HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

School of Information and communications technology

Software Design Document

AN INTERNET MEDIA STORE

Subject: ITSS SOFTWARE DEVELOPMENT

Group 18

Nguyen Duc Thang - 20215244

Pham Tuan Tai - 20215240

Nguyen Viet Thang - 20215245

Nguyen Tien Thanh - 20215243

Vo Van Thanh - 20215242

Hanoi, 06/2024

Table of Contents

1 Introduction

This document represents the design of An Internet Media Store (AIMS) software. This document will guide you through the detailed description, architectural design, user interface design as well as installation guidelines for AIMS software.

1.1 Objective

The software is designed to automate key operations of an online store including product management, online ordering and processing orders.

AIMS is designed to necessitate the needs for customers who wish to search and purchase any products from AIMS. It facilitates the process of selecting, ordering, paying and displaying purchases for desired customers.

AIMS satisfies the demand of administrator to create, modify, delete or display different products as well as orders in the system.

1.2 Scope

AIMS is a software that designated to meet all the objectives of an online store which facilitates the need of various users.

The AIMS will automate core operations of an online store for selling, managing media products including: books, cds, and dvds. It will manage the store's inventory, facilitate purchase transactions at point-of-sale, process customer orders.

Specifically, key capabilities of the software include:
- Managing product catalog in each category
- Enable customers to search for books, cds and dvds and select products to purchase
- Place order and make payments online

The objectives of developing this system are to streamline online store operations, improve customer experience, and reduce reliance on manual processes. It also aims to help store managers optimize management capabilities over product and drive higher sales.

1.3 Glossary

No	Term	Explanation	Example	*Note*

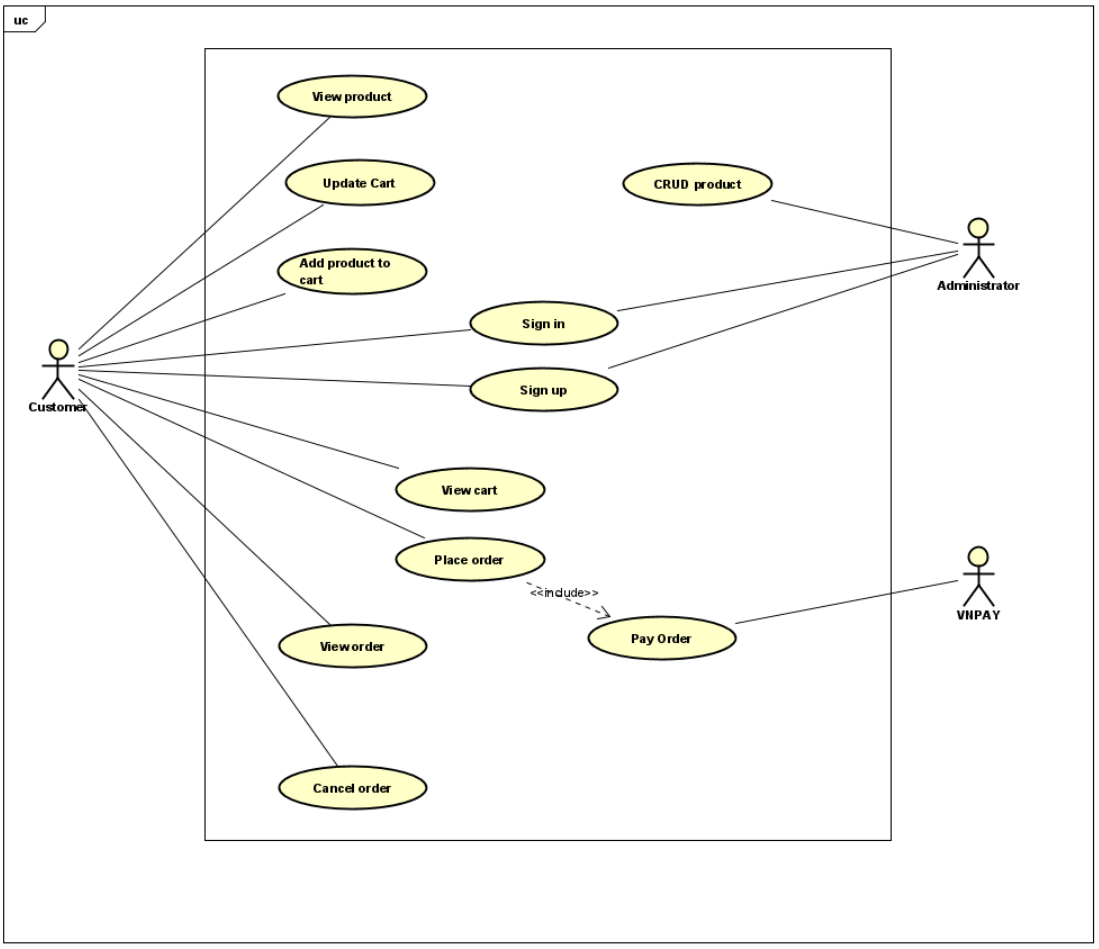
1	AIMS	AIMS stands for "Automated Inventory Management System". It is a software system designed to help businesses manage their inventory and streamline their operations		
2	E-commerce	E-commerce (electronic commerce) refers to the buying and selling of goods and services over the internet.		
3	Customer	A customer is a person or organization that purchases goods or services from a business.		
4	Credit Card	A credit card is a plastic card issued by a bank or financial services company that allows cardholders to borrow funds to purchase goods and services. The borrowed funds must be repaid with interest.		
5	*CRUD*	*Four basic functions, namely Create, Retrieve, Update, Delete*		

1.4 References

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



The AIMS will be a web-based application, with both a backend system and a frontend website. The overall system architecture follows a multi-tiered design pattern with presentation, business logic and data tiers.

Customers and manager will access the system via a web-based user interface,. The system will integrate with a database to retrieve products and order information as well as with payment gateways to process online transactions.

2.2 Assumptions/Constraints/Risks

2.2.1 Assumptions

The following assumptions and dependencies form the context for the AIMS design:

- The application will run on a virtualized server environment with Windows/Linux operating system. It assumes a continuously available network and server infrastructure.
- It depends on a relational database like MySQL to store and manage bookstore data. The database is expected to be fault-tolerant with regular backup and restore capabilities.
- Customers & managers are assumed to have access to modern web browsers on desktops/laptops and mobile devices to use the web interface.
- Payment gateways being used are expected to provide transaction processing according to standard payment protocols.
- The application is intended to serve a single retail bookstore outlet initially. The design keeps room for multi-store support in the future.
- End users are assumed to have basic computer literacy. Training will be provided for complex processes.

2.2.2 Constraints

- Hardware or software environment

The system will run on commodity server hardware within a fixed storage and performance, this may cause difficulties in future integration or development and low performance when excessive number of users using the system simultaneously.

- End-user environment

Customer-facing applications need to support common web browsers and be responsive on mobile devices. This impacts the frontend framework selection.

- Availability or volatility of resources

Downtime for the system needs to be minimized since it handles critical operations. High-availability architecture is required.

- Interoperability requirements

External systems integrate using standard APIs for metadata exchange. Interfaces need to support expected formats and protocols.

- Interface/protocol requirements

Support modern web standards like HTML5, CSS3
Responsive design to adapt on different device screens

- Data repository and distribution requirements
- Security requirements (or other such regulations)

Sensitive customer, financial and inventory data needs protection. Strong authentication, authorization and data protection measures are mandated.

- Memory or other capacity limitations

The system has a fixed memory and Central Processing Unit, therefore may impact future integration process or performance with excessive concurrent users.

- Performance requirements

The system needs to support concurrent usage peaks during shopping seasons/festivals without degradation. Scalability is key.

- Network communications

Dependence on network for accessing central systems. Offline functionality may be needed at branch levels during outages.

- Verification and validation requirements (testing)

Software changes need rigorous validation before production. Automated testing pipelines are required.

- Other means of addressing quality goals

2.2.3 Risks

2.2.3.1. Risk: Security vulnerabilities due to bugs or flaws

Mitigation: Follow secure coding standards, implement input validation, use framework security features like Spring Security. Perform regular code reviews and vulnerability testing.

2.2.3.2. Risk: Downtime impacting business operations

Mitigation: Implement high availability architecture with load balancing, clustering. Enable failover and auto-scaling. Take regular database/application backups.

2.2.3.3. Risk: Integration failures with third parties

Mitigation: Use versioned APIs with fallback logic. Implement thorough testing for external interfaces. Define SLAs for dependent services.

2.2.3.4. Performance degradation:

With high traffic, system performance may degrade due to resource contention. This can impact user experience and functionality.

Mitigation: Implement auto-scaling architecture using cloud infrastructure for elastic capacity.

3 System Architecture and Architecture Design

3.1 Architectural Patterns

Three-tier architecture is a widely adopted architectural pattern in software development that organizes applications into three distinct layers: Presentation, Business Logic, and Data Access. This pattern, especially relevant for projects using technologies like ReactJS for the frontend and Spring Boot for the backend, ensures a clear separation of concerns, which enhances maintainability, scalability, and flexibility of the application.

- The Presentation Tier, implemented with ReactJS, serves as the user interface layer. It is responsible for rendering the visual components of the application, managing client-side logic, and facilitating user interactions. This tier handles routing within the single-page application, manages local and global state using tools like the Context API or Redux, and communicates with the backend via HTTP requests using libraries like Axios or Fetch API.

- The Business Logic Tier, powered by Spring Boot, forms the core of the application where the business rules and application logic reside. This tier includes the controller layer, which exposes RESTful endpoints for client requests and orchestrates the responses, the service layer, which contains the business logic and interacts with the data access layer, and the repository layer, which abstracts the data persistence mechanism. This layer ensures that the application logic is decoupled from the data storage and presentation concerns, allowing for independent development and maintenance.

- The Data Access Tier encompasses the database and the data management framework. This tier is responsible for the storage, retrieval, and manipulation of application data. Using Spring Data JPA or Hibernate, it provides an abstraction over the underlying database (which could be relational, like MySQL or PostgreSQL, or NoSQL, like MongoDB), enabling developers to perform CRUD operations and complex queries without delving into database-specific syntax. This tier manages entity relationships, transactions, and data integrity, ensuring that the data layer is robust and reliable.

- The three-tier architecture enhances the overall robustness of the application by enforcing a modular structure where each tier can evolve independently. This modularity facilitates scalability, as each tier can be scaled horizontally or vertically based on load and performance requirements. For instance, the frontend can be optimized for performance and deployed on a content delivery network (CDN) for faster load times, while the backend services can be distributed across multiple servers or containers to handle increasing user requests.

