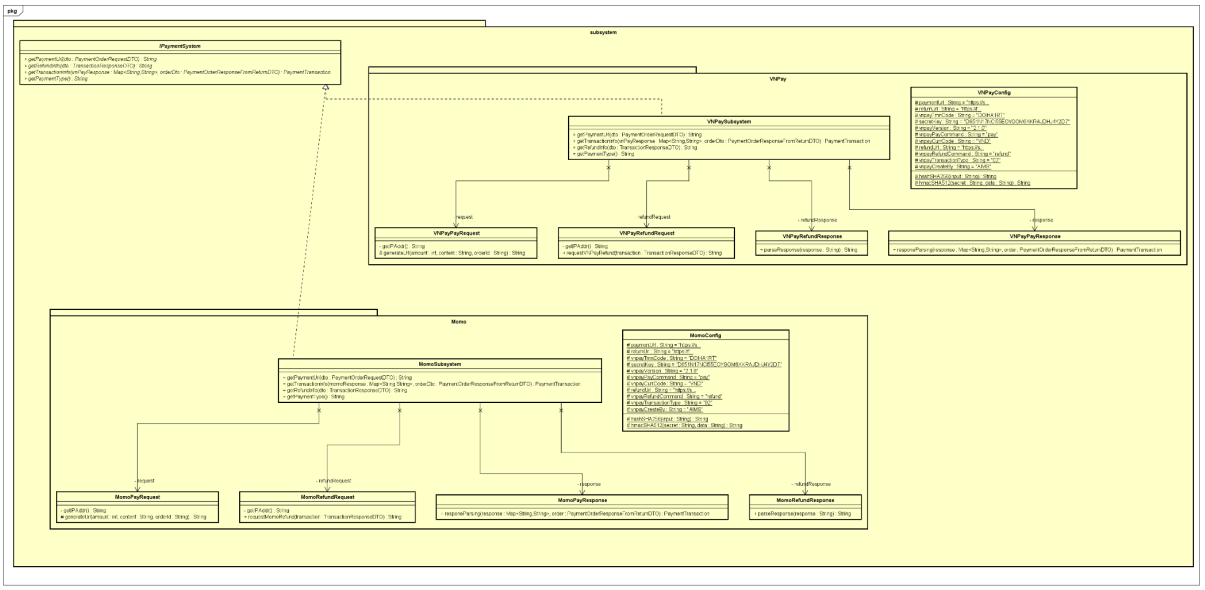
4.4.2.6 Class Diagram for Package Mapper



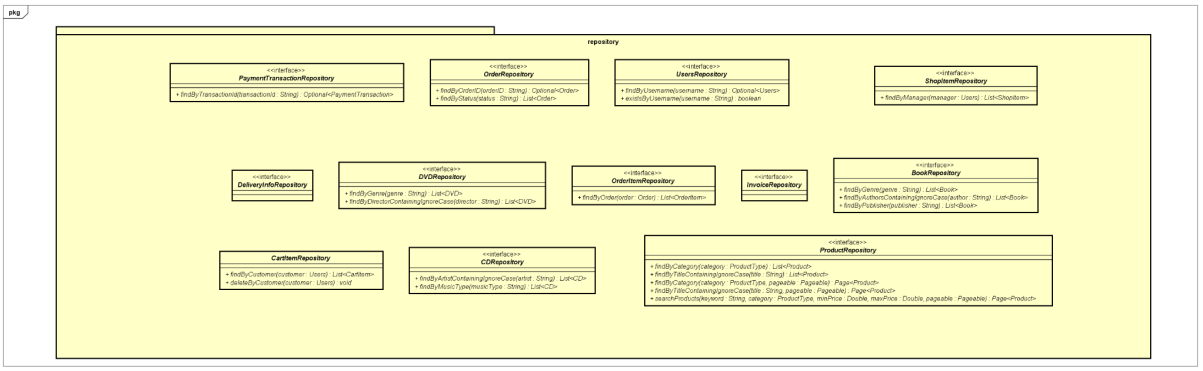
4.4.2.7 Class Diagram for Package Model



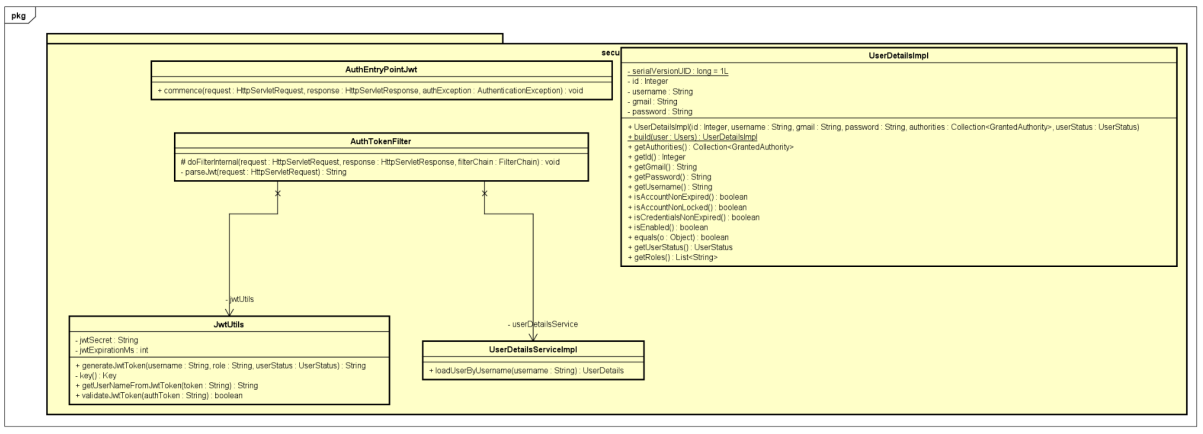
4.4.2.8 Class Diagram for Subsystem



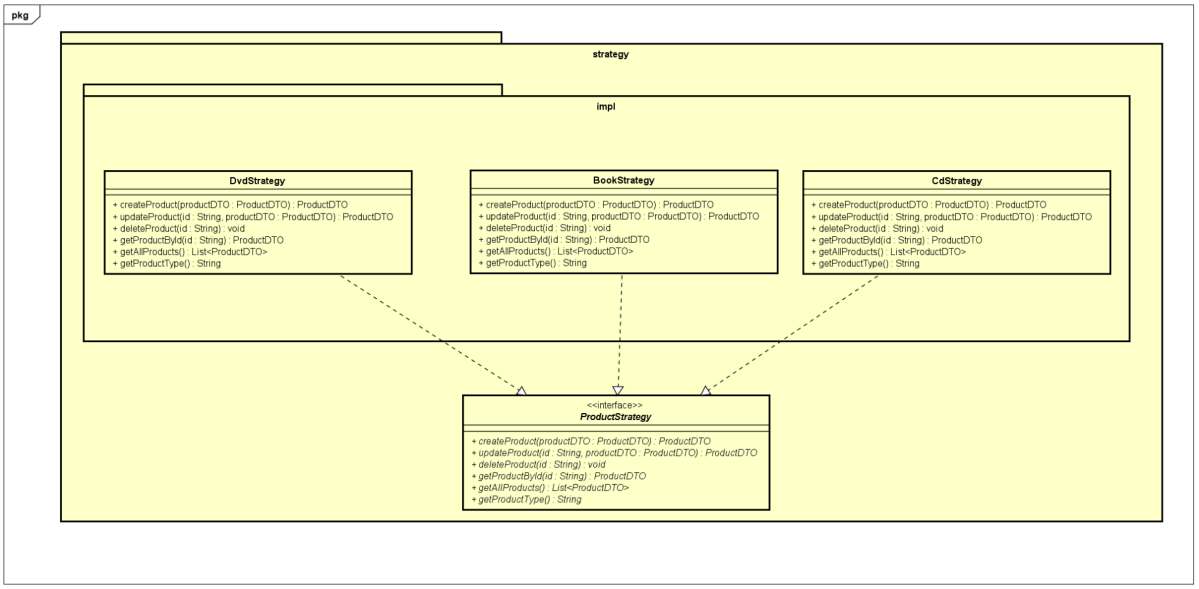
4.4.2.9 Class Diagram for Package Repository



4.4.2.10 Class Diagram for Package Security



4.4.2.11 Class Diagram for Package Strategy



4.4.3 Class Design

4.4.3.1. Class "CartController"

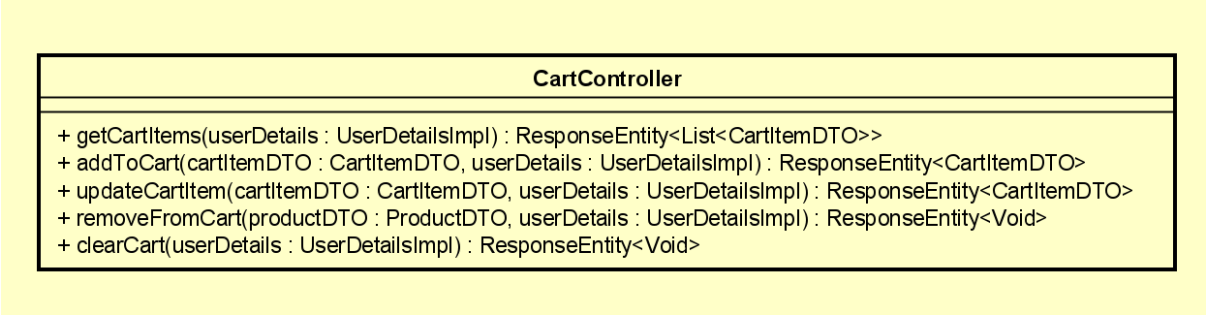


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	cartServic e	CartService	Final (injected)	Service used to manage cart operations

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	getCartItems(userDetails)	ResponseEntity<List<CartItemDT O>>	Retrieve all cart items for the logged-in user

2	addToCart(cartItemDTO, userDetails)	ResponseEntity<CartItemDTO>	Add a product to the user's cart
3	updateCartItem(cartItemDTO, userDetails)	ResponseEntity<CartItemDTO>	Update the quantity of a product in the cart
4	removeFromCart(productDTO, userDetails)	ResponseEntity<Void>	Remove a product from the cart
5	clearCart(userDetails)	ResponseEntity<Void>	Clear all items from the user's cart

---

Method: getCartItems(userDetails)

- **Parameters:**
    1. userDetails: UserDetailsImpl → Authenticated user information
- **Exceptions:** None
- **How to use parameters/attributes:**
    1. Extract customerId from userDetails
    2. Call cartService.getCartItems(customerId)
    3. Return the list of cart items

Method: addToCart(cartItemDTO, userDetails)

- **Parameters:**
    1. cartItemDTO: CartItemDTO → Contains product ID and quantity
    2. userDetails: UserDetailsImpl → Authenticated user
- **Exceptions:** None
- **How to use parameters/attributes:**

1. Extract customerId, productId, and quantity
2. Call cartService.addToCart(customerId, productId, quantity)
3. Return the created cart item

Method: updateCartItem(cartItemDTO, userDetails)

- **Parameters:**
  1. cartItemDTO: CartItemDTO → Contains product ID and new quantity
  2. userDetails: UserDetailsImpl → Authenticated user
- **Exceptions:**
  1. Returns 404 Not Found if update fails (item not found)
- **How to use parameters/attributes:**
  1. Extract customerId, productId, and quantity
  2. Call cartService.updateCartItem(...)
  3. Return updated item or 404 if not found

Method: removeFromCart(productDTO, userDetails)

- **Parameters:**
  1. productDTO: ProductDTO → Product to remove
  2. userDetails: UserDetailsImpl → Authenticated user
- **Exceptions:** None
- **How to use parameters/attributes:**
  1. Extract productId from DTO and customerId from user
  2. Call cartService.removeFromCart(...)

3. Return success response

Method: clearCart(userDetails)

- **Parameters:**
    1. userDetails: UserDetailsImpl → Authenticated user
- **Exceptions:** None
- **How to use parameters/attributes:**
    1. Get customerId
    2. Call cartService.clearCart(customerId)
    3. Return success response

4.4.3.2. Class "EmailController"

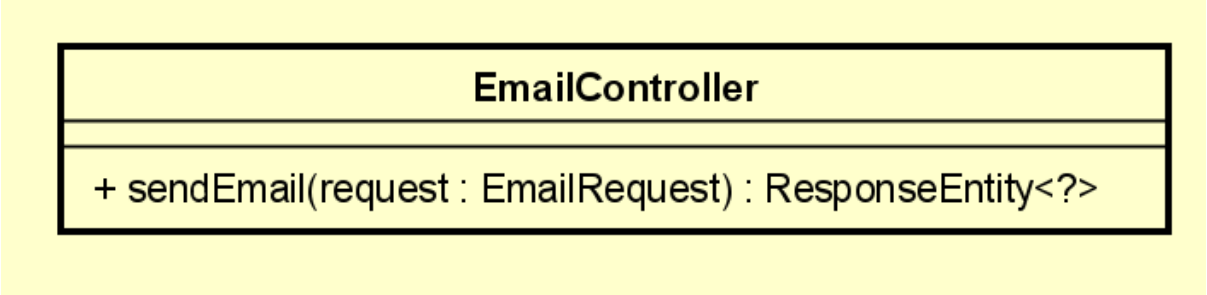


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	emailService	EmailService	Final (injected)	Service used to handle email sending logic

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	sendEmail(request)	ResponseEntity<?>	Send an email with specified recipient, subject, and body

---

Method: sendEmail(request)

- **Parameters:**
    1. request: EmailRequest → Contains recipient address (to), subject, and body of the email
- **Exceptions:**
    1. Returns HTTP 400 Bad Request if email sending fails due to any exception
- **How to use parameters/attributes:**
    1. Log the recipient's email address
    2. Call emailService.send(to, subject, body) using values from the request
    3. If successful, return a 200 OK response with confirmation message
    4. If an exception occurs, log the error and return a 400 Bad Request with error message

4.4.3.3. Class "CancelOrderController"

Table 1: Attribute design

#	Name	Data type	Default value	Description

1	cancelOrderService	CancelOrderService	Injected by @Autowired	Service used to handle order cancellation logic

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	cancelOrder(orderId, transactionId, paymentType)	ResponseEntity<Boolean>	Cancel the specified order and return success status

---

Method: cancelOrder(orderId, transactionId, paymentType)

- **Parameters:**
    1. orderId: String → ID of the order to be canceled
    2. transactionId: String → ID of the related transaction
    3. paymentType: String → Payment method used for the order
- **Exceptions:** None explicitly handled in this controller
- **How to use parameters/attributes:**
    1. Pass the orderId, transactionId, and paymentType to cancelOrderService.cancelOrder(...)
    2. Receive an OrderStatus result (e.g., CANCELLED)
    3. Return a ResponseEntity<Boolean>:
        - true if the status is CANCELLED
        - false otherwise