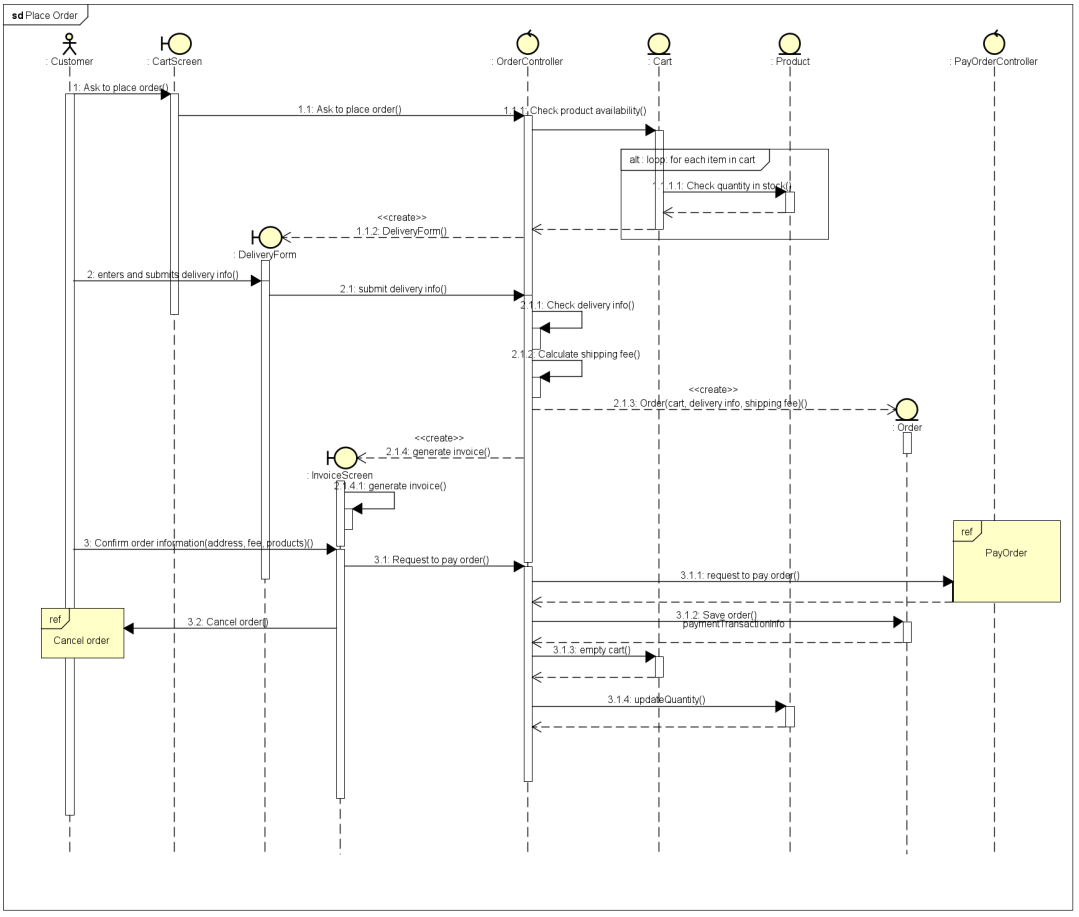
- Additionally, the separation of concerns allows for better maintainability; changes in the business logic or data access layers do not necessitate alterations in the presentation layer, and vice versa. This decoupling also aids

in testing, as each layer can be tested independently, ensuring comprehensive coverage and easier debugging. Furthermore, this architecture supports reusability, as the business logic and data access code can be reused across different projects or parts of the application, promoting code efficiency and reducing duplication.

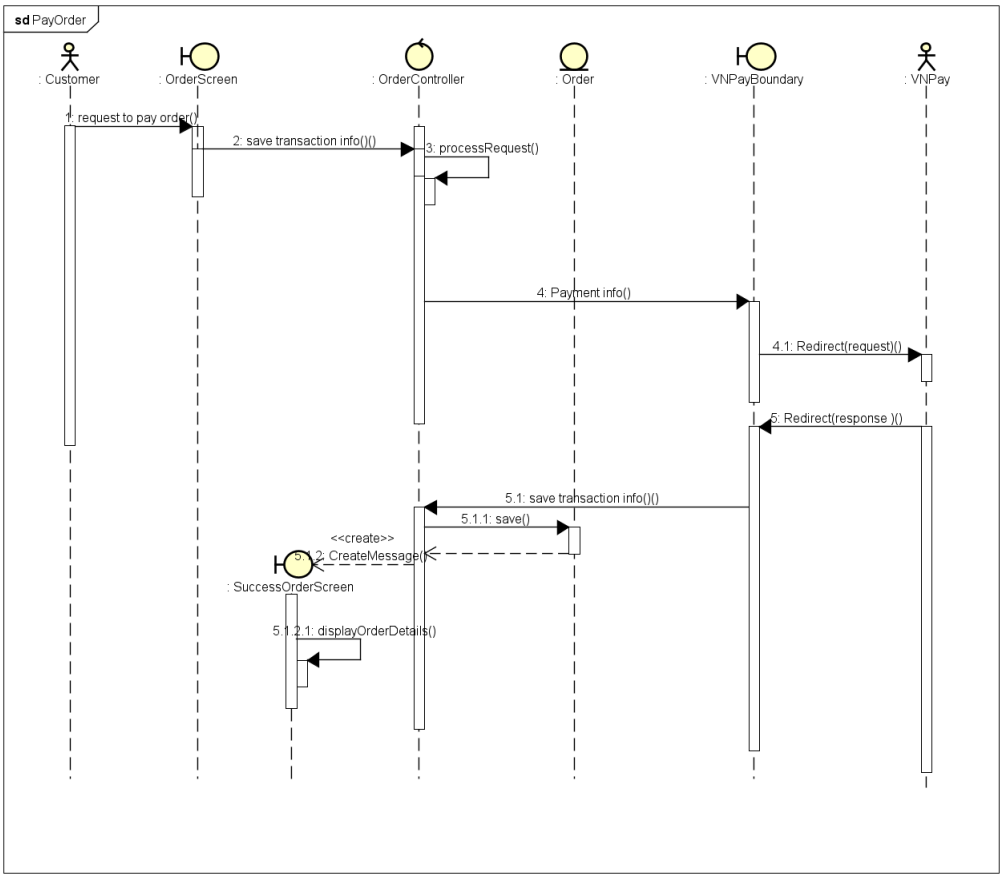
Overall, the three-tier architecture, when implemented with ReactJS and Spring Boot, provides a robust, scalable, and maintainable framework for building modern web applications, aligning with best practices and ensuring a high-quality, modular codebase.

3.2 Interaction Diagrams.

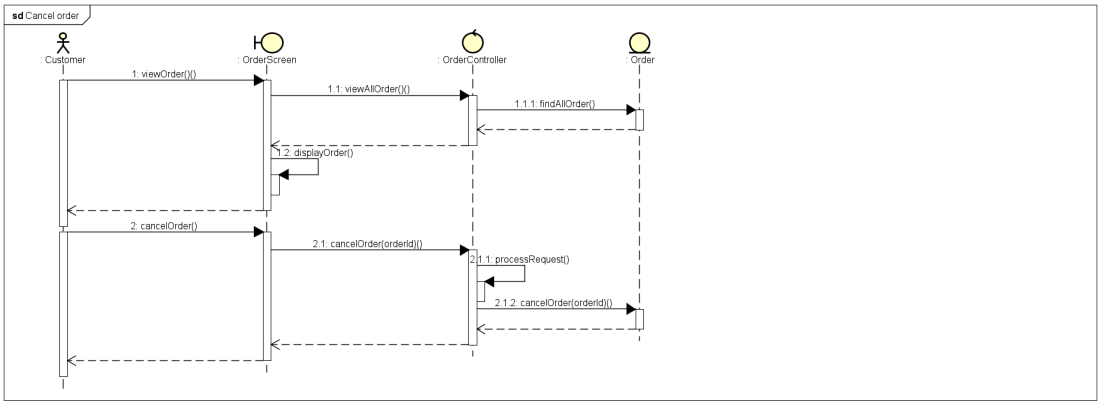
3.2.1. Place order sequence diagram.



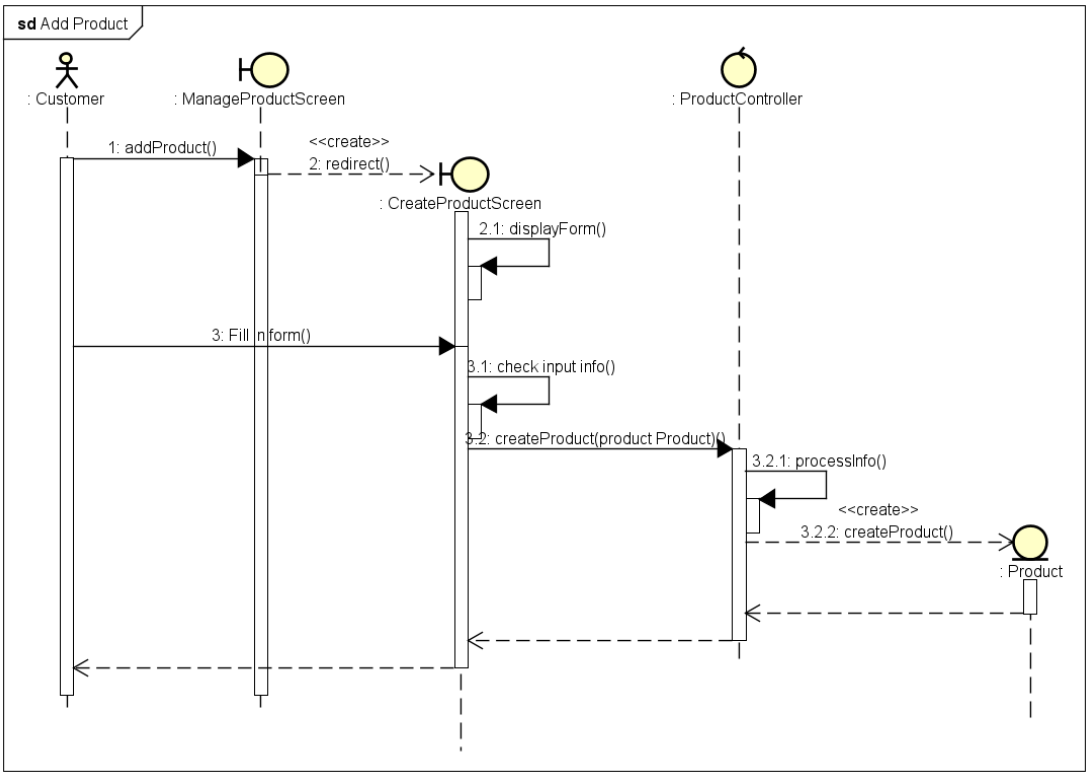
3.2.2. Pay Order sequence diagram



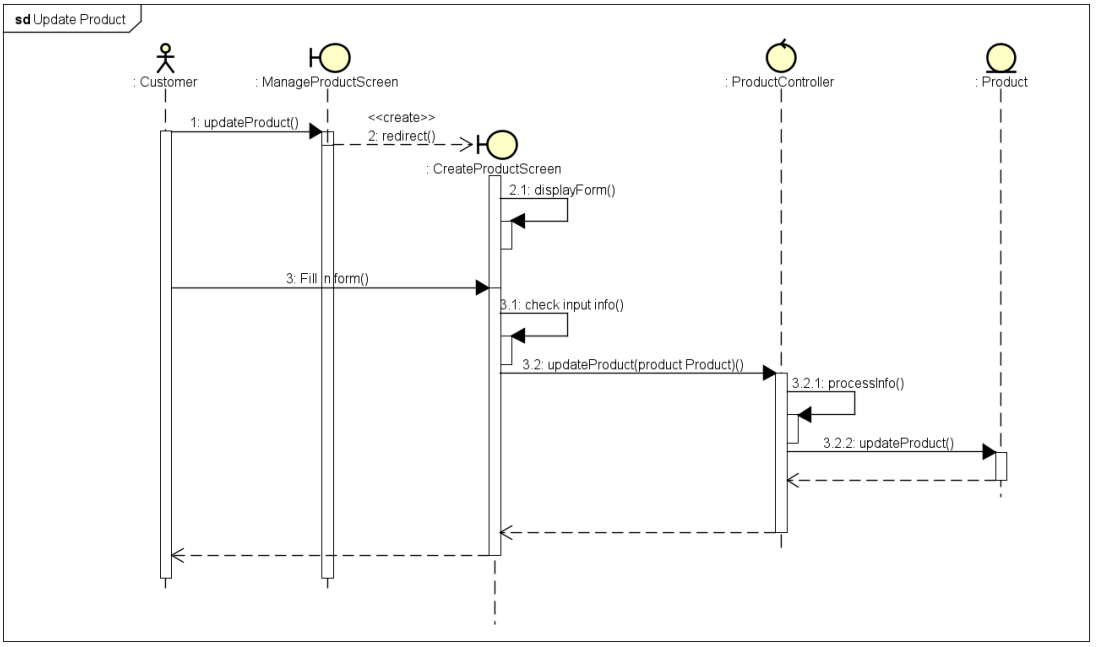
3.2.3. Cancel Order sequence diagram



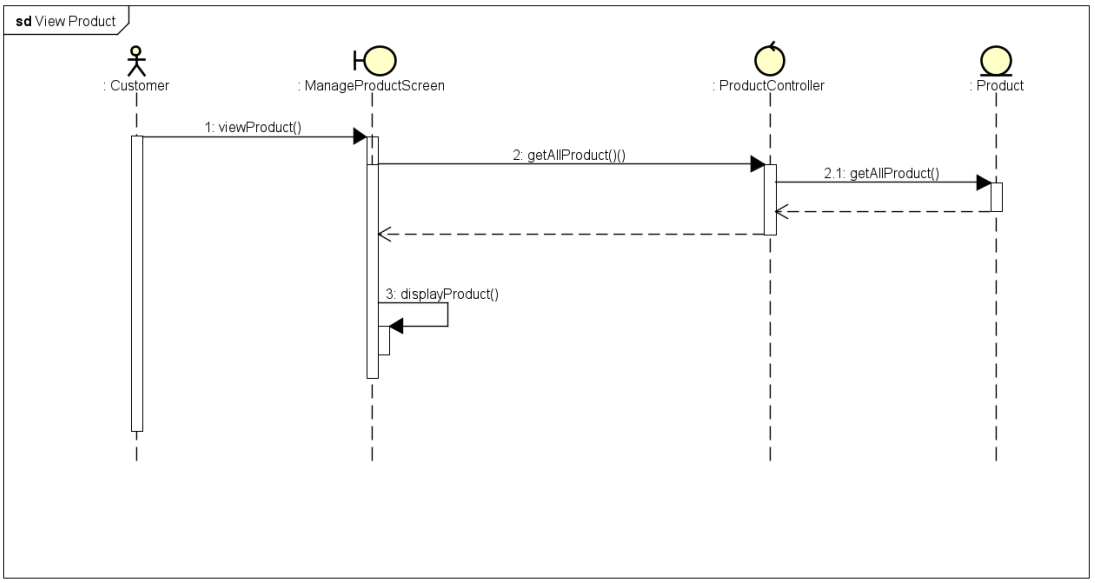
3.2.4. Add product sequence diagram



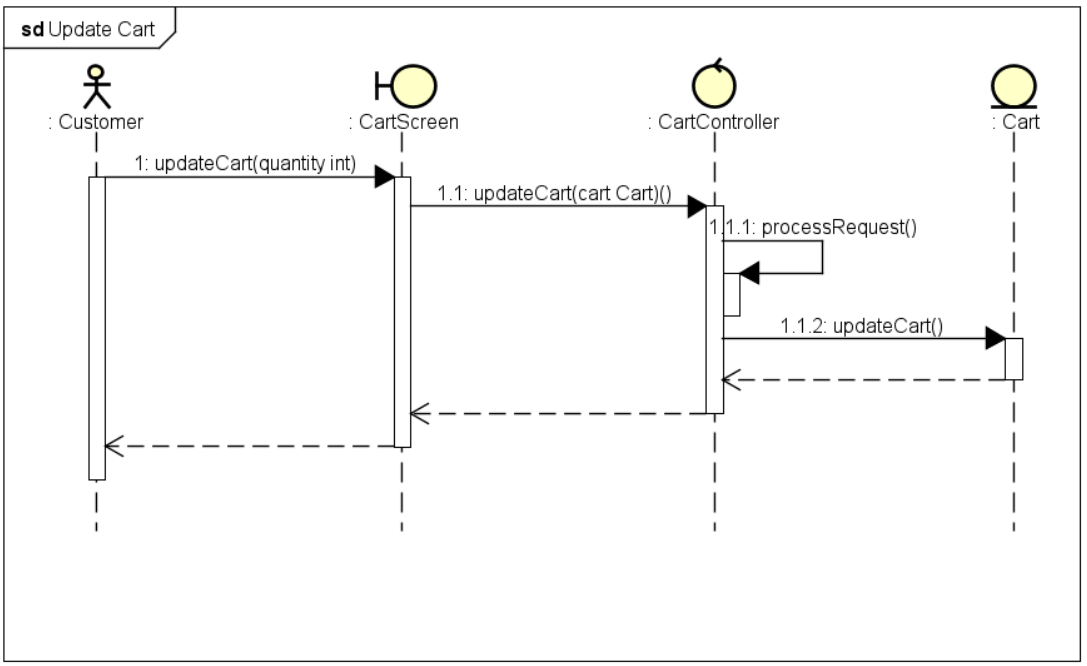
3.2.5. Update Product sequence diagram



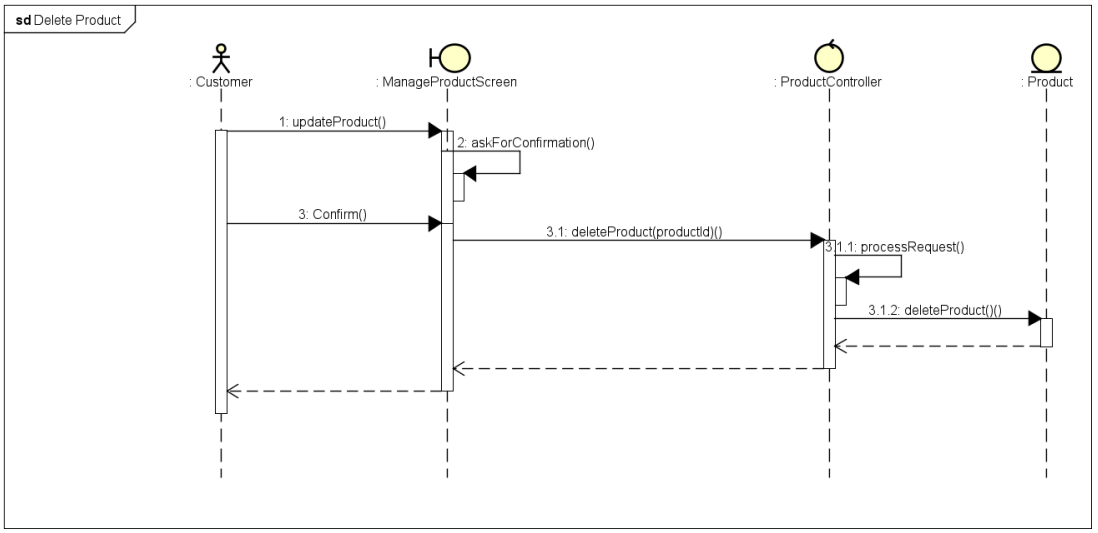
3.2.6. View Product sequence diagram



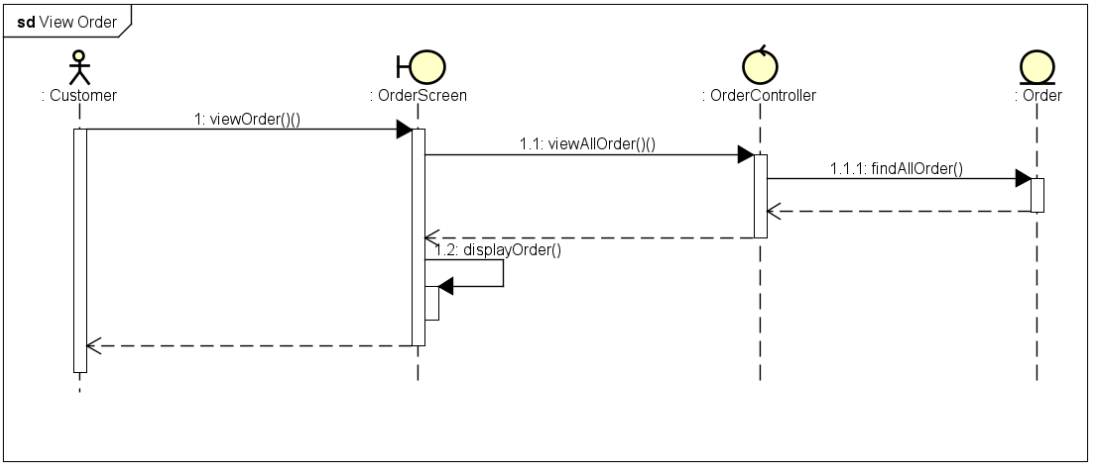
3.2.7. Update cart sequence diagram.



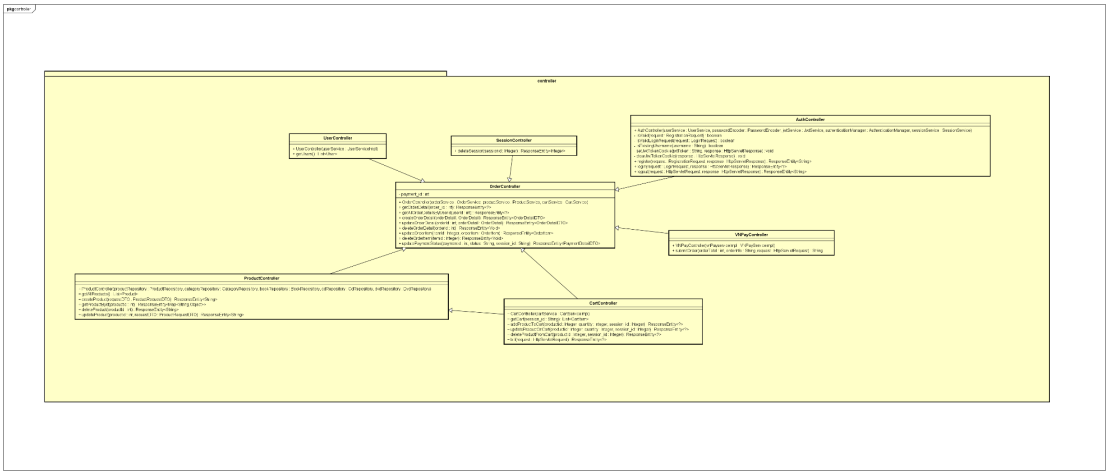
3.2.8. Delete Product sequence diagram.