4.4.3.4. Class "AuthController"

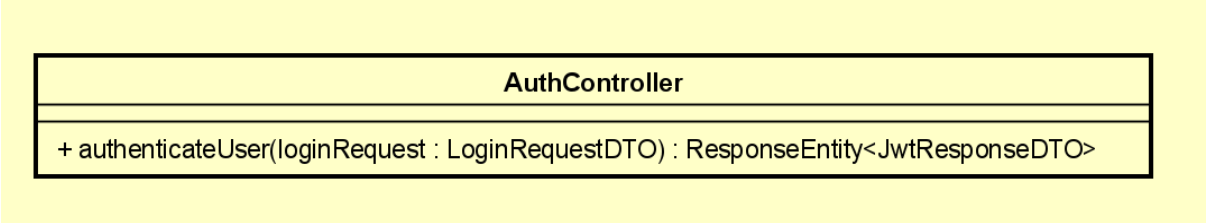


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	authService	AuthService	Final (injected)	Service used to handle authentication logic

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	authenticateUser(loginRequest)	ResponseEntity<JwtResponseDTO>	Authenticate user and return JWT response

---

Method: authenticateUser(loginRequest)

- **Parameters:**
    1. loginRequest: LoginRequestDTO → Contains username and password submitted by the client
- **Exceptions:** None explicitly handled in the controller
- **How to use parameters/attributes:**

1. Pass loginRequest to authService.authenticateUser(...)
2. Receive a JwtResponseDTO containing authentication token and user info
3. Return response with status 200 OK and the JWT data

4.4.3.5. Class "ManagerController"

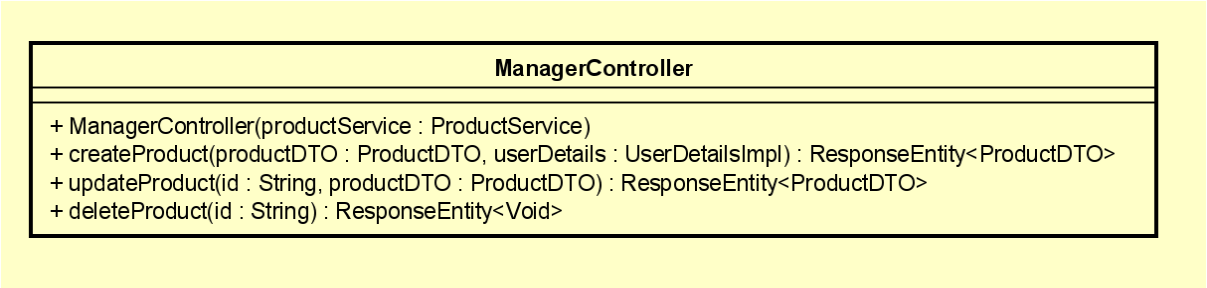


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	productService	ProductService	Injected via constructor	Service used to manage product creation, update, and deletion

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	createProduct(productDTO, userDetails)	ResponseEntity<ProductDTO>	Create a new product by the authenticated manager
2	updateProduct(id, productDTO)	ResponseEntity<ProductDTO>	Update an existing product

3	deleteProduct(id)	ResponseEntity<Void>	Delete a product by its ID

Method: createProduct(productDTO, userDetails)

- **Parameters:**
    1. productDTO: ProductDTO → Product information submitted by the manager
    2. userDetails: UserDetailsImpl → Authenticated manager info
- **Exceptions:** None explicitly handled
- **How to use parameters/attributes:**
    1. Extract managerID from userDetails
    2. Call productService.createProduct(...) with the DTO and manager ID
    3. Return the newly created product (including productID in response)

Method: updateProduct(id, productDTO)

- **Parameters:**
    1. id: String → ID of the product to be updated
    2. productDTO: ProductDTO → Updated product data
- **Exceptions:** None explicitly handled
- **How to use parameters/attributes:**
    1. Call productService.updateProduct(...) with the ID and DTO
    2. Return the updated product in response

Method: deleteProduct(id)

- **Parameters:**
    1. id: String → ID of the product to be deleted
- **Exceptions:** None explicitly handled
- **How to use parameters/attributes:**
    1. Call productService.deleteProduct(id)
    2. Return HTTP 204 No Content to indicate successful deletion

4.4.3.6. Class "PayOrderController"

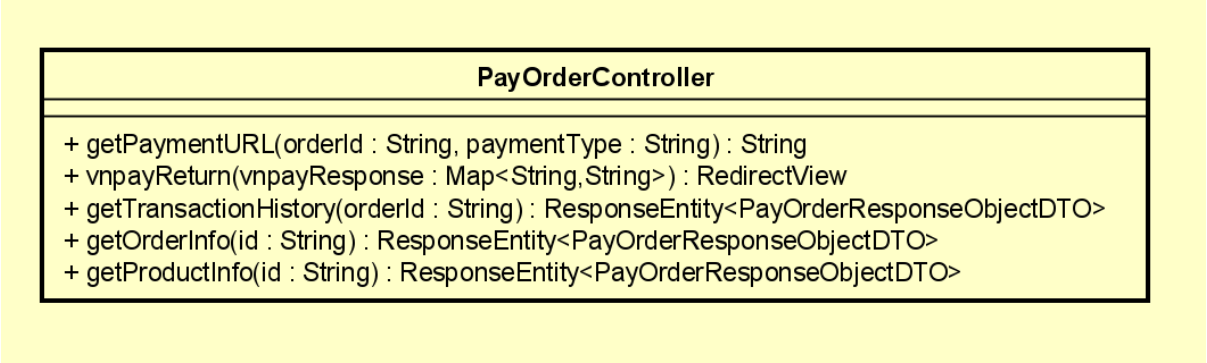


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	payOrderService	PayOrderService	Injected via @Autowired	Service that manages payment logic and order-related queries

---

Table 2: Operation design

#	Name	Return type	Description (purpose)

1	getPaymentURL(...)	String	Create and return a payment URL for a given order ID and method
2	vnpayReturn(...)	RedirectView	Handle response from VNPay gateway and redirect based on result
3	getTransactionHistory(...)	ResponseEntity<PayOrderResponseObjectDTO>	Return payment transaction details for a given order
4	getOrderInfo(...)	ResponseEntity<PayOrderResponseObjectDTO>	Return general info of the order by ID
5	getProductInfo(...)	ResponseEntity<PayOrderResponseObjectDTO>	Return list of products associated with the order

---

Method: getPaymentURL(orderId, paymentType)

- **Parameters:**

    - orderId: ID of the order being paid
    - paymentType: Payment gateway name (e.g., "vnpay", "momo")
- **How to use:**
    - Call payOrderService.getPaymentURL(orderId, paymentType) to get payment URL.
    - Return that URL as plain text in the response.

Method: vnpayReturn(vnpayResponse)

- **Parameters:**
    - vnpayResponse: Map of query parameters returned by VNPay
- **How to use:**
    - Extract orderId from vnp_TxnRef
    - Use payOrderService.processPaymentReturn(...) to handle result and get redirect URL
    - Return RedirectView to that URL

Method: getTransactionHistory(orderId)

- **Parameters:**
    - orderId: ID of the order whose transaction history is needed
- **How to use:**
    - Call payOrderService.getPaymentHistory(orderId)
    - Wrap response in PayOrderResponseObjectDTO with status message and code

---

Method: getOrderInfo(id)

- **Parameters:**
    - id: ID of the order
- **How to use:**
    - Call payOrderService.getOrderInfo(id)
    - Return a PayOrderResponseObjectDTO containing the order info

---

Method: getProductInfo(id)

- **Parameters:**
    - id: ID of the order
- **How to use:**
    - Call payOrderService.getOrderProduct(id)
    - Return a list of OrderItemDTO wrapped in response object

4.4.3.7. Class "PlaceOrderController"

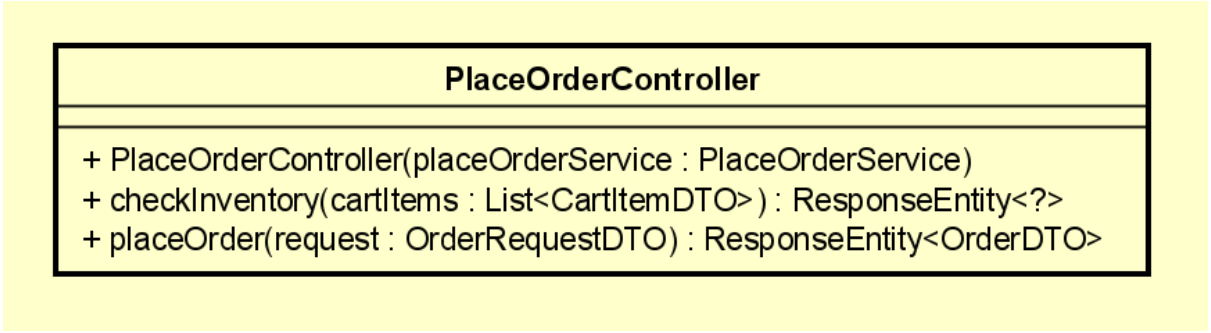


Table 1: Attribute design

#	Name	Data type	Default value	Description

1	placeOrderService	PlaceOrderService	Injected via constructor	Service used to validate inventory and place orders

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	checkInventory(cartItems)	ResponseEntity<?>	Check whether inventory is sufficient for each product in the cart
2	placeOrder(request)	ResponseEntity<OrderDTO>	Create and store a new order from the request payload

---

Method: checkInventory(cartItems)

- **Parameters:**
    1. cartItems: List of CartItemDTO representing items in the user's cart
- **Exceptions:** None explicitly handled
- **How to use parameters/attributes:**
    1. Pass the list of cart items to placeOrderService.checkInventoryAvailability(...)
    2. If all items are available → return success message
    3. If some are insufficient → return error message and list of problematic items

---

Method: placeOrder(request)

- **Parameters:**

1. request: OrderRequestDTO containing order information like user, shipping, items

- **Exceptions:** None explicitly handled
- **How to use parameters/attributes:**
    1. Call placeOrderService.placeOrder(...) with the order request
    2. Return the created OrderDTO inside a 200 OK response

4.4.3.8. Class "RushOrderController"

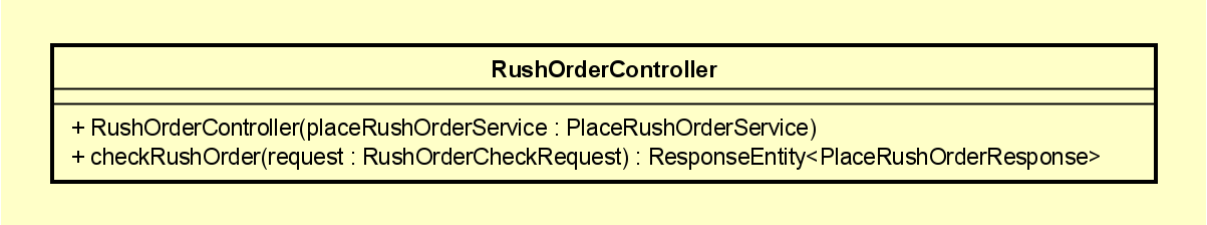


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	placeRushOrderService	PlaceRushOrderService	Injected via constructor	Service used to validate and process rush order requests

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	checkRushOrder(request)	ResponseEntity<PlaceRushOrderResponse>	Check if a rush order is eligible based on

			delivery info and products

Method: checkRushOrder(request)

- **Parameters:**
    1. request: RushOrderCheckRequest → contains DeliveryInfoDTO and list of ProductDTO
- **Exceptions:** None explicitly handled in the controller
- **How to use parameters/attributes:**
    1. Extract deliveryInfo and products from the request object.
    2. Call placeRushOrderService.placeRushOrder(deliveryInfo, products).
    3. Return the response as a PlaceRushOrderResponse wrapped in HTTP 200.

4.4.3.9. Class "UserController"

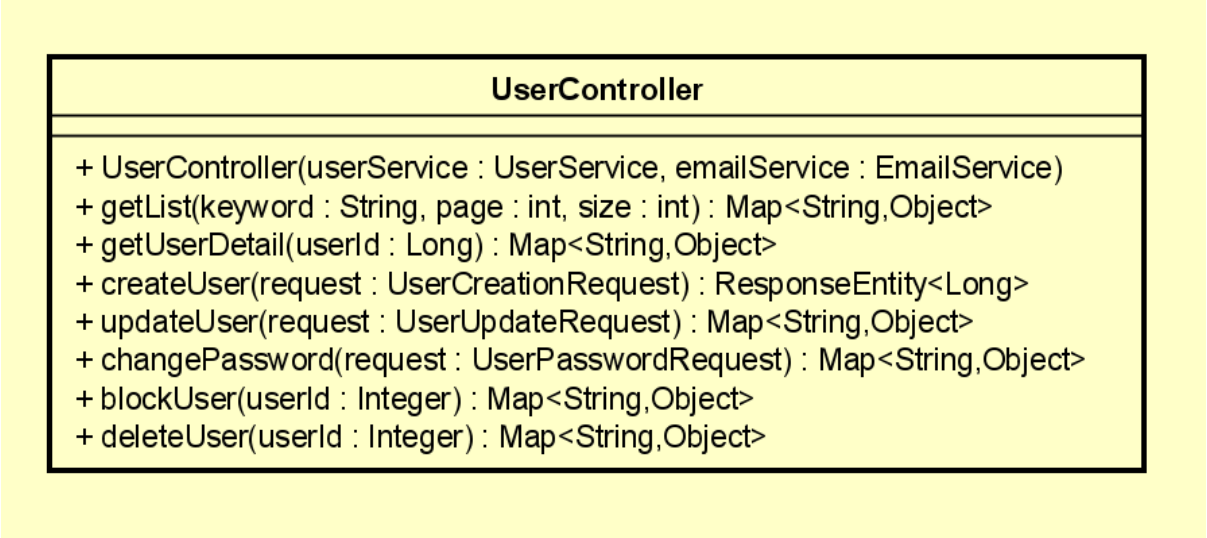


Table 1: Attribute design

#	Name	Data type	Default value	Description

1	userService	UserService	Injected via constructor	Service handling user-related logic
2	emailService	EmailService	Injected via constructor	Service handling email-related logic

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	getList(keyword, page, size)	Map<String, Object>	Get paginated list of users, optionally filtered by keyword
2	getUserDetail(userId)	Map<String, Object>	Get detailed info of a user by user ID
3	createUser(request)	ResponseEntity<Long>	Create a new user
4	updateUser(request)	Map<String, Object>	Update existing user information
5	changePassword(request)	Map<String, Object>	Change password for a user
6	blockUser(userId)	Map<String, Object>	Soft delete (block) a user by user ID
7	deleteUser(userId)	Map<String, Object>	Hard delete a user by user ID

Method: getList(keyword, page, size)

Parameters:

- keyword: optional String to filter users
- page: int, pagination page number
- size: int, page size

Exceptions: None explicitly handled in controller

How to use parameters/attributes:

- Call userService.findAll(keyword, "id", page, size) to get list of users.
- Wrap result in a Map with status, message, and data.
- Return Map as JSON response.