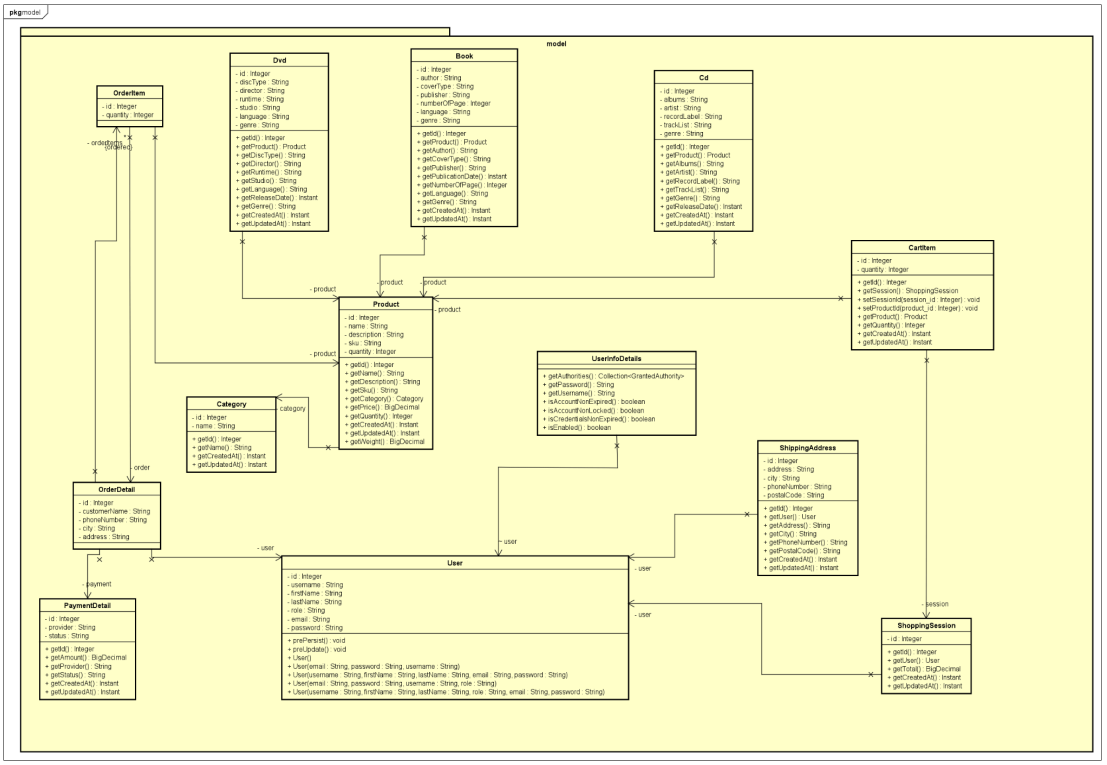


3.2.9. View order sequence diagram

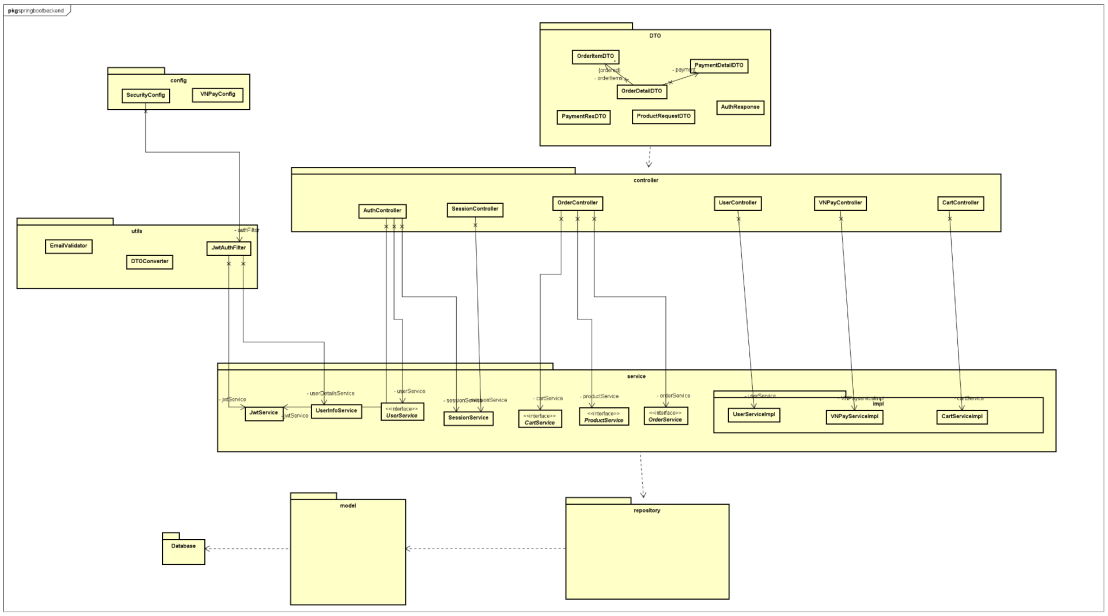


3.3 Analysis Class Diagrams


pkg controller
controller
UserController
SessionController
AuthController
OrderController
VNPayController
ProductController
CartController


pkg model
model
OrderItem
Dvd
Book
Cd
CartItem
Product
UserInfoDetails
Category
ShippingAddress
OrderDetail
User
PaymentDetail
ShoppingSession

3.4 Unified Analysis Class Diagram



3.5 Security Software Architecture

In the context of a three-tier architecture for an e-commerce platform using ReactJS and Spring Boot, focusing on HTTPS and JWT (JSON Web Tokens) provides a robust security foundation. These technologies ensure secure communication and reliable authentication, which are critical for protecting user data and maintaining the integrity of the application.

In this architecture, we aim to secure the system by implementing HTTPS for secure communication and JWT for secure, stateless authentication. This ensures that data transmitted between the client and server is encrypted and that user authentication is handled securely and efficiently.

a, Presentation Tier (ReactJS)

The Presentation Tier is the user interface layer where users interact with the application. Security measures here focus on protecting user data and ensuring secure communication with the backend.

- **HTTPS:** All communication between the client (ReactJS) and the server (Spring Boot) is encrypted using HTTPS. This protects data from being intercepted or tampered with by attackers. Ensure that the server has a valid SSL/TLS certificate and that the ReactJS application enforces HTTPS by redirecting HTTP requests to HTTPS.
- **JWT Authentication:** Upon successful login, the server issues a JWT to the client. The token contains user authentication information and is signed by the server. The client stores the JWT securely, typically in local storage or a

secure HTTP-only cookie. For each subsequent request, the client includes the JWT in the Authorization header using the Bearer schema (Authorization: Bearer <token>).

b, Business Logic Tier (Spring Boot)

The Business Logic Tier handles the core application logic and must ensure that only authenticated and authorized users can access specific resources.

HTTPS: The Spring Boot server enforces HTTPS for all endpoints, ensuring that data transmitted between the server and clients is encrypted. Configure the server to redirect all HTTP requests to HTTPS and ensure the SSL/TLS certificate is properly set up.

JWT Authentication:

- Issuing Tokens: Upon successful user authentication (e.g., login), the server generates a JWT, which includes user-specific claims such as user ID, roles, and expiration time. The JWT is signed using a secret key or an asymmetric key pair to ensure its integrity and authenticity.
- Validating Tokens: For each request that requires authentication, the server verifies the JWT included in the Authorization header. The server checks the token's signature, expiration time, and any other relevant claims to ensure it is valid.
- Role-Based Access Control: Use the claims in the JWT to implement role-based access control (RBAC). This ensures that users can only access resources and perform actions that they are authorized for.
- Token Expiration and Refresh: Implement token expiration to limit the lifespan of JWTs. This reduces the risk of token misuse. Provide a mechanism for refreshing tokens to maintain user sessions without requiring frequent logins.

c, Data Access Tier (Database)

The Data Access Tier involves securing data storage and ensuring that only authorized access to data is allowed.

- **Secure Data Storage:** Although HTTPS and JWT do not directly interact with the database, their proper implementation ensures that only authenticated and authorized requests reach the data access layer. Implement additional database security measures such as encryption at rest, strict access controls, and regular security audits.
- **Access Control:** Use the information in JWTs to enforce fine-grained access controls at the data access layer. Ensure that database queries respect user permissions and only return data that the user is authorized to access.

By implementing HTTPS and JWT in the three-tier architecture, we ensure that our e-commerce platform has a robust security foundation. HTTPS guarantees that all data transmitted between the client and server is encrypted, protecting it from interception and tampering. JWT provides a secure, stateless authentication mechanism that simplifies the process of verifying user identities and enforcing access controls. Together, these technologies help maintain the security, integrity, and reliability of the application, aligning with our project aim of delivering a secure and user-friendly e-commerce experience.

4 Detailed Design

4.1 User Interface Design

4.1.1 Screen Configuration Standardization

a, Display

Number of colors supported: 16,777,216 colors.

Resolution: Responsive

- Extra Small (XS): Up to 576 pixels
- Small (SM): 576 pixels to 768 pixels
- Medium (MD): 768 pixels to 992 pixels
- Large (LG): 992 pixels to 1200 pixels
- Extra Large (XL): 1200 pixels and above

b, Screen

- **Location of standard buttons**: At the bottom (vertically) and in the middle (horizontally) of the frame. Standard buttons include actions such as Submit, Cancel, Back, and other primary navigation or form action buttons.
- **Location of the messages:** Starting from the top vertically and in the right horizontally of the page down to the bottom.
- **Display of the screen title:** The title is located at the top of the frame in the middle. Titles should clearly indicate the purpose of the page, such as "Home", "Login", "Product Details", etc.
- **Consistency in expression of alphanumeric numbers:** Comma for separator of thousand while strings only consist of characters, digits, commas, dots, spaces, underscores, and hyphen symbols.

c, Control

- Size of the text: Medium size (mostly 14px). Headlines and important text may use larger sizes (e.g., 24px for section headers)
- Font: Segoe UI . Uniform font usage across the website for consistency and readability
- Color: #000000. Primary text color ensuring good contrast and readability.
- Input check process: Should check if it is empty or not. Next, check if the input is in the correct format or not (e.g., email format, phone number format). Provide real-time validation feedback to users where applicable.
- Sequence of moving the focus: Each screen will be separated; no stack frames will be used. Pop-up messages (such as modals) are considered separate screens and will disable interaction with the underlying screen until closed.
- Use logical and intuitive tab order for navigation through form fields.

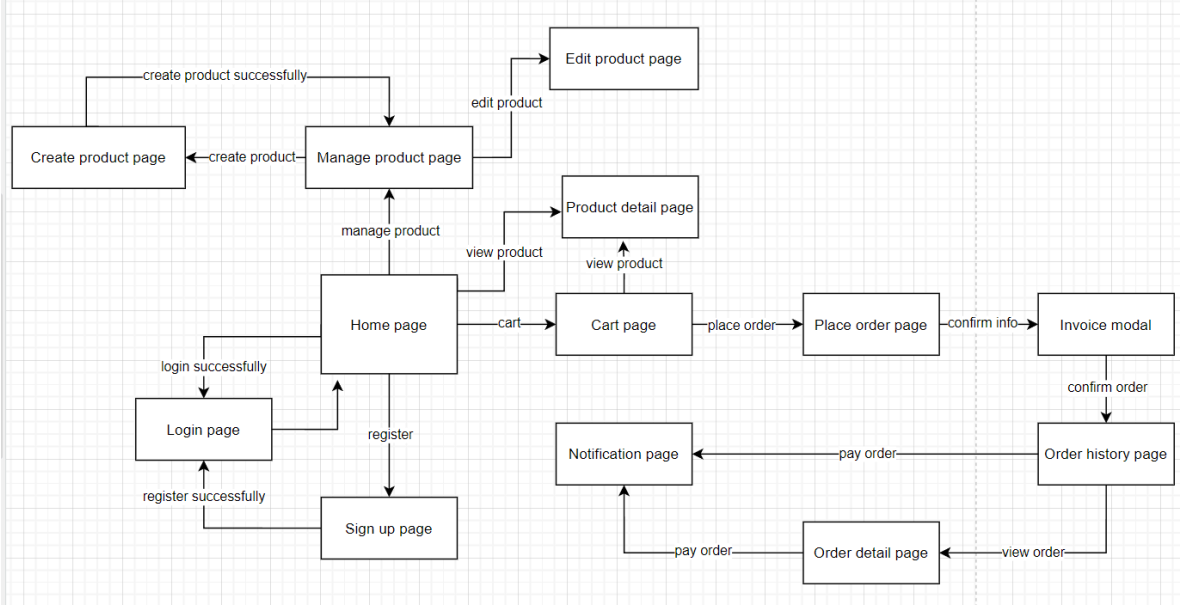
d, Direct input from the keyboard

- There will be no shortcuts: Users will navigate through clickable buttons and links.
- There are back buttons to move back to the previous screen: Back buttons should be prominently placed and clearly labeled.
- There is the close button “Cancel” located at the bottom of the screen: For modal dialogs and pop-ups.

e, Error

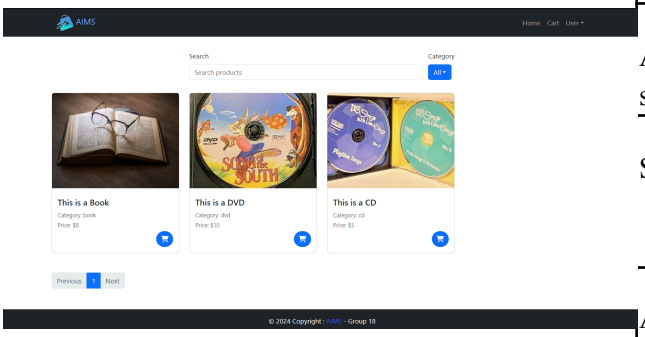
A message will be sent to notify the users what the problem is.

4.1.2 Screen Transition Diagrams



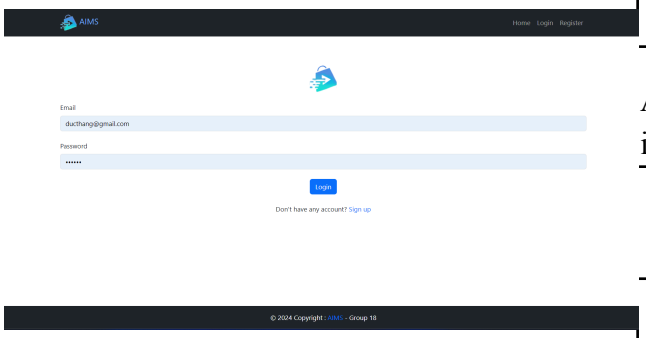
4.1.3 Screen Specifications

a, Home page

AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Home page	18/6/2024			Nguyen Duc Thang
		Control	Operation	Function	
		Area for displaying search content	Initial	Display the search content	
		Search button	Click	Search for items which have title matching with search content	
		Area for displaying items in the shop	Initial	Display the items in the shop	
		Add to cart button	Click	Add item to the cart	

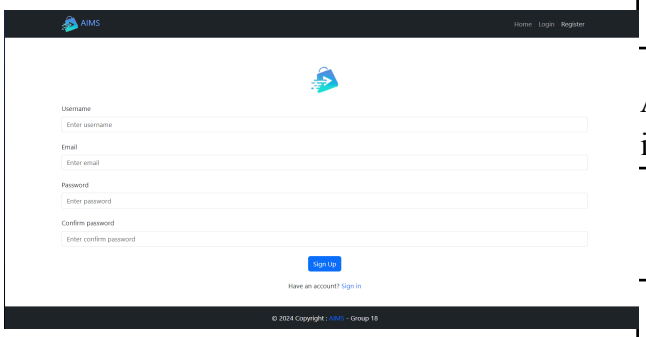
	Pagination	Click	Display another pages of items in the shop
	Area for displaying item information	Initial	Display title, price, category of that item
	Category dropdown	Clock	Search for items which have category matching with option in dropdown

b, Login page

AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Login page	18/6/2024			Nguyen Duc Thang
		Control	Operation	Function	
		Area for display user information	Initial	Display some questions to request customer to fill	
		Login button	Click	Perform login action	
		Signup text	Click	Navigate to sign up page	

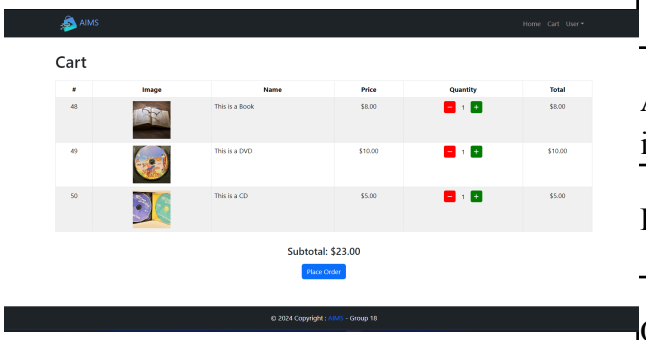
c, Sign up page

AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Sign up page	18/6/2024			Nguyen Duc Thang

	Control	Operation	Function
	Area for display user information	Initial	Display some questions to request customer to fill
	Sign up button	Click	Perform signup action
	Sign in text	Click	Navigate to sign in page

d, Cart page

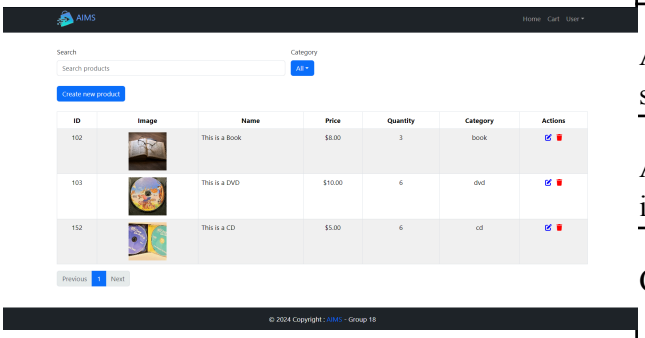
AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Cart page	18/6/2024			Nguyen Duc Thang

	Control	Operation	Function
	Area for display items in the cart	Initial	Display the media with the corresponding information
	Plus button	Click	Increase the amount of item that user might want to add to cart
	Quantity text	Click	Open update modal
	Item image	Click	Navigate to product detail page
	Minus button	Click	Decrease the amount of item that user might want to decrease
	Update modal	Initial	Update quantity of item that user might want to update
	Delete modal	Initial	Delete the item that user want to edit
	Area for displaying the subtotal	Initial	Display the subtotal

	Place order button	Click	Navigate to place order page

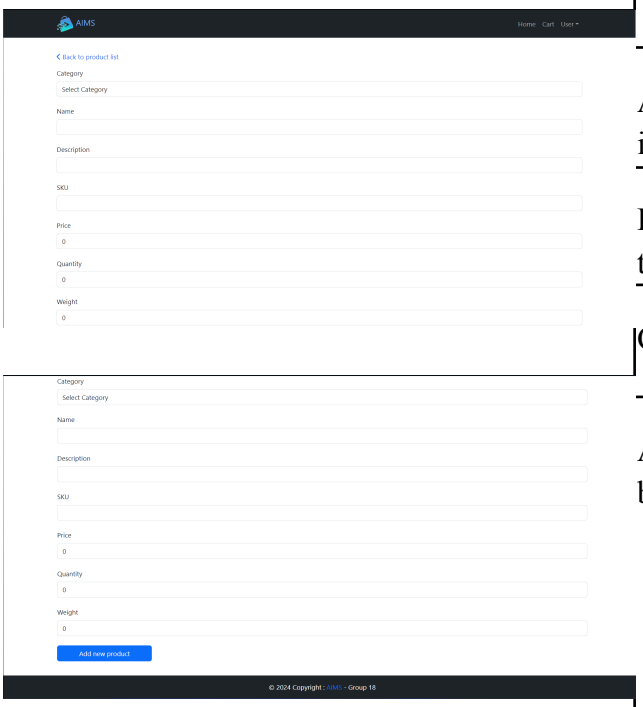
e, Manage product page

AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Manage product page	18/6/2024			Nguyen Duc Thang

	Control	Operation	Function
	Area for displaying search content	Initial	Display the search content
	Area for displaying items in the shop	Initial	Display the items in the shop
	Category dropdown	Click	Search for items which have category matching with option in dropdown
	Area for displaying item information	Initial	Display title, price, category of that item
	Create new product button	Click	Navigate to create product page
	Edit icon	Click	Navigate to edit product page
	Trash icon	Click	Delete product
	Pagination	Click	Display another pages of items in the shop

f, Create product page

AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Create product page	18/6/2024			Nguyen Duc Thang



Control	Operation	Function
Area for display item information	Initial	Display some questions to request customer to fill
Back to product list text	Click	Navigate to manage product page
Category dropdown	Click	Display questions for specific type of product
Add new product button	Click	Add new product to shop

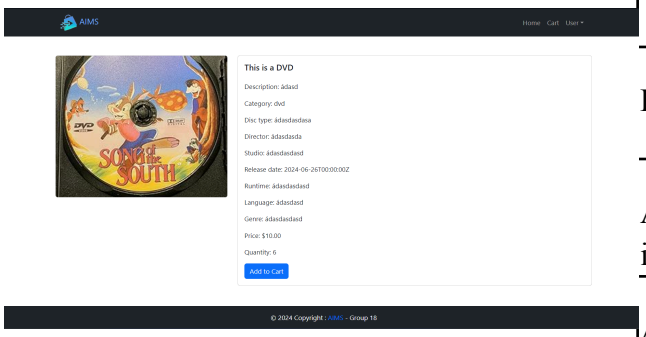
g, Edit product page

AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Home page	18/6/2024			Nguyen Duc Thang

Control	Operation	Function
Save button	Click	Save item information
Area for display item information	Initial	Display some questions to request customer to fill

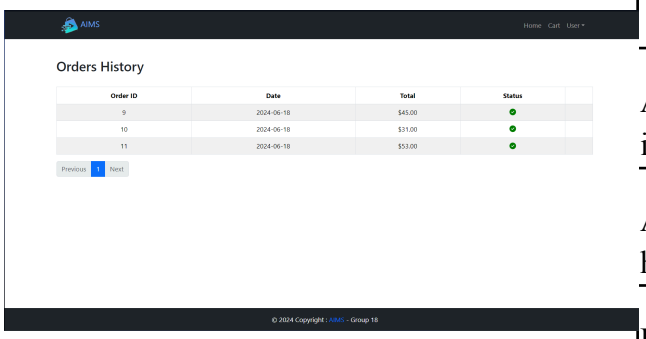
			

h, Product detail page

AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Product detail page	18/6/2024			Nguyen Duc Thang
		Control	Operation	Function	
		Product image	Initial	Display item image	
		Area for display item information	Initial	Display item information	
		Add to cart button	Click	Add item to cart	

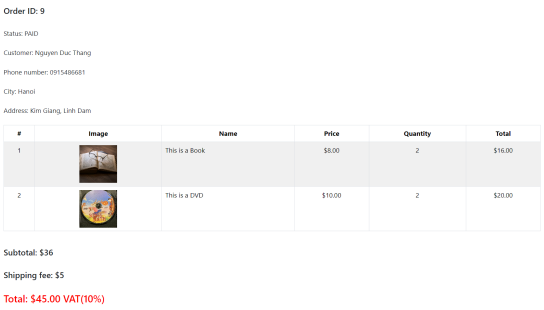
j, Order history page

AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Order history page	18/6/2024			Nguyen Duc Thang