Method: getUserDetail(userId)

Parameters:

- userId: Long, ID of the user to retrieve

Exceptions: None explicitly handled in controller

How to use parameters/attributes:

- Call userService.findById(userId.intValue()) to get user details.
- Wrap result in a Map with status, message, and data.
- Return Map as JSON response.

Method: createUser(request)

Parameters:

- request: UserCreationRequest containing data to create a user

Exceptions: None explicitly handled in controller

How to use parameters/attributes:

- Call userService.save(request) to create the user.
- Return HTTP 201 CREATED with created user ID (here, dummy 1L).

Method: updateUser(request)

Parameters:

- request: UserUpdateRequest containing data to update user

Exceptions: None explicitly handled in controller

How to use parameters/attributes:

- Call userService.update(request) to update user data.
- Return a Map with status ACCEPTED, message, and empty data.

Method: changePassword(request)

Parameters:

- request: UserPasswordRequest containing new password info

Exceptions: None explicitly handled in controller

How to use parameters/attributes:

- Call userService.changePassword(request).
- Log the password change.
- Return Map with status NO_CONTENT and success message.

Method: blockUser(userId)

Parameters:

- userId: Integer ID of user to block

Exceptions: None explicitly handled in controller

How to use parameters/attributes:

- Call userService.block(userId) to block user.
- Log the action.
- Return Map with status RESET_CONTENT and success message.

Method: deleteUser(userId)

Parameters:

- userId: Integer ID of user to delete

Exceptions: None explicitly handled in controller

How to use parameters/attributes:

- Call userService.delete(userId) to hard delete user.
- Return Map with status OK and success message.

4.4.3.10. Class "ViewProductController"

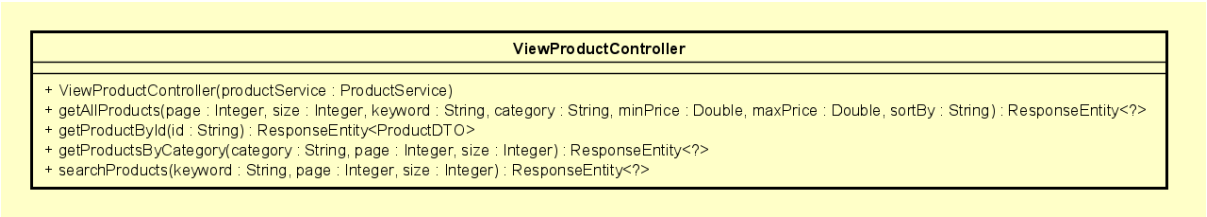


Table 1: Attribute design

#	Name	Data type	Default value	Description

1	productServic e	ProductServi ce	Injected via constructor	Service used to retrieve and search product data

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	getAllProducts(...)	ResponseEntity<?>	Get all products with optional filters and pagination
2	getProductById(id)	ResponseEntity<ProductDT O>	Retrieve a single product by its ID
3	getProductsByCategory(. ..)	ResponseEntity<?>	Retrieve products by category with optional pagination
4	searchProducts(...)	ResponseEntity<?>	Search for products by keyword in title with optional pagination

Method: getAllProducts(...)

Parameters:

- page: Integer (optional) — page number for pagination
- size: Integer (optional) — number of items per page
- keyword: String (optional) — filter by keyword
- category: String (optional) — filter by category
- minPrice, maxPrice: Double (optional) — filter by price range
- sortBy: String (optional) — sort field

Exceptions:
None explicitly handled in the controller

How to use parameters/attributes:

- Check for presence of any filter (keyword, category, price range, sort).
- If filters are present, call productService.getFilteredProducts(...).
- If pagination is specified but no filters, call productService.getAllProducts(page, size).
- Otherwise, call productService.getAllProducts() to get full list.

Method: getProductById(id)

Parameters:

- id: String — ID of the product to retrieve

Exceptions:
None explicitly handled in the controller

How to use parameters/attributes:

- Call productService.getProductById(id) to retrieve the product.
- Return result as ResponseEntity<ProductDTO>.

Method: getProductsByCategory(category, page, size)

Parameters:

- category: String — product category (book, cd, dvd, etc.)
- page, size: Integer (optional) — for pagination

Exceptions:
None explicitly handled in the controller

How to use parameters/attributes:

- If pagination params exist, call productService.getProductsByCategory(category, page, size).
- Otherwise, call productService.getProductsByCategory(category) for full list.

Method: searchProducts(keyword, page, size)

Parameters:

- keyword: String — text to search for in product title
- page, size: Integer (optional) — for pagination

Exceptions:
None explicitly handled in the controller

How to use parameters/attributes:

- If pagination params exist, call productService.searchProducts(keyword, page, size).
- Otherwise, call productService.searchProducts(keyword) for full result list.

4.4.3.11. Class "ProductServiceImpl"

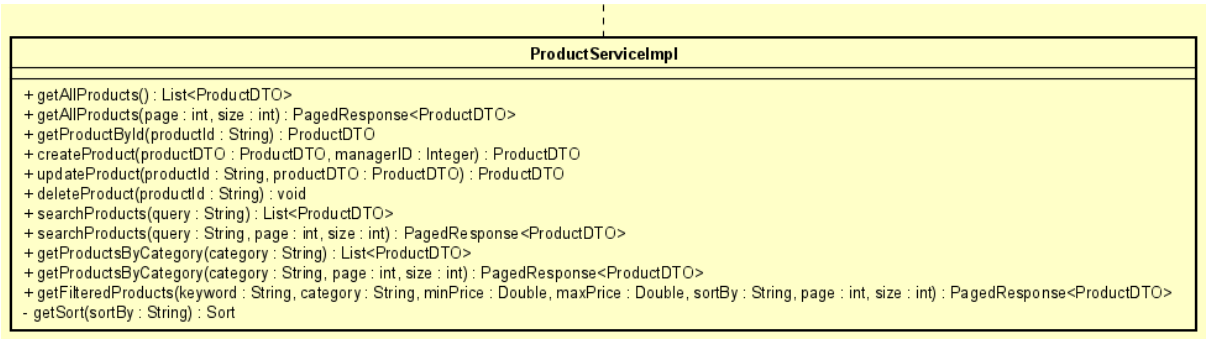


Table 1: Attribute design

#	Name	Data type	Default value	Description

1	productFactory	ProductFactory	Injected via constructor	Factory to determine product strategy based on product type
2	productRepository	ProductRepository	Injected via constructor	Repository for querying raw product entities from the database

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	getAllProducts()	List<ProductDTO>	Get list of all products (across all types)
2	getAllProducts(page, size)	PagedResponse<ProductDTO>	Get paginated list of all products
3	getProductById(productId)	ProductDTO	Retrieve a product by its ID
4	createProduct(productDTO, managerID)	ProductDTO	Create a new product using the correct strategy
5	updateProduct(productId, productDTO)	ProductDTO	Update an existing product by ID
6	deleteProduct(productId)	void	Delete a product by ID

7	searchProducts(query)	List<ProductDTO>	Search for products by keyword
8	searchProducts(query, page, size)	PagedResponse<ProductDTO>	Search for products by keyword with pagination
9	getProductsByCategory(category)	List<ProductDTO>	Get all products by category
10	getProductsByCategory(category, page, size)	PagedResponse<ProductDTO>	Get paginated products by category
11	getFilteredProducts(...)	PagedResponse<ProductDTO>	Get products filtered by keyword, category, price range, and sort options

---

Method: getAllProducts()

Description:
Retrieve all products across supported types using their corresponding strategy.

How it works:

- Loop through all product types from the factory
- Use each strategy's getAllProducts()
- Combine into a single list of ProductDTO

Method: getAllProducts(page, size)

Description:
Get paginated list of all products, then convert them to ProductDTO using corresponding strategy.

How it works:

- Use repository to fetch paginated Product
- Map each product to its strategy
- Build and return PagedResponse<ProductDTO>

Method: getProductById(productId)

Description:
Find product by ID, determine its strategy, and return detailed ProductDTO.

How it works:

- Query product by ID
- Throw ResourceNotFoundException if not found
- Use its category to get strategy and return product via getProductById

---

Method: createProduct(productDTO, managerID)

Description:
Create a new product using the strategy based on product category.

How it works:

- Get product type from productDTO.getCategory()
- Use the factory to get corresponding strategy
- Call createProduct(productDTO)

Method: updateProduct(productId, productDTO)

Description:
Update an existing product using appropriate strategy.

How it works:

- Determine product type from the DTO
- Get strategy and call updateProduct(productId, productDTO)

Method: deleteProduct(productId)

Description:
Delete product using strategy inferred from its stored category.

How it works:

- Query product
- Get category and appropriate strategy
- Call deleteProduct(productId)

Method: searchProducts(query)

Description:
Find products with title containing keyword (case-insensitive).

How it works:

- Query repository
- For each result, use category to get strategy and return via getProductById

---

Method: searchProducts(query, page, size)

Description:
Same as searchProducts, but with pagination.

How it works:

- Perform paged search by keyword

- Map result to ProductDTO using strategy
- Return PagedResponse

Method: getProductsByCategory(category)

Description:
Get all products of a specific category.

How it works:

- Use ProductFactory to get strategy by category
- Call getAllProducts() of that strategy

Method: getProductsByCategory(category, page, size)

Description:
Get paginated list of products filtered by category.

How it works:

- Convert category to ProductType enum
- Fetch paginated results from repository
- Map result to ProductDTO using strategy
- Return PagedResponse

Method: getFilteredProducts(keyword, category, minPrice, maxPrice, sortBy, page, size)

Description:
Advanced search supporting keyword, category, price range, and sorting.

How it works:

- Convert category to enum (nullable)

- Use helper getSort() to determine Sort type
- Call productRepository.searchProducts(...)
- Use category of each result to get appropriate strategy
- Return PagedResponse<ProductDTO>

Helper method: getSort(sortBy)

Description:
Convert sort keyword into Spring Sort object.

Supports:

- title_asc, title_desc, price_asc, price_desc

4.4.3.12. Class AddressRushEligibility

Table 1: Attribute design

#	Name	Data type	Default value	Description
1	properties	RushAddressProperties	Injected via constructor	Configuration containing allowed city and list of eligible districts

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	isRushAllowed(info)	boolean	Determine whether the delivery address is eligible for rush delivery

---

Method: isRushAllowed(deliveryInfo)

Parameters:

- deliveryInfo: DeliveryInfoDTO — Contains information about the delivery address, specifically city and district.

Exceptions:

- None explicitly thrown. Returns false in case of invalid or incomplete data.

How to use parameters/attributes:

- Check if deliveryInfo, city, or district is null → if any are, return false.
- Trim and normalize the strings.
- Compare city with allowed city from RushAddressProperties.
- Check if the given district matches any entry in the list of allowed districts (case-insensitive).

4.4.3.13. Class ProductRushEligibility

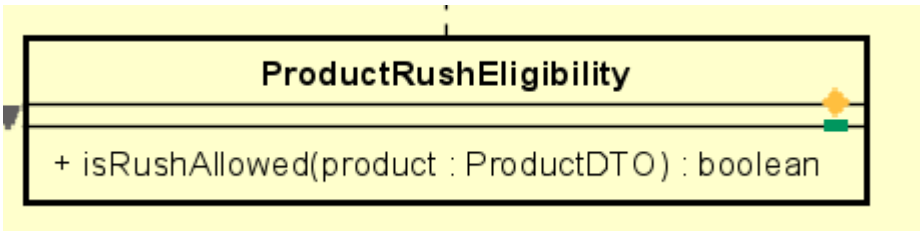


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	logger	Logger	Initialized via factory	Logger used for warning and debug messages during eligibility checks

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	isRushAllowed(product)	boolean	Check if a product is eligible for rush delivery based on its eligible field

---

Method: isRushAllowed(product)

Parameters:

- product: ProductDTO — The product to be evaluated for rush delivery eligibility.

Exceptions:

- No exceptions thrown. Handles null values gracefully with logging.

How to use parameters/attributes:

- If product is null, log a warning and return false.
- If product.getEligible() is null, log debug and return false.
- Otherwise, return the boolean value of product.getEligible() and log the result.

4.4.3.14. Class PlaceRushOrderService

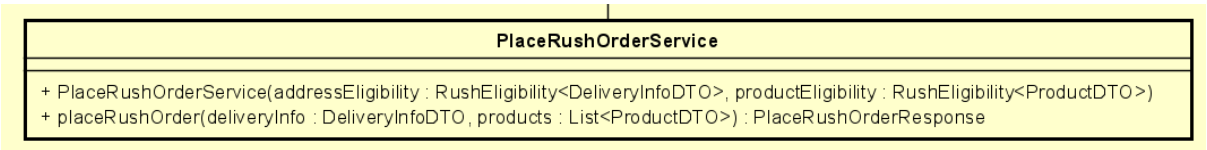


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	addressEligibility	RushEligibility<DeliveryInfoDTO>	Injected via constructor	Checks if the delivery address is eligible for rush delivery
2	productEligibility	RushEligibility<ProductDTO>	Injected via constructor	Checks if each product is eligible for rush delivery

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	placeRushOrder(...)	PlaceRushOrderResponse	Checks delivery address and product eligibility for rush delivery and forms a response

---

Method: placeRushOrder(deliveryInfo, products)

Parameters:

- deliveryInfo: DeliveryInfoDTO — contains shipping address information.
- products: List<ProductDTO> — list of products the user wants to order.

Exceptions:

- No explicit exceptions are thrown; if checking a product's eligibility throws an exception, the product is treated as **not eligible** for rush.

How to use parameters/attributes:

1. Use addressEligibility.isRushAllowed(deliveryInfo) to check if the delivery address qualifies.

2. Iterate through the product list:

    - Use productEligibility.isRushAllowed(product) to determine if each product is eligible.

    - Categorize products into **rushable** and **regular**.

3. Construct and return a PlaceRushOrderResponse:

    - Set supported flag if at least one product is eligible **and** the address is valid.

    - Include both rush and regular product lists.

    - Set promptMessage if rush delivery is **not** supported.