	Control	Operation	Function
	Area for display order information	Initial	Display order information
	Area for display order history	Initial	Display list of orders
	Pagination	Click	Display another pages of items in the shop

k, Order detail page

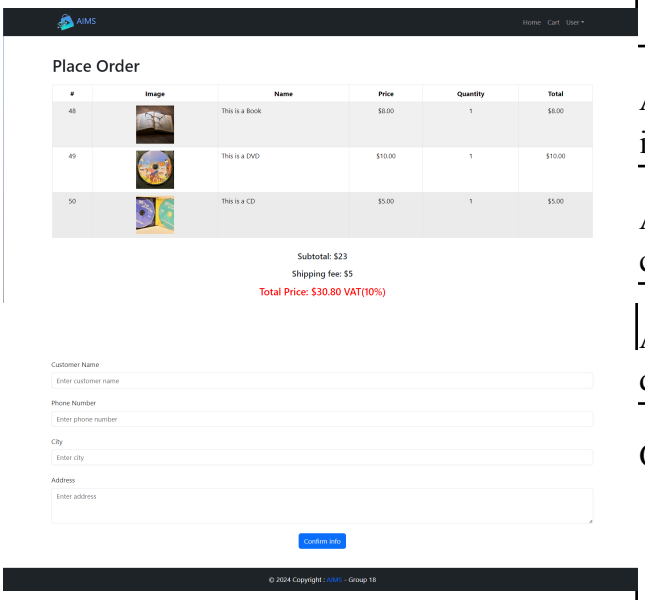
AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Order detail page	18/6/2024			Nguyen Duc Thang



Control	Operation	Function
Area for display order items	Initial	Display order items with corresponding information
Area for display all costs	Initial	Display order subtotal, shipping fee, total
Area for display order information	Initial	Display order information like customer, phone number, city, address

l, Place order page

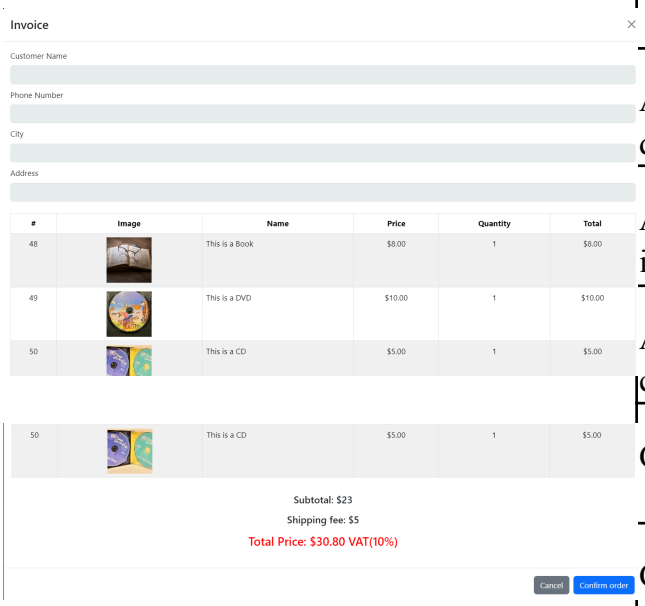
AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Place order page	18/6/2024			Nguyen Duc Thang



Control	Operation	Function
Area for display cart items	Initial	Display cart items with corresponding information
Area for display all costs	Initial	Display order subtotal, shipping fee, total
Area for display delivery information	Initial	Display some questions to request customer to fill
Confirm info button	Click	Open invoice modal which displays order information

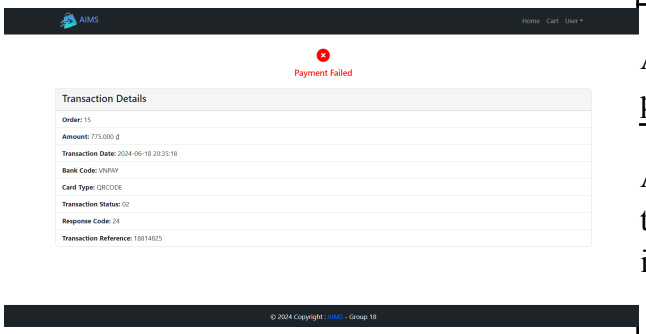
m, Invoice modal

AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Invoice modal	18/6/2024			Nguyen Duc Thang



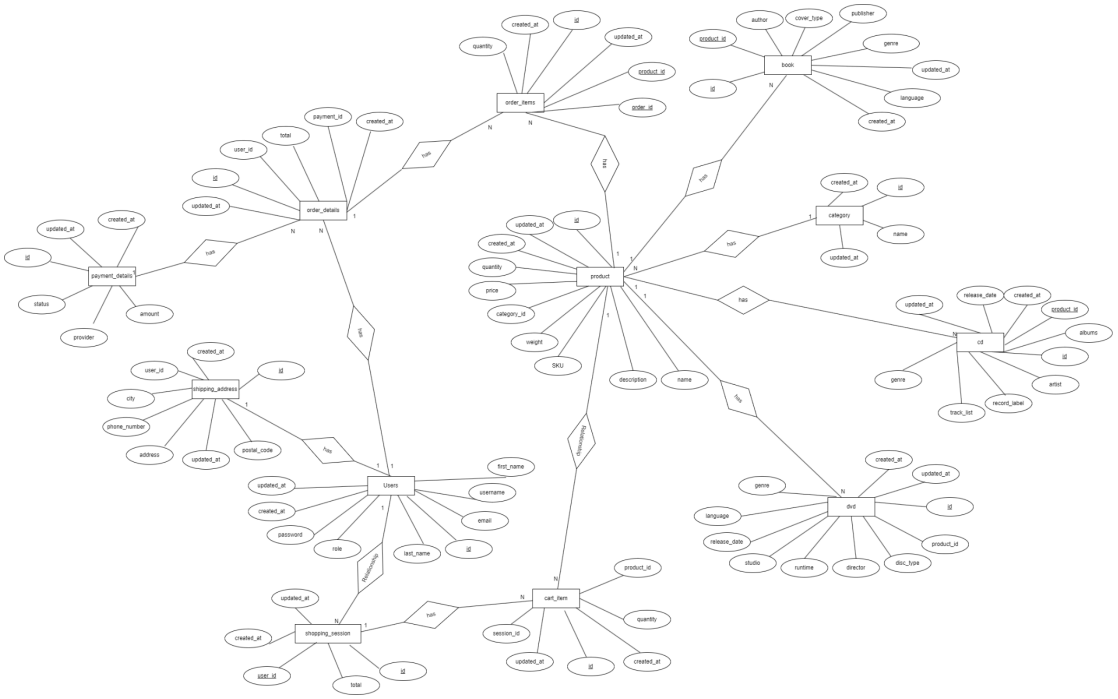
Control	Operation	Function
Area for display delivery information	Initial	Display customer delivery information
Area for display order items	Initial	Display orderitems with corresponding information
Area for display all costs	Initial	Display order subtotal, shipping fee, total
Confirm order button	Click	Place an order then navigate to order history page
Cancel button	Click	Close invoice modal

n, Notification page

AIMS Website		Date of creation	Approved by	Reviewed by	Person in charge
Screen specification	Notification page	18/6/2024			Nguyen Duc Thang
		Control	Operation	Function	
		Area for display payment status	Initial	Display payment status	
		Area for display transaction information	Initial	Display transaction information	

4.2 Data Modeling

4.2.1 Conceptual Data Modeling



a, Entities:

- **book:** Represents a book with various attributes like author, genre, and publication details. Each book is associated with a product.
- **cd:** Represents a CD with attributes like albums, record label, and release date. Each CD is associated with a product.

- **dvd:** Represents a DVD with attributes like disc type, genre, and studio. Each DVD is associated with a product.
- **product:** Represents a general product with attributes like description, price, and quantity. Products are categorized and can be associated with books, cds, and dvds.
- **category:** Represents a category of products with a name and timestamps for creation and updates.
- **cart_item:** Represents an item in a shopping cart, including the quantity and references to the product and shopping session it belongs to
- **shopping_session:** Represents a shopping session, tracking the user and total cost of items in the session.
- **users:** Represents a user with login credentials, contact information, and role.
- **order_items:** Represents items within an order, including quantity and references to the product and order details.
- **order_details:** Represents the details of an order, including total amount, customer information, and references to payment details and shipping address
- **payment_details:** Represents payment information, including amount, provider, and status.
- **shipping_address:** Represents a shipping address, including details like city, phone number, and postal code.

b, Relationships

book - product

- Relationship: One-to-One
- Description: Each book is associated with one product, and each product can be associated with one book.

cd - product

- Relationship: One-to-One
- Description: Each CD is associated with one product, and each product can be associated with one CD.

dvd - product

- Relationship: One-to-One
- Description: Each DVD is associated with one product, and each product can be associated with one DVD.

product - category

- Relationship: Many-to-One
- Description: Many products can belong to one category, but each product is assigned to only one category.

cart_item - product

- Relationship: Many-to-One

- Description: Many cart items can reference one product, but each cart item references only one product.

cart_item - shopping_session

- Relationship: Many-to-One
- Description: Many cart items can be part of one shopping session, but each cart item is associated with one shopping session.

shopping_session - users

- Relationship: Many-to-One
- Description: Many shopping sessions can be initiated by one user, but each shopping session is linked to one user.

order_items - product

- Relationship: Many-to-One
- Description: Many order items can reference one product, but each order item references only one product.

order_items - order_details

- Relationship: Many-to-One
- Description: Many order items can belong to one order, but each order item is associated with one order.

order_details - payment_details

- Relationship: Many-to-One
- Description: Many order details can reference one payment, but each order detail references only one payment.

order_details - shipping_address

- Relationship: Many-to-One
- Description: Many order details can reference one shipping address, but each order detail references only one address.