4.4.3.15. Class UserServiceImpl

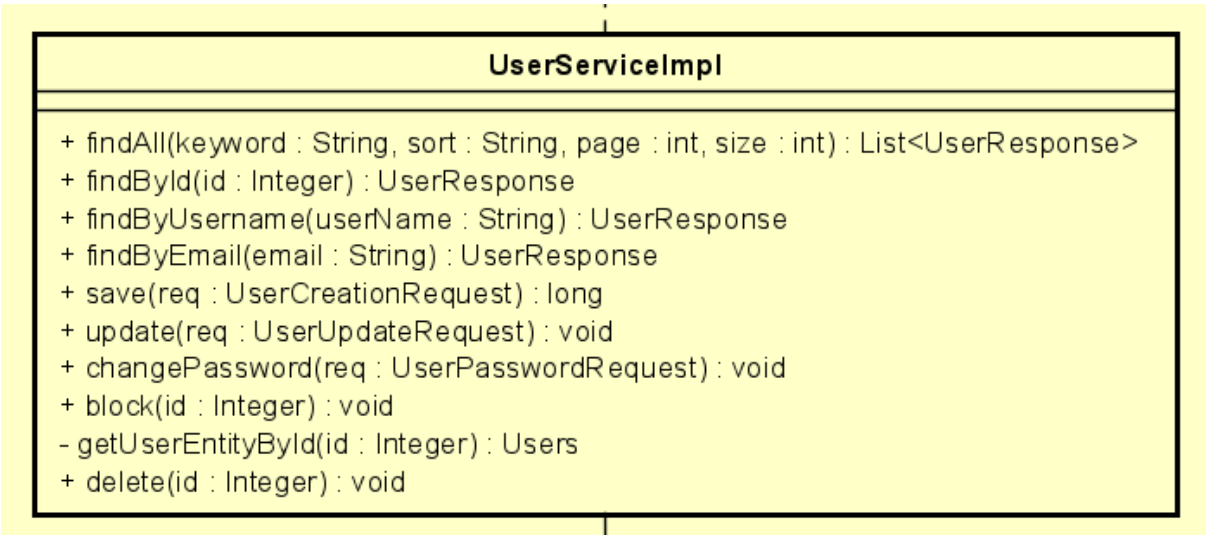


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	userRepository	UsersRepository	Injected	Repository for accessing and manipulating user data in the database
2	passwordEncoder	PasswordEncoder	Injected	Used to encode passwords before saving them securely
3	emailService	EmailService	Injected	Used to send email notifications related to user actions

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	findAll(...)	List<UserResponse>	Retrieve paginated and optionally sorted list of all users
2	findById(id)	UserResponse	Get detailed user info by ID
3	findByUsername()	UserResponse	Currently unimplemented: placeholder for lookup by username
4	findByEmail()	UserResponse	Currently unimplemented: placeholder for lookup by email
5	save(req)	long	Create a new user from request and encode password

6	update(req)	void	Update user fields, save to DB, and send notification email
7	changePassword()	void	Change password if confirmed, encode it, save to DB, send email
8	block(id)	void	Soft delete by setting status to BLOCKED
9	delete(id)	void	Hard delete user from DB and send deletion notification email
10	getUserEntityById(id)	Users	Private helper: retrieve user or throw if not found

Method: findAll(String keyword, String sort, int page, int size)

Parameters:

- keyword: optional, currently unused (search not yet implemented)
- sort: string in the form "field:asc" or "field:desc"
- page, size: pagination control

Logic:

- Uses regex to parse sort condition
- Builds Pageable and queries repository
- Maps Users → UserResponse

Method: save(UserCreationRequest req)

Parameters:

- req: includes username, password, Gmail, and user type

How attributes are used:

- Encode password with passwordEncoder
- Set status as UserStatus.NONE
- Save to database via userRepository

Return value:

- Always returns 1L (hardcoded; can be improved)

Method: update(UserUpdateRequest req)

Parameters:

- DTO containing user ID and updated fields

How attributes are used:

- Fetch user, update fields
- Send email notification using emailService

Exceptions:

- Catch block logs but does not throw on email failure

Method: changePassword(UserPasswordRequest req)

Parameters:

- id, password, and confirmPassword

How parameters are used:

- Check match between password fields
- Encode new password and update user
- Send email notification

Exceptions:

- Throws RuntimeException if passwords don’t match

Method: block(Integer id)

Purpose:

- Perform a "soft delete" by setting status to BLOCKED

How it's done:

- Fetch user by ID
- Modify status
- Save user

Method: delete(Integer id)

Purpose:

- Permanently delete user from the database

How it's done:

- Fetch and delete user
- Send email notification
- Catch and log errors

4.4.3.16. Class: AuthService

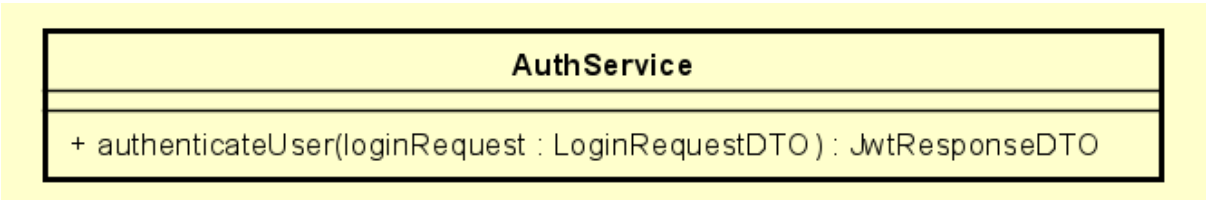


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	authenticationManager	AuthenticationManager	Injected via constructor	Component used to perform user authentication
2	jwtUtils	JwtUtils	Injected via constructor	Utility class for generating and handling JWT tokens

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	authenticateUser	JwtResponseDTO	Authenticate user based on login credentials and return JWT and user info

---

Method: authenticateUser(loginRequest)

- **Parameters:**
    1. loginRequest: LoginRequestDTO — Contains login information including username and password.
- **Exceptions:**
    1. Any exceptions during authentication are thrown outwards to be handled externally.
- **How to use parameters/attributes:**
    1. Use authenticationManager to authenticate using username and password from loginRequest.

2. If authentication is successful, retrieve user details (UserDetailsImpl) from the Authentication object.

3. Generate a JWT token by calling jwtUtils.generateJwtToken() with username, authority (role), and user status.

4. Return a JwtResponseDTO object containing the JWT token and user information such as id, username, email, and roles.

4.4.3.17. Class: CalculateFeeService

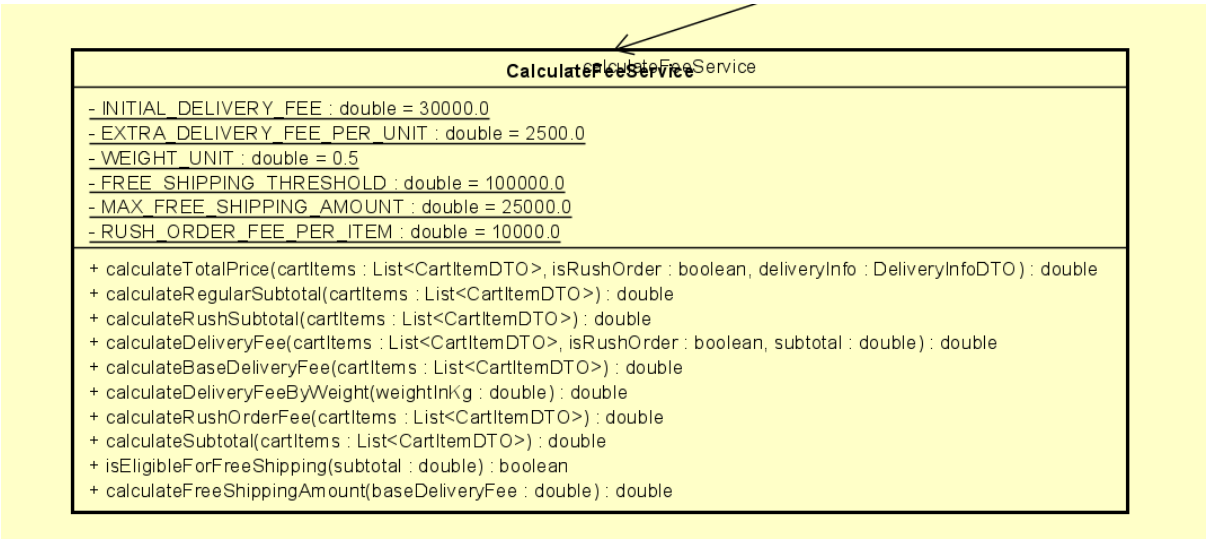


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	INITIAL_DELIVERY_FEE	double	30000.0	Delivery fee for the first 0.5 kg
2	EXTRA_DELIVERY_FEE_PER_UNIT	double	2500.0	Additional fee for each extra 0.5 kg

3	WEIGHT_UNIT	double	0.5	Weight unit used in delivery fee calculations
4	FREE_SHIPPING_THRESHOLD	double	100000.0	Minimum subtotal for free shipping eligibility
5	MAX_FREE_SHIPPING_AMOUNT	double	25000.0	Maximum discount for shipping when free
6	RUSH_ORDER_FEE_PER_ITEM	double	10000.0	Fee per rush-eligible product item

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	calculateTotalPrice(...)	double	Calculate total order price including all components
2	calculateRegularSubtotal(...)	double	Calculate subtotal of regular (non-rush) products
3	calculateRushSubtotal(...)	double	Calculate subtotal of rush-eligible products
4	calculateDeliveryFee(...)	double	Calculate delivery fee based on cart, subtotal, and rush flag

5	calculateBaseDeliveryFee(...)	double	Calculate fee based on the heaviest product
6	calculateDeliveryFeeByWeight(...)	double	Calculate fee based on given weight
7	calculateRushOrderFee(...)	double	Calculate rush fee per item
8	calculateSubtotal(...)	double	Total price of all items without any fee
9	isEligibleForFreeShipping(...)	boolean	Check if subtotal qualifies for free shipping
10	calculateFreeShippingAmount(...)	double	Return the amount to be discounted as free shipping

---

Method: calculateTotalPrice(cartItems, isRushOrder, deliveryInfo)

Parameters:

- cartItems: List<CartItemDTO> — List of items in the cart.
- isRushOrder: boolean — Whether the order is a rush order.
- deliveryInfo: DeliveryInfoDTO — Shipping info for the order.

Exceptions:

No exceptions are explicitly thrown.

How to use parameters/attributes:

- Call calculateRegularSubtotal(cartItems)
- Call calculateRushSubtotal(cartItems)

- Call calculateDeliveryFee(cartItems, isRushOrder, regularSubtotal)
- Call calculateRushOrderFee(cartItems)
- Return the sum of all the above.

Method: calculateRegularSubtotal(cartItems)

Parameters:

- cartItems: List<CartItemDTO>

Return type:

- double

Exceptions:

None

How to use parameters/attributes:

- Filter items where item.getProductDTO().getEligible() != true
- Sum product price × quantity
- Return the result

Method: calculateRushSubtotal(cartItems)

Parameters:

- cartItems: List<CartItemDTO>

Return type:

- double

Exceptions:

None

How to use parameters/attributes:

- Filter items where item.getProductDTO().getEligible() == true
- Sum product price × quantity
- Return the result

Method: calculateDeliveryFee(cartItems, isRushOrder, subtotal)

Parameters:

- cartItems: List<CartItemDTO>
- isRushOrder: boolean
- subtotal: double

Return type:

- double

Exceptions:

None

How to use parameters/attributes:

- If isRushOrder is true, call calculateBaseDeliveryFee(cartItems) and return it.
- Else, if subtotal > FREE_SHIPPING_THRESHOLD, discount up to MAX_FREE_SHIPPING_AMOUNT from the base fee.
- Return the computed delivery fee.

Method: calculateBaseDeliveryFee(cartItems)

Parameters:

- cartItems: List<CartItemDTO>

Return type:

- double

Exceptions:

None

How to use parameters/attributes:

- Find max weight from all items.
- Call calculateDeliveryFeeByWeight(maxWeight).
- Return the result.

Method: calculateDeliveryFeeByWeight(weightInKg)

Parameters:

- weightInKg: double

Return type:

- double

Exceptions:

None

How to use parameters/attributes:

- If weight ≤ 0, return 0
- If weight ≤ WEIGHT_UNIT, return INITIAL_DELIVERY_FEE
- Otherwise, calculate number of extra 0.5kg units and compute:
  INITIAL_DELIVERY_FEE + extraUnits * EXTRA_DELIVERY_FEE_PER_UNIT
- Return the result

Method: calculateRushOrderFee(cartItems)

Parameters:

- cartItems: List<CartItemDTO>

Return type:

- double

Exceptions:

None

How to use parameters/attributes:

- Filter rush-eligible items (item.getProductDTO().getEligible() == true)
- Multiply quantity * RUSH_ORDER_FEE_PER_ITEM
- Sum and return

Method: calculateSubtotal(cartItems)

Parameters:

- cartItems: List<CartItemDTO>

Return type:

- double

Exceptions:

None

How to use parameters/attributes:

- Sum all items: price × quantity regardless of eligibility
- Return the result

Method: isEligibleForFreeShipping(subtotal)

Parameters:

- subtotal: double

Return type:

- boolean

Exceptions:

None

How to use parameters/attributes:

- Return subtotal > FREE_SHIPPING_THRESHOLD

Method: calculateFreeShippingAmount(baseDeliveryFee)

Parameters:

- baseDeliveryFee: double

Return type:

- double

Exceptions:

None

How to use parameters/attributes:

- Return min(baseDeliveryFee, MAX_FREE_SHIPPING_AMOUNT)

4.4.3.18. Class: CancelOrderService

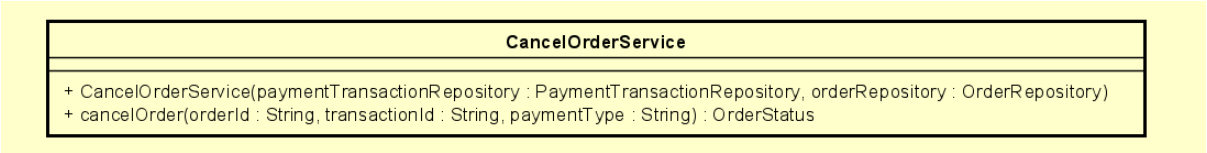


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	paymentTransactionRepository	PaymentTransactionRepository	Injected via @Autowired	Used to access and update payment transaction data
2	orderRepository	OrderRepository	Injected via @Autowired	Used to retrieve and update order information
3	transactionMapper	TransactionMapper	Injected via @Autowired	(Unused in current method) Used to convert transaction data to DTO
4	paymentSystemFactory	PaymentSystemFactory	Injected via @Autowired	(Unused in current method) Used to retrieve specific payment

				system handler

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	cancelOrder(...)	OrderStatus	Cancels an order if it's still pending, otherwise returns its current status.

---

cancelOrder(orderId, transactionId, paymentType)

Parameters:

- orderId: String — The unique ID of the order that needs to be canceled.
- transactionId: String — The ID of the associated payment transaction (not used yet in this version).
- paymentType: String — Type of payment gateway (e.g., vnpay, momo) (also unused here).

Exceptions:

- Throws RuntimeException if:
    - The order does not exist.
    - The order cannot be canceled at its current status.

How to use parameters/attributes:

1. Use orderRepository.findByOrderID(orderId) to retrieve the order.
2. If not found, throw "Order not found" exception.

3. If the status is already CANCELLED, return it immediately.
4. If the status is APPROVED or REJECTED, return that status without making changes.
5. If the status is PENDING, update the status to CANCELLED and save the order.
6. In other cases, throw "Order cannot be cancelled at this stage" exception.
7. Future versions may also handle payment transaction refund logic using transactionId and paymentSystemFactory.

4.4.3.19. Class: CartService

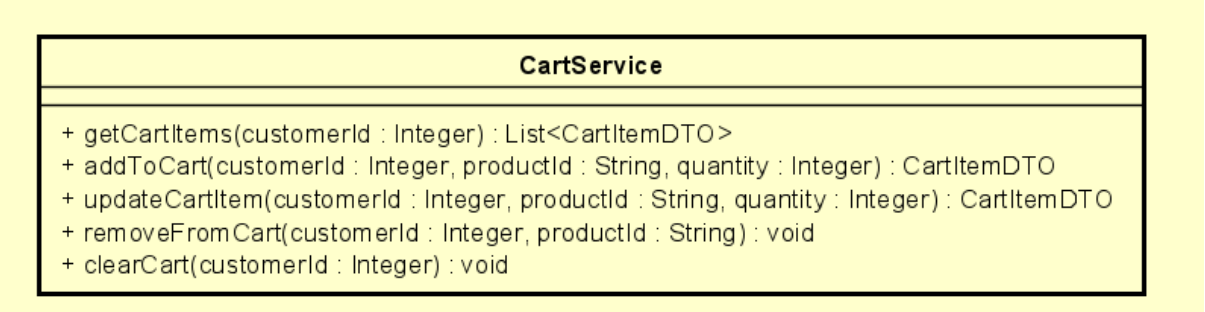


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	cartItemRepository	CartItemRepository	Injected via constructor	Used to access cart item data in the database
2	userRepository	UsersRepository	Injected via constructor	Used to retrieve customer/user information
3	cartItemMapper	CartItemMapper	Injected via constructor	Used to map between CartItem and CartItemDTO

4	productRepository	ProductRepository	Injected via constructor	Used to retrieve product information

---

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	getCartItems(customerId)	List<CartItemDTO>	Retrieve all items in a user's cart
2	addToCart(customerId, productId, quantity)	CartItemDTO	Add a product to the cart or update quantity if it already exists
3	updateCartItem(customerId, productId, quantity)	CartItemDTO	Update quantity of an existing item in the cart
4	removeFromCart(customerId, productId)	void	Remove an item from the cart
5	clearCart(customerId)	void	Remove all items from a customer's cart

---

Method getCartItems(customerId)

Parameters:

- customerId: Integer — ID of the customer whose cart is being fetched.

Exceptions:

- Throws ResourceNotFoundException if customer not found.

How to use parameters/attributes:

- Use userRepository.findById(customerId) to get the customer.
- If not found, throw exception.
- Use cartItemRepository.findByCustomer(...) to get cart items.
- Map each item to DTO via cartItemMapper.toDTO(...).
- Return the list of DTOs.

Method addToCart(customerId, productId, quantity)

Parameters:

- customerId: Integer — ID of the customer.
- productId: String — ID of the product to add.
- quantity: Integer — Quantity to be added.

Exceptions:

- ResourceNotFoundException if customer or product is not found.
- BadRequestException if quantity is invalid or stock is insufficient.

How to use parameters/attributes:

- Retrieve customer and product using their respective repositories.
- Create a new CartItem with composite ID (customerId, productId).
- Set customer, product, and quantity.
- Save to cartItemRepository.
- Convert to DTO via cartItemMapper.toDTO and return.

Method updateCartItem(customerId, productId, quantity)

Parameters:

- customerId: Integer — ID of the customer.
- productId: String — ID of the product.
- quantity: Integer — New quantity to update.

Exceptions:

- ResourceNotFoundException if product or cart item is not found.
- BadRequestException if quantity is invalid or stock is insufficient.

How to use parameters/attributes:

- Use productRepository.findById(productId) to verify product existence.
- Use cartItemRepository.findById(...) to get cart item.
- Update cartItem.setQuantity(...).
- Save and return DTO via mapper.

Method removeFromCart(customerId, productId)

Parameters:

- customerId: Integer — ID of the customer.
- productId: String — ID of the product.

Exceptions:

- ResourceNotFoundException if the cart item does not exist.

How to use parameters/attributes:

- Create composite ID: new CartItemId(customerId, productId).

- Check if item exists in cartItemRepository.
- If not, throw exception.
- Otherwise, delete it via cartItemRepository.deleteById(...).

Method clearCart(customerId)

Parameters:

- customerId: Integer — ID of the customer.

Exceptions:

- ResourceNotFoundException if customer not found.

How to use parameters/attributes:

- Use userRepository.findById(...) to retrieve customer.
- If not found, throw exception.
- Use cartItemRepository.deleteByCustomer(...) to delete all cart items for that customer.

4.4.3.20. Class: EmailService

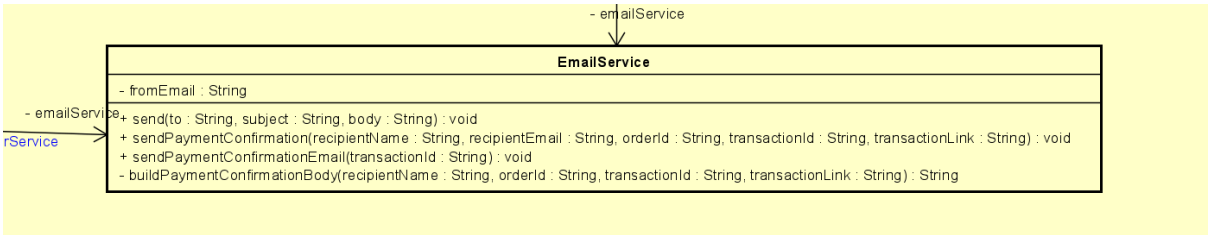


Table 1: Attribute design

#	Name	Data type	Default value	Description
1	mailSender	JavaMailSender	Injected via constructor	Used to send emails
2	paymentTransactionRepository	PaymentTransactionRepository	Injected via constructor	Used to fetch payment transaction and order details
3	fromEmail	String	Loaded from config	Sender email address (spring.mail.username)

Table 2: Operation design

#	Name	Return type	Description (purpose)
1	send(to, subject, body)	void	Sends a plain email with specified recipient, subject, and content
2	sendPaymentConfirmation(recipientName, recipientEmail, orderId, transactionId, transactionLink)	void	Sends payment confirmation email with order and transaction details

3	sendPaymentConfirmationEmail(transactionId)	void	Finds the transaction and sends a payment confirmation email to the customer
4	buildPaymentConfirmationBody(recipientName, orderId, transactionId, transactionLink)	String	Builds the content/body of the payment confirmation email

---

Method send(to, subject, body)

Parameters:

- to: String — Email address of the recipient
- subject: String — Email subject
- body: String — Email content

Exceptions:

- Throws Exception if email sending fails (e.g., due to SMTP issues)

How to use parameters/attributes:

- Create SimpleMailMessage
- Set sender (fromEmail), receiver, subject, and body
- Call mailSender.send(message)
- Log success or failure using log

Method sendPaymentConfirmation(recipientName, recipientEmail, orderId, transactionId, transactionLink)

Parameters:

- recipientName: String — Customer's name
- recipientEmail: String — Customer's email
- orderId: String — ID of the order
- transactionId: String — ID of the transaction
- transactionLink: String — URL to transaction details

Exceptions:

- Throws Exception if email sending fails

How to use parameters/attributes:

- Build subject and body with buildPaymentConfirmationBody(...)
- Call send(...) to dispatch the email

Method sendPaymentConfirmationEmail(transactionId)

Parameters:

- transactionId: String — ID of the transaction

Exceptions:

- Throws IllegalArgumentException if transaction not found
- Throws Exception if email sending fails

How to use parameters/attributes:

- Use paymentTransactionRepository.findByTransactionId(...) to get transaction
- Retrieve order, recipient's name and email from order.getDeliveryInfo()
- Build transaction link manually
- Call sendPaymentConfirmation(...)

Method buildPaymentConfirmationBody(recipientName, orderId, transactionId, transactionLink)

Parameters:

- recipientName: String
- orderId: String
- transactionId: String
- transactionLink: String

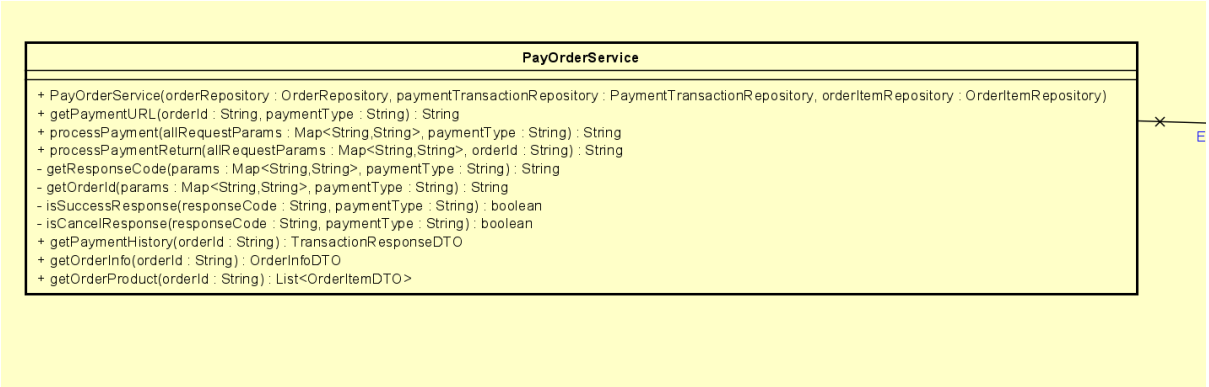
Return value:

- String — Email body content

How to use parameters/attributes:

- Format a plain text email using String.format(...) with placeholders for recipient info and transaction details

4.4.3.21. Class: PayOrderService



---

table 1: attribute design

#	name	data type	default value	description
1	currentOrder	OrderRepository	injected via constructor	used to retrieve and update orders
2	currentPaymentTransaction	PaymentTransactionRepository	injected via constructor	used to store and fetch payment transactions
3	orderItemRepository	OrderItemRepository	injected via constructor	used to retrieve items belonging to an order
4	errorMapperFactory	PaymentErrorMapperFactory	@Autowired	used to get error mappers for different payment systems
5	orderMapper	OrderMapper	@Autowired	used to map between Order and DTOs
6	emailService	EmailService	@Autowired	used to send payment confirmation emails
7	paymentSystemFactory	PaymentSystemFactory	@Autowired	used to get payment system handler

				(e.g., VNPay, MoMo)

---

table 2: operation design

#	name	return type	description
1	getPaymentURL(orderId, paymentType)	String	generate a payment URL for an order
2	processPayment(params, paymentType)	String	process a payment response from gateway and update system
3	processPaymentReturn(params, orderId)	String	determine payment type and delegate to processPayment
4	getResponseCode(params, paymentType)	String	extract response code from payment callback parameters
5	getOrderId(params, paymentType)	String	extract order ID from payment callback parameters
6	isSuccessResponse(code, paymentType)	boolean	check if payment response indicates success

7	isCancelResponse(code, paymentType)	boolean	check if payment response indicates cancellation
8	getPaymentHistory(orderId)	TransactionResponseDTO	retrieve payment transaction details for an order
9	getOrderInfo(orderId)	OrderInfoDTO	retrieve delivery and status info for an order
10	getOrderProduct(orderId)	List<OrderItemDTO>	retrieve list of products and quantities in an order

---

method getPaymentURL(orderId, paymentType)

parameters:

- orderId: String — ID of the order
- paymentType: String — payment gateway type (e.g., "vnpay", "momo")

exceptions:

- IllegalArgumentException if order not found
- IllegalStateException if order is not in PENDING state

how to use parameters/attributes:

- fetch order using currentOrder.findByOrderID(...)
- map to DTO using orderMapper
- call paymentSystemFactory.getPaymentSystem(paymentType).getPaymentUrl(dto)

---

method processPayment(allRequestParams, paymentType)

parameters:

- allRequestParams: Map<String, String> — callback params from gateway
- paymentType: String — gateway type

returns:

- redirect URL (success, decline, or error page)

how to use parameters/attributes:

- extract responseCode, orderID
- on success:
    - fetch order, map to DTO
    - get transaction info, save it
    - update order status to PENDING
    - send confirmation email
- on cancel: return decline page
- on error: use error mapper to handle

---

method processPaymentReturn(allRequestParams, orderId)

parameters:

- allRequestParams: Map<String, String>
- orderId: String

returns:

- redirect URL (same as processPayment)

how to use parameters/attributes:

- fetch transaction to get paymentType
- if not found, infer type from params
- call processPayment(...)