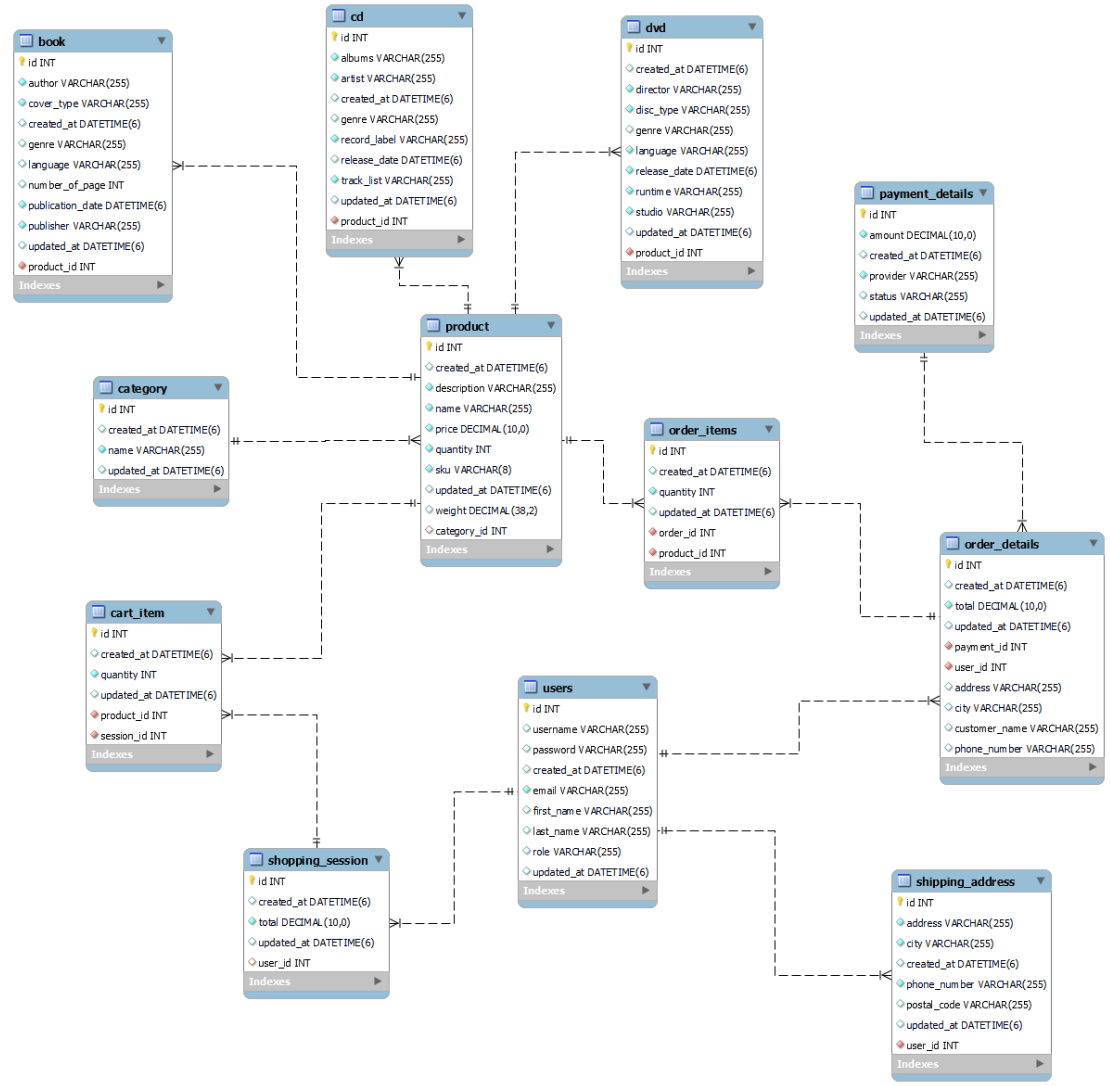
4.2.2 Database Design

4.2.2.1 Database Management System

The choice of a Database Management System (DBMS) is critical to the successful implementation and performance of a database. It depends on several factors, including the nature of the application, expected workload, scalability requirements, budget constraints, and the team's expertise. For implementing AIMS project, a

relational DBMS (RDBMS) would be the most appropriate choice due to the structured nature of the data and the clear relationships between entities.

4.2.2.2 Database Diagram



4.2.2.3 Database Detail Design

Table 1. Table users

#	PK	FK	Column Name	Data Type	Default Value	Mandatory	Description
1	x		id	integer		Yes	Primary key, auto-incrementing identifier

2			username	varchar		Yes	Unique username for the user
3			first_name	varchar			User's first name
4			last_name	varchar			User's last name
5			role	varchar	"USER"		Role of the user, default is "USER"
6			email	varchar		Yes	Unique email address of the user
7			password	varchar		Yes	User's password
8			created_at	timestamp			Timestamp when the user was created
9			updated_at	timestamp			Timestamp when the user was last updated

Table 2. Table products

#	PK	FK	Column Name	Data Type	Default Value	Mandatory	Description
1	x		id	integer		Yes	Primary key, auto-incrementing identifier
2			name	varchar		Yes	Name of the product
3			description	varchar		Yes	Description of the product
4			SKU	varchar(8)		Yes	Stock Keeping Unit, 8 characters
5			category_id	int			Identifier for the product category

6			weight	decimal		Yes	Weight of the product
7			price	decimal		Yes	Price of the product
8			quantity	int		Yes	Quantity of the product in stock
9			created_at	timestamp			Timestamp when the product was created

Table 3. Table dvd

#	PK	FK	Column Name	Data Type	Default Value	Mandatory	Description
1	x		id	integer	Auto Increment	Yes	Primary key
2		x	product_id	integer		Yes	Foreign key referencing product.id
3			disc_type	varchar		Yes	Type of disc (e.g., DVD, Blu-ray)
4			director	varchar		Yes	Director of the DVD content
5			runtime	varchar		Yes	Duration of the DVD content
6			studio	varchar		Yes	Production studio
7			language	varchar		Yes	Language of the content
8			release_date	timestamp		Yes	Release date of the DVD
9			genre	varchar		No	Genre of the DVD content

Table 4. Table category

#	PK	FK	Column Name	Data Type	Default Value	Mandatory	Description
1	x		id	integer	Auto Increment	Yes	Primary key
2			name	varchar		Yes	Name of the category
3			created_at	timestamp		No	Record creation timestamp
4			updated_at	timestamp		No	Record update timestamp

Table 5. Table order_details

#	PK	FK	Column Name	Data Type	Default Value	Mandatory	Description
1	x		id	integer	Auto Increment	Yes	Primary key
2			user_id	integer		Yes	ID of the user placing the order
3			total	decimal		Yes	Total amount of the order
4		x	payment_id	integer		Yes	Foreign key referencing payment_details.id
5			created_at	timestamp		No	Record creation timestamp
6			updated_at	timestamp		No	Record update timestamp

Table 6. Table shopping_session

#	PK	FK	Column name	Data type	Default value	Mandatory	Description

1	x		id	INTEGER		x	Primary key
2			user_id	INTEGER			Foreign key to user table
3			total	DECIMAL			Total amount for the session
4			created_at	TIMESTAMP			Timestamp of session creation
5			updated_at	TIMESTAMP			Timestamp of last update

Table 7. Table cart_item

#	PK	FK	Column name	Data type	Default value	Mandatory	Description
1	x		id	INTEGER		x	Primary key
2		x	session_id	INTEGER		x	Foreign key to shopping_session
3		x	product_id	INTEGER		x	Foreign key to product table
4			quantity	INTEGER			Quantity of products in cart

5			created_at	TIMESTAMP			Timestamp of item addition
6			updated_at	TIMESTAMP			Timestamp of last update

Table 8. Table book

#	PK	FK	Column name	Data type	Default value	Mandatory	Description
1	x		id	INTEGER		x	Primary key
2		x	product_id	INTEGER		x	Foreign key to product table
3			author	VARCHAR		x	Author(s) of the book
4			cover_type	VARCHAR		x	Type of cover (hardcover, paperback, etc.)
5			publisher	VARCHAR		x	Publisher of the book
6			publication_date	TIMESTAMP			Date when the book was published

7			number_of_pages	INTEGER			Number of pages in the book
8			language	VARCHAR			Language of the book
9			genre	VARCHAR			Genre of the book
10			created_at	TIMESTAMP			Timestamp of book creation
11			updated_at	TIMESTAMP			Timestamp of last update

Table 9. Table cd

#	PK	FK	Column name	Data type	Default value	Mandatory	Description
1	x		id	INTEGER		x	Primary key
2		x	product_id	INTEGER		x	Foreign key to product table
3			albums	VARCHAR			Name of the album or CD
4			artist	VARCHAR		x	Artist or band name

5			record_label	VARCHAR		x	Record label of the CD
6			track_list	VARCHAR			List of tracks on the CD
7			genre	VARCHAR			Genre of the CD
8			release_date	TIMESTAMP			Release date of the CD
9			created_at	TIMESTAMP			Timestamp of CD creation
10			updated_at	TIMESTAMP			Timestamp of last update

Table 10. Table order_items

#	PK	FK	Column name	Data type	Default value	Mandatory	Description
1	x		id	INTEGER		x	Primary key
2		x	order_id	INTEGER		x	Foreign key to orders table
3		x	product_id	INTEGER		x	Foreign key to product table

4			quantity	INTEGER			Quantity of products in order
5			created_at	TIMESTAMP			Timestamp of item addition to order
6			updated_at	TIMESTAMP			Timestamp of last update

Table 11. Table payment_details

#	PK	FK	Column name	Data type	Default value	Mandatory	Description
1	x		id	INTEGER		x	Primary key
2			amount	DECIMAL			Amount of payment
3			provider	VARCHAR			Payment provider
4			status	VARCHAR			Payment status
5			created_at	TIMESTAMP			Timestamp of payment creation
6			updated_at	TIMESTAMP			Timestamp of last update

Table 12: Table shipping_address

#	PK	FK	Column name	Data type	Default value	Mandatory	Description
1	x		id	INTEGER		x	Primary key
2			address	DECIMAL			Address of user
3			city	VARCHAR			City
4			phone_number	VARCHAR			Phone number of user
5			postal_code	VARCHAR			
6			created_at	TIMESTAMP			Timestamp of shipping address creation
7			updated_at	TIMESTAMP			Timestamp of last update
8			user_id	INTEGER			User ID

4.3 Non-Database Management System Files

Description of Files:

4.3.1 application.properties

Purpose: To configure the settings of the Spring Boot application.

Type of File: Input file.

Modules that Read/Write the File: Read by the Spring Boot application upon startup.

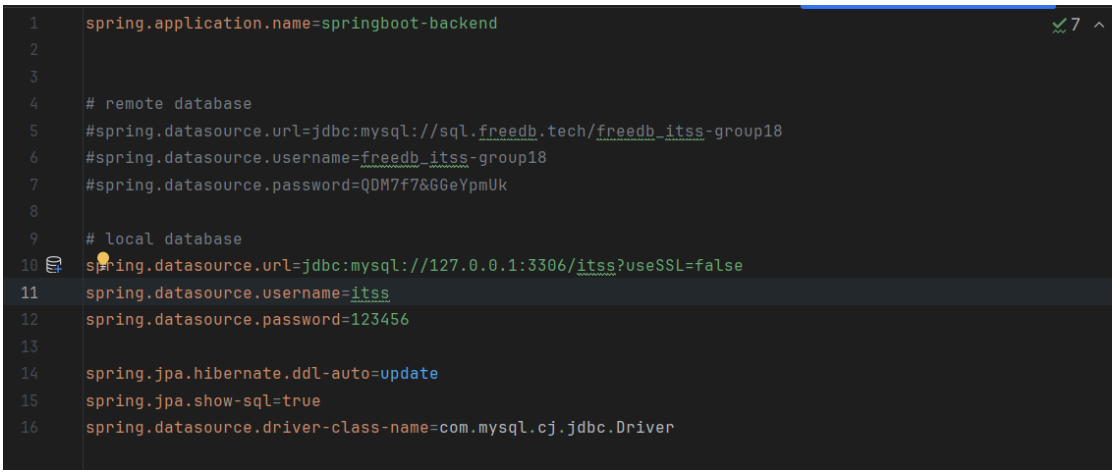
Access Method: Random access

Estimated File Size: 1KB

Update Frequency: Occasionally, when there are configuration changes

Backup and Recovery Specifications: Weekly backups, retaining backups for 4 weeks

File Structure:



4.3.2 tsconfig.json

Purpose: To configure the TypeScript compiler options for the React application.

Type of File: Input file

Modules that Read/Write the File: Read by the TypeScript compiler when starting the React application

File Structure

```
{
  "compilerOptions": {
    "target": "ES2020",
    "useDefineForClassFields": true,
    "lib": ["ES2020", "DOM", "DOM.Iterable"],
    "module": "ESNext",
    "skipLibCheck": true,

    /* Bundler mode */
    "moduleResolution": "bundler",
    "allowImportingTsExtensions": true,
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx",

    /* Linting */
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noFallthroughCasesInSwitch": true
  },
  "include": ["src"],
  "references": [{ "path": "./tsconfig.node.json" }]
}
```

Access Method: Random access.

Estimated File Size: 2KB.

Update Frequency: Occasionally, when there are changes in compilation settings.