---

method getResponseCode(params, paymentType)

parameters:

- params: Map<String, String>
- paymentType: String

returns:

- String — response code based on gateway type

how to use parameters/attributes:

- extract vnp_ResponseCode regardless of type (VNPay & MoMo both use it)

---

method getOrderId(params, paymentType)

parameters:

- params: Map<String, String>
- paymentType: String

returns:

- String — order ID from gateway parameters

how to use parameters/attributes:

- extract vnp_TxnRef as the order ID from all types

---

method isSuccessResponse(responseCode, paymentType)

parameters:

- responseCode: String
- paymentType: String

returns:

- boolean — true if payment is successful ("00")

---

method isCancelResponse(responseCode, paymentType)

parameters:

- responseCode: String
- paymentType: String

returns:

- boolean — true if payment is canceled ("24")

---

method getPaymentHistory(orderId)

parameters:

- orderId: String

returns:

- TransactionResponseDTO with order & transaction info

exceptions:

- IllegalArgumentException if transaction not found

how to use parameters/attributes:

- fetch transaction using currentPaymentTransaction
- convert to DTO and embed order info via orderMapper

---

method getOrderInfo(orderId)

parameters:

- orderId: String

returns:

- OrderInfoDTO

exceptions:

- IllegalArgumentException if order not found

how to use parameters/attributes:

- fetch from currentOrder, convert via orderMapper

---

method getOrderProduct(orderId)

parameters:

- orderId: String

returns:

- List<OrderItemDTO>

exceptions:

- IllegalArgumentException if order not found

how to use parameters/attributes:

- get order info
- fetch order items via orderItemRepository.findByOrder(...)
- map to OrderItemDTO with product ID, price, title, quantity

4.4.3.22. Class PlaceOrderService

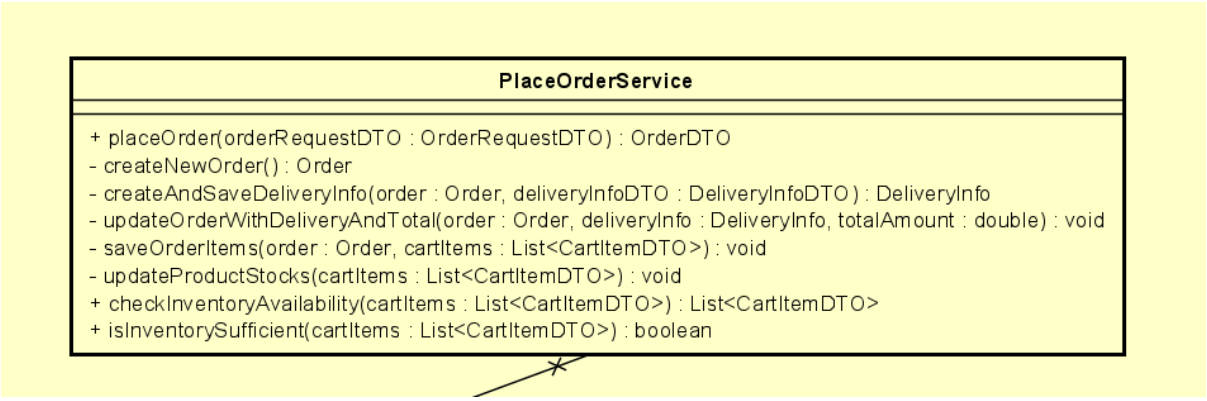


Table 1: Attribute design

Name	Data type	Default value	Description
orderRepository	OrderRepository	Injected via constructor	Provides methods for saving and retrieving orders
orderItemRepository	OrderItemRepository	Injected via constructor	Used to persist order items (product-quantity pairs)
productRepository	ProductRepository	Injected via constructor	Retrieves and updates product inventory

deliveryInfoMapper	DeliveryInfoMapper	Injected via constructor	Maps DeliveryInfoDTO to DeliveryInfo entity
orderMapper	OrderMapper	Injected via constructor	Converts Order to OrderDTO for response
deliveryInfoRepository	DeliveryInfoRepository	Injected via constructor	Persists delivery address and rush delivery info
calculateFeeService	CalculateFeeService	Injected via constructor	Calculates total amount including delivery and VAT

Table 2: Operation design

Name	Return type	Description (purpose)
placeOrder(...)	OrderDTO	Creates an order, saves delivery info, saves items, updates inventory, calculates total
checkInventoryAvailability(...)	List<CartItemDTO>	Returns a list of items that don't have sufficient stock
isInventorySufficient(...)	boolean	Returns true if all products in the cart have sufficient inventory

---

Method: placeOrder(orderRequestDTO)

Parameters:

- orderRequestDTO: OrderRequestDTO — Contains delivery info and a list of cart items.

Exceptions:

- If a product ID is not found, a RuntimeException is thrown with a descriptive message.

How to use parameters/attributes:

1. Use createNewOrder() to generate and save a new order with status PENDING.
2. Call createAndSaveDeliveryInfo(order, deliveryInfoDTO) to persist delivery details and link them with the order.
3. Use saveOrderItems(order, cartItems) to store each product and quantity in OrderItemRepository.
4. Call updateProductStocks(cartItems) to subtract ordered quantities from available product stock.
5. Use calculateFeeService.calculateTotalPrice(...) to get the total price (subtotal + VAT + delivery).
6. Call updateOrderWithDeliveryAndTotal(order, deliveryInfo, totalAmount) to store delivery info and total amount into the order.
7. Use orderMapper.toOrderDTO(order) to convert the entity to DTO and return it.

Method: checkInventoryAvailability(cartItems)

Parameters:

- cartItems: List<CartItemDTO> — Cart items to validate against product inventory.

Returns:

- List of cart items that have requested quantity greater than available stock.

Behavior:

- Loops through each item, compares with product stock.

- If insufficient, appends the item (with updated actual quantity) to result list.

Method: isInventorySufficient(cartItems)

Parameters:

- cartItems: List<CartItemDTO> — Same as above.

Returns:

- true if inventory is sufficient for all items, otherwise false.

Behavior:

- Calls checkInventoryAvailability(cartItems) and checks if the result is empty.

4.4.3.23. Class ProductStrategy (Interface)

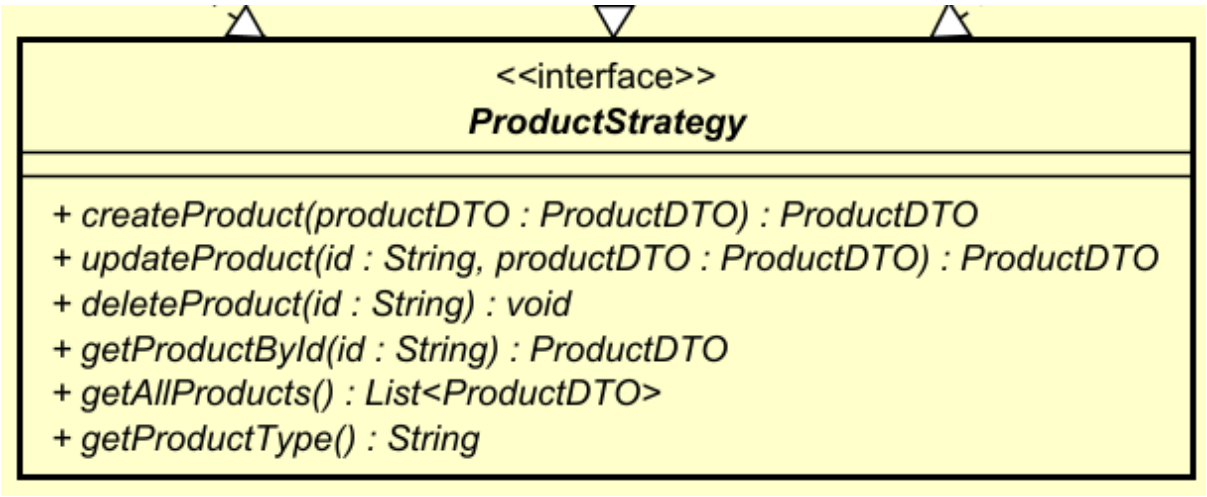


Table 1: Attribute design
Interface — no attributes are defined.

Table 2: Operation design

Name	Return type	Description (purpose)
createProduct(...)	ProductDTO	Creates a new product of a specific type
updateProduct(...)	ProductDTO	Updates an existing product by ID
deleteProduct(...)	void	Deletes the product by ID

getProductById(...)	ProductDTO	Retrieves product details for a given ID
getAllProducts()	List<ProductDTO>	Returns a list of all products of the specific type
getProductType()	String	Returns the product type this strategy handles (e.g., "book", "CD", "DVD")

Method: createProduct(productDTO)
Parameters:

- productDTO: ProductDTO — Contains the product details to be created.

Exceptions:

- Depends on implementation; might throw validation or persistence exceptions.

How to use parameters/attributes:

- Implementations use the input DTO to validate and create a product entity, then return the created DTO.

Method: updateProduct(id, productDTO)
Parameters:

- id: String — ID of the product to update.

- productDTO: ProductDTO — Updated product information.

How to use parameters/attributes:

- Locates the product by ID, updates it with new values, and returns the updated DTO.

Method: deleteProduct(id)
Parameters:

- id: String — ID of the product to delete.

How to use parameters/attributes:

- Removes the product from the database or marks it as inactive, depending on strategy implementation.

Method: getProductById(id)
Parameters:

- id: String — ID of the product to retrieve.

How to use parameters/attributes:

- Fetches product details by ID and returns them as a ProductDTO.

Method: getAllProducts()
Returns:

- A list of all products managed by this strategy implementation.

How to use parameters/attributes:

- Useful for displaying or processing all products of a specific type.

Method: getProductType()
Returns:

- A string representing the type of product the strategy supports (e.g., "book", "CD").

How to use parameters/attributes:

- Allows the system to map or route requests to the correct ProductStrategy implementation.

4.4.3.24. Class IPaymentSystem (Interface)

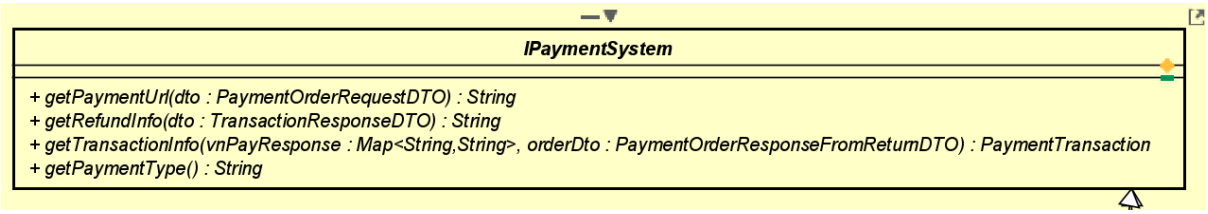


Table 1: Attribute design

Interface — No attributes defined.

Table 2: Operation design

Name	Return type	Description (purpose)
getPaymentUrl(...)	String	Generates a payment URL that redirects the user to the payment gateway

getRefundInfo(...)	String	Generates information used for processing a refund request
getTransactionInfo(...)	PaymentTransaction	Builds a PaymentTransaction object from the VNPAY response and order information
getPaymentType()	String	Returns the payment type identifier (e.g., "vnpay", "momo")

Method: getPaymentUrl(dto)

Parameters:

- dto: PaymentOrderRequestDTO — Includes order details like amount, description, redirect URL, etc.

Returns:

- A String representing a complete payment URL to redirect the customer to the payment provider.

How to use parameters/attributes:

- The implementation formats the request according to the provider's API (e.g., signs the payload, adds query params).

Method: getRefundInfo(dto)

Parameters:

- dto: TransactionResponseDTO — Contains details about the transaction to be refunded.

Returns:

- A String containing necessary data to request a refund via the third-party payment API.

How to use parameters/attributes:

- Constructs refund info or payload, possibly signed or encrypted, depending on the gateway.

Method: getTransactionInfo(vnPayResponse, orderDto)

Parameters:

- vnPayResponse: Map<String, String> — Response from the payment provider (e.g., VNPAY's callback or return URL).

- orderDto: PaymentOrderResponseFromReturnDTO — Contains order-related info to associate with the transaction.

Returns:

- A PaymentTransaction object populated with info extracted from the provider's response.

How to use parameters/attributes:

- Validates signature, extracts status, transaction ID, and maps data into the internal model.

Method: getPaymentType()

Returns:

- A String identifying the type of payment system (e.g., "vnpay", "momo").

How to use parameters/attributes:

- Used to determine which implementation of IPaymentSystem should be invoked for a given order.

4.4.3.25. Class ProductMapperFactory

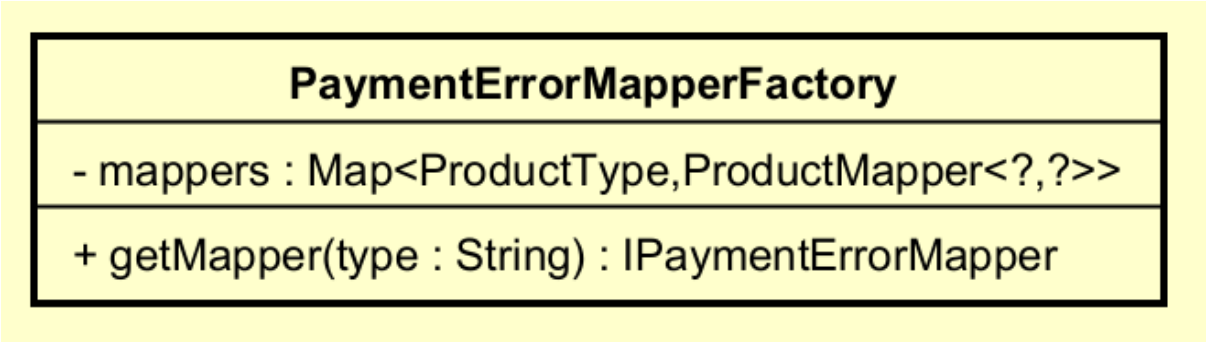


Table 1: Attribute design

Name	Data type	Default value	Description
mappers	Map<ProductType, ProductMapper<?, ?>>	Built via constructor injection	Maps each ProductType to its corresponding ProductMapper implementation

---

Table 2: Operation design

Name	Return type	Description (purpose)
getMapper(...)	ProductMapper<?, ?>	Returns the corresponding ProductMapper based on ProductType; throws exception if not found
getSupportedTypes()	List<ProductType>	Returns a sorted list of all supported product types registered in the factory

---

Method: getMapper(productType)

Parameters:

- productType: ProductType — Enum value representing the product type (e.g., book, dvd, cd).

Exceptions:

- Throws BadRequestException if productType is null or not supported.

How to use parameters/attributes:

- Looks up the internal Map<ProductType, ProductMapper> for a matching mapper.
- If found, returns it; if not, throws an error with a list of supported types.

Method: getSupportedTypes()

Parameters:

- None

Returns:

- A sorted list of all ProductType values currently registered in the factory.

How to use parameters/attributes:

- Useful for displaying available types in UIs or validating incoming requests.

Construction Logic:

- Constructor receives a List<ProductMapper<?, ?>> injected by Spring (thanks to @Component).
- Uses Collectors.toMap(...) to convert the list into a Map<ProductType, ProductMapper> using each mapper's getProductType() method as key.

4.4.3.26. Class ProductFactory

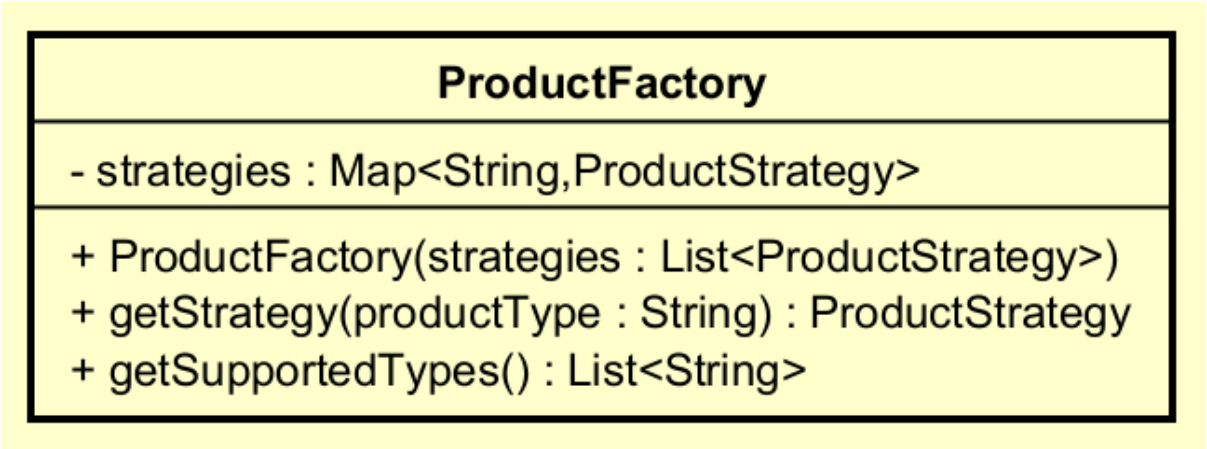


Table 1: Attribute design

Name	Data type	Default value	Description
strategies	Map<String, ProductStrategy>	Built via constructor injection	Maps product type strings (e.g., "book", "dvd") to corresponding ProductStrategy implementations

---

Table 2: Operation design

Name	Return type	Description (purpose)
getStrategy(...)	ProductStrategy	Retrieves the strategy for a given product type string; throws if type is invalid
getSupportedTypes()	List<String>	Returns a sorted list of all supported product types

---

Method: getStrategy(productType)

Parameters:

- productType: String — A string identifier of the product type (e.g., "book", "dvd").

Exceptions:

- Throws BadRequestException if the type is null, empty, or not found in the strategy map.

How to use parameters/attributes:

- Normalizes the type by trimming and converting to lowercase.
- Retrieves the corresponding ProductStrategy from the internal map.
- If no strategy exists, throws a BadRequestException with a list of supported types.

Method: getSupportedTypes()

Parameters:

- None

Returns:

- A sorted list of strings representing the registered product types (keys of the strategy map).

How to use parameters/attributes:

- Useful for frontend validation, dropdown menus, or API documentation.

Construction Logic:

- Constructor receives a List<ProductStrategy> from Spring context (@Component).
- Uses Collectors.toMap(...) to build a map where:
    - The key is the lowercase result of ProductStrategy.getProductType()
    - The value is the strategy implementation itself.

4.4.3.27. Class PaymentSystemFactory

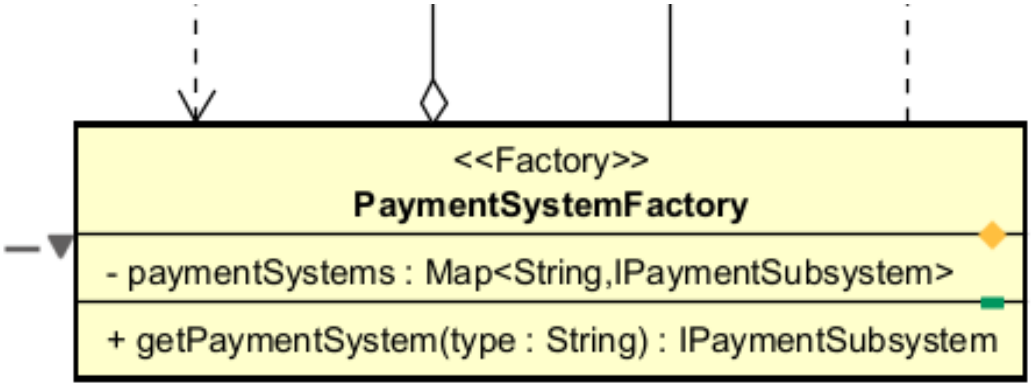


Table 1: Attribute design

Name	Data type	Default value	Description
paymentSystems	Map<String, IPaymentSystem>	Built via constructor injection	Maps each lowercase payment type string (e.g., "vnpay", "momo") to its corresponding IPaymentSystem implementation

---

Table 2: Operation design

Name	Return type	Description (purpose)
getPaymentSystem(...)	IPaymentSystem	Retrieves the payment system instance by type; throws exception if not found

---

Method: getPaymentSystem(type)

Parameters:

- type: String — Identifier of the payment system (e.g., "vnpay", "momo").

Exceptions:

- Throws IllegalArgumentException if type is null or not mapped to any known IPaymentSystem.

How to use parameters/attributes:

- Converts the input to lowercase to normalize.
- Looks up the corresponding IPaymentSystem in the internal map.
- Throws a detailed error if the type is unsupported.

Construction Logic:

- The constructor receives a List<IPaymentSystem> through Spring's auto-wiring (@Component).
- Converts the list into a Map<String, IPaymentSystem> by mapping each system's getPaymentType() as the key.
- All keys are expected to be lowercase to support case-insensitive lookup.

4.4.3.28. Class PaymentErrorMapperFactory

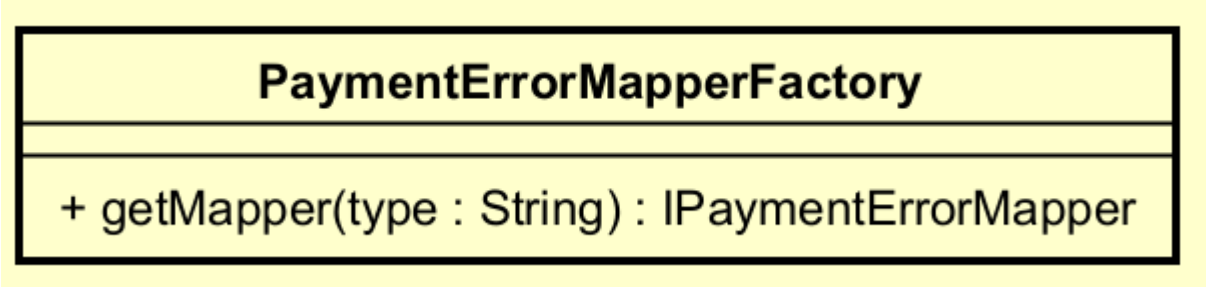


Table 1: Attribute design

Name	Data type	Default value	Description

(None)	*(No attributes stored)*	N/A	This factory does not cache or store instances; it creates new mappers on demand via switch-case.

Table 2: Operation design

Name	Return type	Description (purpose)
getMapper(...)	IPaymentErrorMapper	Returns a payment error mapper based on the given type ("vnpay", "momo"). Throws exception if unsupported.

Method: getMapper(type)

Parameters:

- type: String — The identifier for the payment provider (e.g., "vnpay", "momo").

Exceptions:

- Throws IllegalArgumentException if type is null or not supported.

How to use parameters/attributes:

- Uses switch-case with type.toLowerCase() for case-insensitive matching.
- Instantiates and returns a new concrete implementation of IPaymentErrorMapper:
    - Returns VNPayErrorMapper for "vnpay"
    - Returns MomoErrorMapper for "momo"

5 Design Considerations

<Describe issues which need to be addressed or resolved before attempting to devise a complete design solution. Remember that, you have to refactor your source code to strictly follow the final design>

5.1 Goals and Guidelines

The design of AIMS (An Internet Media Store) is guided by several core goals and principles, ensuring a robust, user-friendly, and efficient platform.

5.1.1 Design Goals

- **High Performance:** The system prioritizes swift responses to user requests, aiming for interactions within 2 seconds under normal load and 5 seconds during peak traffic. This is crucial for a smooth e-commerce experience and user satisfaction, as slow responses can lead to abandonment.
- **Scalability:** AIMS is designed to support up to 1,000 concurrent users seamlessly. This goal is vital for accommodating future growth and ensuring system stability as the user base expands. Achieved through modular architecture, efficient resource management, and load balancing.
- **Reliability and Availability:** The system is engineered to remain operational for at least 300 hours continuously and recover within 1 hour following any downtime. This minimizes service interruptions and maintains user trust, particularly important for an e-commerce platform where transactions are frequent.
- **Security:** Strong security practices are integrated throughout the design, including encrypted data storage, secure user authentication (e.g., JWT), role-based access control, and input validation. This is paramount to protect sensitive user data, payment information, and maintain compliance.
- **User-Friendliness (Intuitive UI/UX):** The user interface is designed to be simple, intuitive, and accessible for non-technical users. This focus ensures a positive user experience, reduces the learning curve, and encourages engagement with the platform.
- **Maintainability:** The system is designed with clear separation of concerns (e.g., Client-Server architecture) and adherence to design principles (e.g., SOLID) to simplify future updates, bug fixes, and feature additions, reducing long-term development costs and efforts.
- **Extensibility:** The architecture is modular to support the addition of new product types, payment methods, and external integrations without significant overhauls. This ensures the system can adapt to evolving business needs and technological advancements.

5.1.2 Design Guidelines and Conventions

- **Coding Standards:** Adherence to established coding conventions (e.g., Java code conventions for Spring Boot backend, React best practices for frontend) is mandatory. This promotes code readability, consistency, and simplifies collaboration among developers.
- **API Design Consistency (RESTful Principles):** All backend APIs follow RESTful principles, using standard HTTP methods (GET, POST, PUT, DELETE), clear

resource naming, and stateless communication. This ensures predictable interactions and easier integration for various client applications.

- **Error Handling and Feedback:** Robust error handling mechanisms are implemented across both frontend and backend. The system provides user-friendly error feedback for invalid inputs, failed transactions, or system issues, improving the user experience and aiding troubleshooting.
- **Input Validation:** Strict input validation is enforced at all entry points (e.g., user registration, product creation, order placement) to prevent common vulnerabilities like SQL injection and cross-site scripting (XSS), and to maintain data integrity.
- **Responsive Design:** The frontend UI is developed using a responsive design approach (e.g., MUI's Grid system) to ensure optimal display and functionality across various devices and screen sizes (web and mobile).
- **Logging and Auditing:** Comprehensive logging is implemented for key system events, user actions, and transactions. This supports auditing, debugging, and provides valuable data for performance monitoring and security analysis.
- **Third-party Integration Standards:** When integrating with external services (e.g., VNPay), strict adherence to their API standards and security protocols is maintained. This ensures reliable and secure communication with external systems.
- **Database Schema Naming Conventions:** Consistent naming conventions (e.g., snake_case for table and column names) are used in the database schema design to improve readability and maintainability.

5.1.3 Design Policies and Tactics

- **Technology Stack Choice:** The choice of React for the frontend and Spring Boot for the backend is a deliberate policy. React offers a component-based approach for building dynamic UIs, while Spring Boot provides a robust framework for enterprise-grade backend services with extensive ecosystem support. PostgreSQL was selected as the RDBMS for its reliability, scalability, and adherence to SQL standards.
- **DTO (Data Transfer Object) Usage:** DTOs are widely used for data exchange between layers (Controller, Service) and clients. This decouples the internal data models from external interfaces, improving security, flexibility, and preventing over-exposure of data.
- **Centralized Configuration Management:** Configuration settings (e.g., database connection strings, API keys) are managed centrally (e.g., using Spring Boot's application.properties/yml) to facilitate easier deployment and environment-specific adjustments.
- **Asynchronous Processing for Non-Critical Operations:** For operations that do not require immediate user feedback (e.g., sending notification emails, processing large data batches), asynchronous processing is considered to prevent blocking the main thread and improve system responsiveness.
- **Caching Strategy:** Caching mechanisms (e.g., in-memory cache, distributed cache) are employed for frequently accessed, non-volatile data (e.g., product categories, popular products) to reduce database load and improve response times.
- **Modular Component Design:** Components are designed to be loosely coupled and highly cohesive, allowing for independent development, testing, and deployment. This is evident in the separation of concerns between Controllers, Services,

Repositories, and DTOs.

5.2 Architectural Strategies

5.2.1 RESTful API Design

The system adopts a RESTful API architecture using Spring Boot for the backend. This approach standardizes communication between the frontend and backend, enabling stateless interactions, clear resource modeling, and easy integration with various clients, including the ReactJS frontend.

5.2.2 Component-Based UI Development

ReactJS is used for the frontend, leveraging its component-based paradigm. This allows for the creation of reusable, modular UI components, improving maintainability, scalability, and development speed. The UI follows modern design standards, with a fixed header, responsive grid layout, and consistent styling.

5.2.3 Object-Oriented Programming (OOP)

The backend is structured using OOP principles, with clear separation of concerns across service, repository, and controller layers. DTOs are used to decouple internal models from API contracts, enhancing maintainability and testability.

5.2.4 Database Management

PostgreSQL is chosen as the relational database for structured data storage, supporting complex queries and ensuring data integrity. JPA/Hibernate is used for ORM, simplifying database interactions and enabling database-agnostic development.

5.2.5 Security and Authentication

JWT-based authentication is implemented to secure API endpoints, providing stateless and scalable user session management. Sensitive configuration (e.g., secrets, passwords) is managed via environment variables and externalized configuration files.

5.2.6 Email and Notification Integration

The system integrates with external email services (Gmail SMTP and SendGrid) for user notifications, password resets, and administrative alerts. This is configured via properties files for flexibility and easy updates.

5.2.7 Error Handling and Validation

Centralized error handling and input validation are implemented in both frontend and backend. The backend uses exception handlers to provide meaningful error responses, while the frontend displays user-friendly error messages.

5.2.8 Testing and Quality Assurance

Comprehensive unit tests are written for both business logic and API endpoints, using Mockito for mocking dependencies. This ensures reliability and facilitates future refactoring.

5.2.9 Scalability and Extensibility

The architecture is designed for future growth, with modular code organization and clear abstractions. New features or integrations (e.g., payment gateways, analytics) can be added with minimal impact on existing components.

5.2.10 Configuration and Environment Management

Application settings, such as database credentials and API keys, are externalized in properties and environment files, supporting easy deployment across different environments (development, staging, production).

5.3 Design and Program Evaluation

<Evaluate your design and describe which levels of coupling and cohesion that your design is at. Give proofs for your assumptions. Explain if there is any special design or exceptions>

<You may show the previous design from which you made improvements to get better levels of coupling and cohesion. You should clarify how and why you did these improvements>

<Does your design follow the SOLID principles if there are new requirements/changing requirements in the future? Give proofs for your assumptions. Explain if there is any special design or exceptions>

<You may show the previous design from which you made improvements to get a better design, which follows SOLID principles in spite of additional requirements. You should clarify how and why you did these improvements>

5.3.1 Cohesion & SRP

Explain in detail for each class in the table.

Table 4. Cohesion & SRP of AIMS

#	*Class Name*	*PIC*	*Cohesion*	*SRP*	*Solution*

1	PlaceRushOrderService.java	Quyen_20226063	High –This class does only one job: managing the rush order process by checking eligibility of delivery address and products. All its methods and dependencies are related to that single responsibility	Yes – Handles only rush order logic. 1 responsibility	
2	ProductRushEligibility	Quyen_20226063	Medium - The main purpose is to evaluate if a product is eligible for rush. However, it also includes logging, null checks, and multiple decision paths, which although useful, slightly dilute the focus of the class. The core logic could be clearer if these concerns were extracted, hence medium cohesion.	Yes – Focused on evaluating one product. 1 responsibility	

3	RushEligibility	Quyen_20226063	High- This is a single-method interface with a very narrow and specific purpose: determining if an input is eligible for rush processing.	Yes – Declares a single method for checking eligibility. A generic interface to abstract rush eligibility logic for any type T. Supports OCP and DIP. 1 responsibility	
4	AddressRushEligibility	Quyen_20226063	Medium - The class focuses only on checking whether an address qualifies for rush. However, there is some cluttered logic around string trimming, null-checks, and business rule verification in a single method.	Yes – Evaluates address only. 1 responsibility	Could benefit from helper methods for null checks and trimming.
5	PlaceRushOrderResponse	Quyen_20226063	High – This class has exactly one purpose: representing the result of a rush order eligibility check. All fields, methods, and logic support this.	Yes – Stores response results. 1 responsibility	

6	RushOrderCheckRequest	Quyen_20226063	High- Acts solely as a request model bundling input (delivery info + products) for checking rush eligibility.	Yes – Bundles input for rush check. 1 responsibility	
7	RushOrderController	Quyen - 20226063	High - Contains only one endpoint and delegates to a clearly defined service. No extra business logic or utility code. Each method and field directly supports the controller's role in handling rush order check requests.	Yes – Controller for rush order checking. 1 responsibility	
8	Product	Quyen_20226063 Huyen_20220073	High – All fields and methods pertain to product identity and rush eligibility.	Yes – Only manages product data (ID, name, rush eligibility). 1 responsibility.	
9	RushAddressProperties	Quyen_20226063	High – Solely binds configuration values from properties.	Yes – Manages external rush delivery address configuration. 1 responsibility.	

10	Order	Duong_20226034 Thao_20226001 Huyen_20220073	Functional Cohesion – All methods and fields are related to the single responsibility: managing an order	No – Just a data class.	
11	PaymentTransaction	Duong_20226034	Functional Cohesion – All fields serve the single purpose of representing a payment transaction.	No – Just a data class.	If future business logic related to payment arises, consider moving those responsibilities into a separate service class like PaymentService.
12	PayOrderService	Duong_20226034	Communicational Cohesion – Methods share common data and purpose (processing payments) but test logic reduces clarity of single-purpose design	Yes – 1 responsibilities: encapsulate business logic for PayOrderController	
13	CancelOrderService	Duong_20226034	Functional Cohesion - The methods and logic in the class work together toward a	Yes – 1 responsibilities: encapsulate business logic for CancelOrderController	

			single, clearly defined purpose: cancelling an order.		
14	IPaymentSystem(Interface)	Duong_20226034	Functional Cohesion — all operations contribute to the unified goal of defining payment system behavior.	Yes, The interface is responsible only for interactions with an external payment system	
15	VNPay Subsystem	Duong_20226034	Functional Cohesion – All fields and methods support the single purpose of integrating with VNPay payment system	Yes, The interface is responsible only for interactions with VNPay System	
16	EmailService	Duong_20226034	Functional cohesion: the class is focused, and all parts contribute meaningfully to its purpose.	Yes - 1 responsibilities: the class handles only email-related tasks and is not mixed with unrelated business logic	
17	CartService	Manh_20225984	**Functional Cohesion** – All methods handles a specific cart management task with clear	**Yes** – Single responsibility: Handle shopping cart business logic exclusively. All methods focus on cart item	**Minor improvements**: Consider extracting cart item validation logic to

			purpose and single responsibility.	manipulation and validation within the cart domain.	separate validator class.
18	DVD, CD, Book	Manh_20225984 Huyen_20220073	**Functional**: Holds fields related to products for each specific type only.	No – Just a data class.	
19	CartItem	Manh_20225984	**Functional**: Holds fields related to cart for only	No – Just a data class.	
20	ProductMapperFactory, ProductMapperHelper	Manh_20225984	**Functional Cohesion:** Choose appropriate mapper for a product	Yes - Single responsibility	
21	CartController	Manh_20225984	**Functional Cohesion** – All methods handle complete HTTP request/response operations for cart management.	**Yes** – Single responsibility: Handle HTTP layer for cart operations.	**Extract parameter extraction**: Create helper methods for extracting customerId and productId to reduce code duplication.
22	ViewProductController	Manh_20225984	**Functional Cohesion** – All methods handle product viewing/retrieval operations.	**Yes** – Single responsibility: Handle HTTP layer for product viewing operations	

23	ProductServiceImpl	Huyen_20220073, Manh_20225984	Functional: All methods are related to CRUD product processing	**Yes**: Focuses only on product business logic	-
24	OrderManagementService	Huyen_20220073	Functional: All methods are related to order management	**Yes:** Focuses only on approve/reject order logic	-
25	ProductFactory	Huyen_20220073	Functional: All methods are related to product creation	**Yes:** Focuses only on product creation logic	-
26	ProductStrategy, BookStrategy, DvdStrategy, CdStrategy	Huyen_20220073, Manh_20225984	Functional: All methods are related to separate logic for each product type	**Yes:** Each implementation focuses only on its own product type logic	-
27	OrderController	Huyen_20220073	Communicational: All methods handle order request/response data	**Yes:** Only responsible for request/response handling	-
28	OrderItem	Huyen_202273	Functional: All methods relate to processing order items	**Yes:** Only handles item information in an order	-
29	OrderMapper	Huyen_20220073	Functional: All methods relate to data transformation	**Yes:** Only responsible for mapping between Order and DTO	-

30	BookMapper, CdMapper, DvdMapper	Huyen_20220073	Functional: All methods relate to data transformation	**Yes:** Only responsible for mapping between Book and DTO	
31	UserController	Quyen_20226063	Medium - All methods relate to user management, but the responsibilities are broad: listing, updating, password changing, deleting.	No – Contains multiple responsibilities (CRUD, change password, sort users)	Could benefit from separating auth or password logic into another controller for SRP.
32	UserCreationRequest	Quyen_20226063	High- Simple DTO with only creation-relevant fields. Clean and focused.	Yes – Only stores user creation input. 1 responsibility	
33	UserPasswordRequest	Quyen_20226063	High- Only concerned with password inputs. No unrelated logic or fields.	Yes – Focused solely on password change. 1 responsibility	
34	UserUpdateRequest	Quyen_20226063	High - Clean separation, fields used only for updates. No excess data.	Yes – Only used for update input. 1 responsibility	
35	UserResponse	Quyen_20226063	High - Pure output model for users. No logic or	Yes – Only for outputting user info. 1 responsibility	

			unrelated fields.		
36	UserService	Quyen_20226063	High - Defines responsibilities cleanly. Single focus: user operations.	Yes – Defines contract for user operations. 1 responsibity	
37	UserServiceImpl	Quyen_20226063	Medium - While user business logic is handled correctly, this class also sends emails and logs notifications	Partial – Contains business logic + email logic. 2 responsibilities	Implements user logic: creation, update, deletion. Includes email sending, which could be extracted to a separate class to increase cohesion and SRP adherence.
38	UsersRepository	Quyen_20226063	High- The interface only declares methods related to user data persistence.	Yes – Handles data access only	
39	UserStatus	Quyen_20226063	High - Enum is focused entirely on modeling one domain concept	Yes – Represents user status only	
40	UserType	Quyen_20226063	High- Enum is focused entirely on modeling one	Yes – Represents user roles only	

			domain concept		
41	CalculateFeeService	Thao_20226001	**Functional Cohesion** – Excellent. All methods within the class (calculateDeliveryFee, calculateRushOrderFee, calculateSubtotal, etc.) are directly involved in and essential for the single, well-defined purpose of calculating fees for an order.	**Mostly Yes.** The class has one primary responsibility: **calculating costs**. However, it also embeds specific business rules (e.g., hardcoded city names and fee values), which slightly blurs the lines. The core function remains singular.	To improve maintainability and strictness, externalize all constants (fees, weights, city names) into a configuration file (e.g., application.properties). This allows business rules to be changed without modifying the service's source code, better adhering to the Open/Closed Principle.
42	PlaceOrderService	Thao_20226001	**Sequential Cohesion (Low)** – The class methods are grouped because they must be executed in a specific order to complete the "place order" process (create order -> save delivery info -> save items -> update stock -> calculate total). This is a	**No.** As your comments correctly identify, this class violates SRP by having multiple responsibilities: 1. **Order/Entity Persistence**: Creates and saves Order, OrderItem, and DeliveryInfo entities. 2. **Inventory Management**: Checks for and updates product stock levels.	**Refactor into specialized services.** Keep PlaceOrderService as a high-level **coordinator/facade**, but delegate the actual work: • Move inventory logic (checkInventoryAvailability, updateProductStocks) to a dedicated

			lower form of cohesion because the individual tasks (database interaction, stock management) are functionally separate.	3. **Process Orchestration**: Manages the high-level workflow of placing an order.	InventoryService. • Move delivery info creation (createAndSaveDeliveryInfo) to a DeliveryService. • The PlaceOrderService will then be lean, with a single responsibility: orchestrating calls to these other services in the correct sequence.
43	AuthService	Thao_20226001	**Functional Cohesion (High)** – Excellent. All elements in this class, including its dependencies (AuthenticationManager, JwtUtils) and its single public method, cooperate to perform one well-defined task: authenticating a user and	**Yes** – The class has one clear and distinct responsibility: to handle the user authentication process. It doesn't manage user registration, profile updates, or other unrelated tasks.	**None needed.** The class is well-designed, cohesive, and adheres strictly to the Single Responsibility Principle. It correctly delegates specific concerns like password validation and token creation to other specialized components.

			issuing a session token.		

5.3.2 Coupling & Other SOLID

Explain in detail for each problem in the table. Each problem, you may provide a class diagram before and after the improvement.

Table 5. Coupling & other SOLID of AIMS

#	*Problem*	*PIC*	*Location*	*Solution*
	Control coupling	TuấnNH	Class A and class B	Create a new class C as a superClass A and B…
	<Any bad coupling level or any SOLID violation>	<person in charge>	<list of class or you may provide a class diagram if necessary>	
1	Medium Cohesion (Contains business logic + email logic.)	Quyen_20226063	UserServiceImpl	Can extract logic to separate classes to increase cohesion and SRP adherence.
2	Control Coupling – logic depends on how RushEligibility<T> is implemented	Quyen_20226063	PlaceRushOrderService	Add a coordinator like RushPolicyEvaluator to delegate address and product checks in a more declarative way.
3	Control Coupling (Medium) - hard coded logic depends on	Duong_20226034	PayOrderService	Suggestion: In future, Create Refund Strategy Interface, Refund Strategy Factory

	choosing PaymentMethod			
4	Violate OCP	Duong_20226034	PayOrderService	Strategy Pattern for Payment Processing, Configuration-Based URL Management,
5	Violate DIP	Duong_20226034	PayOrderService	Create Interface for Repositories, Services
6	Missing authorization check in ManagerController – Only authentication is handled, role is not checked	Huyen_20220073	ManagerController	Use Spring Security with @PreAuthorize annotation to verify roles
7	**Control Coupling:** ProductFactory tightly controls which strategy is used for each product type, making extension harder.	Huyen_20220073, Manh_20225984	ProductFactory, ProductStrategy, BookStrategy, CdStrategy, DvdStrategy	Use dependency injection and configuration for strategy registration. Decouple factory from concrete strategies.
8	**SRP Violation:** OrderController handles both order retrieval and status management, with manual DTO conversion.	Huyen_20220073	OrderController, OrderStatusManager, IOrderManagementService	Move status management to a dedicated service. Use mappers for DTO conversion.

9	Interface pollution – IOrderManagementService contains unrelated methods	Huyen_20220073	IOrderManagementService	Apply Interface Segregation by splitting into smaller, more focused interfaces
10	Encapsulation Violation: OrderStatusManager directly manipulates Order entity fields.	Huyen_20220073	Order, OrderStatusManager	Use encapsulated methods in Order for status changes, hide direct field access.
11	**Violation of Dependency Inversion Principle (DIP)** – The service directly depends on concrete repository classes instead of abstractions (interfaces). This couples the business logic directly to the data access implementation.	Thao_20226001	PlaceOrderService	**Depend on Abstractions.** Refactor the service to depend on repository **interfaces** (e.g., IProductRepository, IOrderRepository). This decouples the service from the specific data access technology and makes testing with mock objects significantly easier.
12	**Violation of Single Responsibility Principle (SRP) / Low Cohesion** – The class acts as a "god object" for placing an order, handling responsibilities like inventory	Thao_20226001	PlaceOrderService	**Refactor into Specialized Services.** Keep PlaceOrderService as a coordinator but extract the logic into new classes: InventoryService (for stock updates/checks) and OrderPersistenceSer

	management, entity persistence, and workflow orchestration.			vice (for saving order-related entities). PlaceOrderService will then only be responsible for calling these services in the correct sequence.
13	**Violation of Open/Closed Principle (OCP)** – The service has hardcoded business rules (fees, weights, city names). Adding a new shipping rule or changing a fee requires modifying the source code.	Thao_2022 6001	CalculateFeeService	**Externalize Configuration**. Move all magic numbers and strings to an external .properties file or a database table. Load these values at runtime using a @ConfigurationPrope rties class. This allows business rules to be changed without altering the code.
14	Multiple Responsibilities (SRP Violation)	Quyen_202 26063	PlaceRushOrderServ ice	Apply SRP by extracting each responsibility (eligibility logic, DTO building) into its own class.
15	**Functional Coupling** **SRP+IDP**	Manh_2022 5984	CartService - 4 repository dependencies	Extract UserValidationService
16	**Stamp Coupling ISP**	Manh_2022 5984	CartController - Receives full DTO but uses parts	Create specific request DTOs

5.4 Design Patterns

<Do you use any design patterns for your design? If yes, describe detailly why you use those design patterns? Describe in detail on the solutions and how to implement each design pattern>

Several design patterns are used in our system design to improve modularity, maintainability, and scalability. Below are the design patterns applied, along with explanations of why and how they are used:

1. **Factory Pattern**

- PaymentErrorMapperFactory: Used to return the appropriate error mapper for each payment subsystem (e.g., VNPay, MoMo). This centralizes the logic and makes it easy to add new payment systems.

- PaymentSystemFactory: Determines which payment service to use based on the user's selection. This decouples the instantiation logic from business logic.

- CartServiceFactory: Dynamically switches between `apiCartService` and `localCartService` depending on whether the user is logged in. This enables flexible cart management for both authenticated and guest users.

- ProductFactory: Used to obtain the correct product strategy (e.g., BookStrategy, CdStrategy, DvdStrategy) based on the product type. This supports a clean separation between internal logic and API contracts, and facilitates consistent handling of different product types.

2. **Strategy Pattern**
   Applied in product-related operations such as view, search, add, update, and delete. Different strategies (BookStrategy, CdStrategy, DvdStrategy) implement a common ProductStrategy interface, allowing interchangeable logic based on product type. This pattern promotes flexibility and makes it easier to maintain or extend product-specific logic without changing the main workflow.

3. **State Pattern**

- Used in the `OrderStatusManager` to manage valid transitions between order statuses (e.g., PENDING to APPROVED or REJECTED).
- This ensures that only valid state changes are allowed, centralizing validation logic and preventing inconsistent order behavior.

4. **Interface Segregation Pattern(ISP)**

- Applied through the `IOrderManagementService` interface, which provides focused methods for order approval and rejection.

- This avoids the fat interface anti-pattern and ensures that clients only depend on relevant methods, following SOLID design principles.