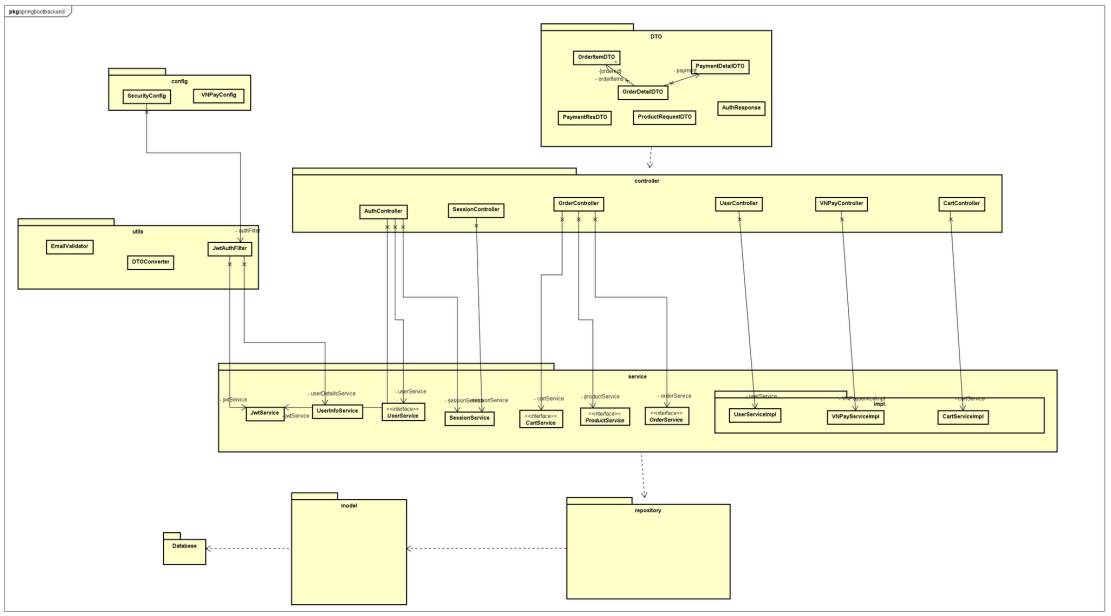
Backup and Recovery Specifications: Weekly backups, retaining backups for 4 weeks.

- **File application.properties**: Used to configure the settings of the Spring Boot application, including database connection information and Hibernate settings.
- **File tsconfig.json**: Used to configure TypeScript compiler options for the React application, including target compilation settings, libraries used, and linting rules.

These details ensure that all non-DBMS files are managed and utilized effectively, ensuring data integrity and recoverability when necessary.

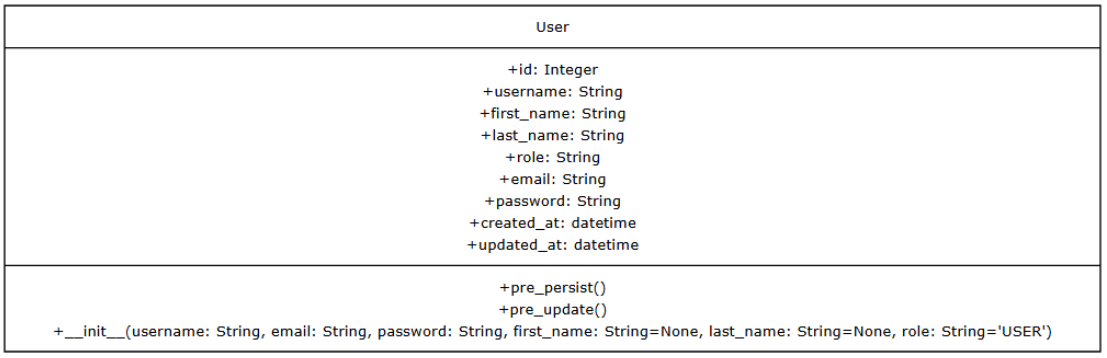
4.4 Class Design

4.4.1 General Class Diagram

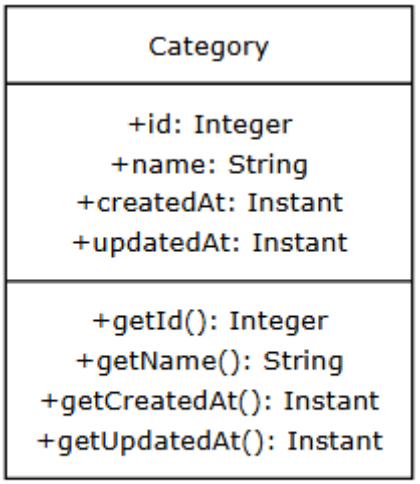


4.4.2 Class Diagrams

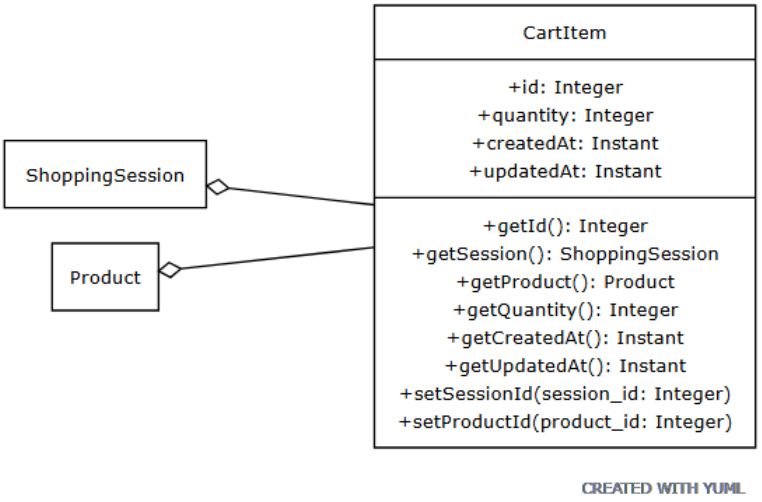
4.4.2.1 User Class Diagram



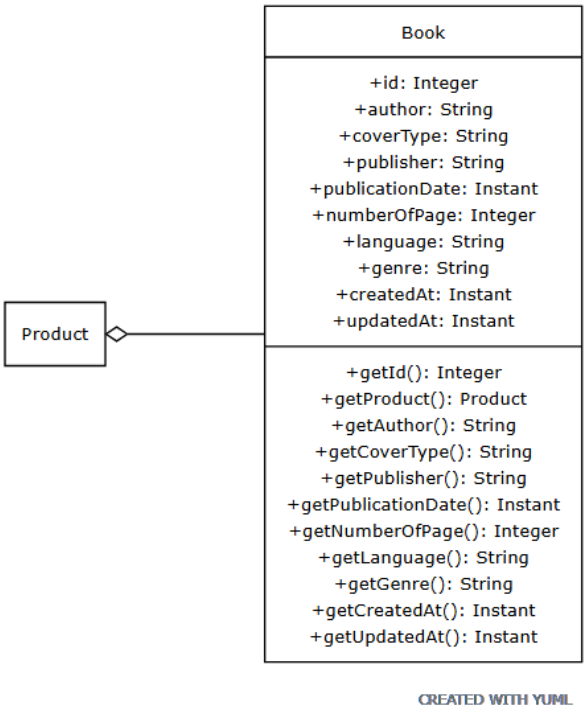
4.4.2.2 Category Diagram



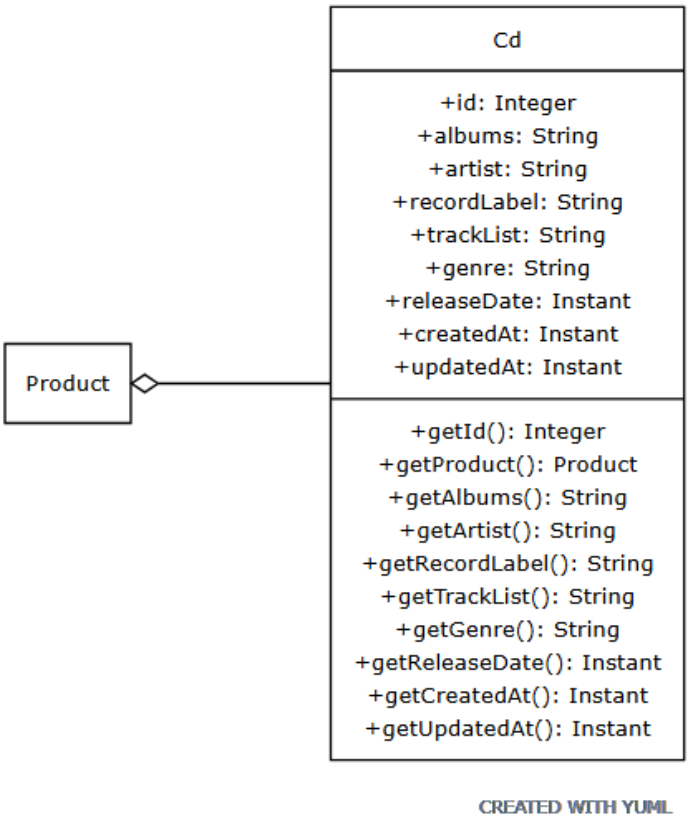
4.4.2.3 Cart Item Diagram



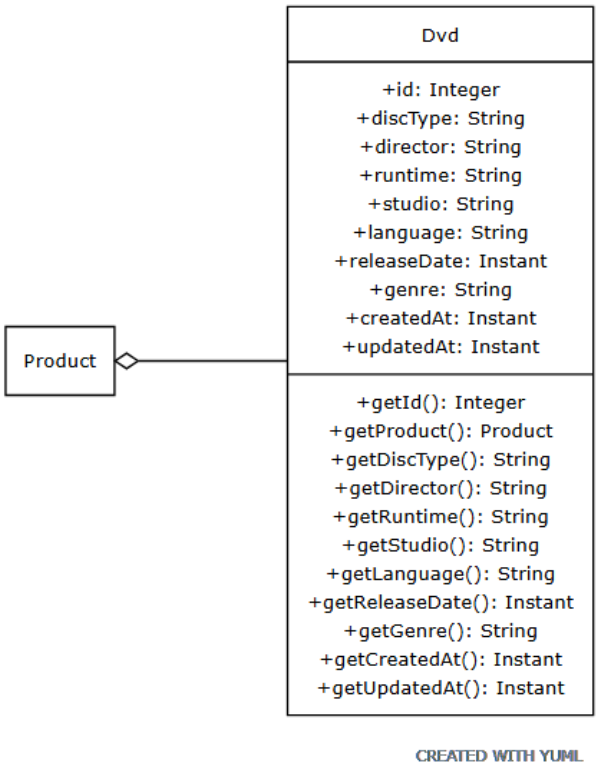
4.4.2.4 Book Diagram



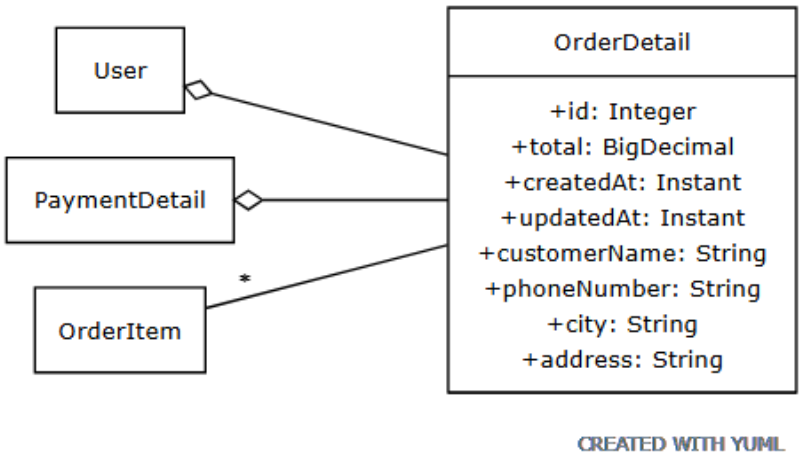
4.4.2.5 cd Diagram



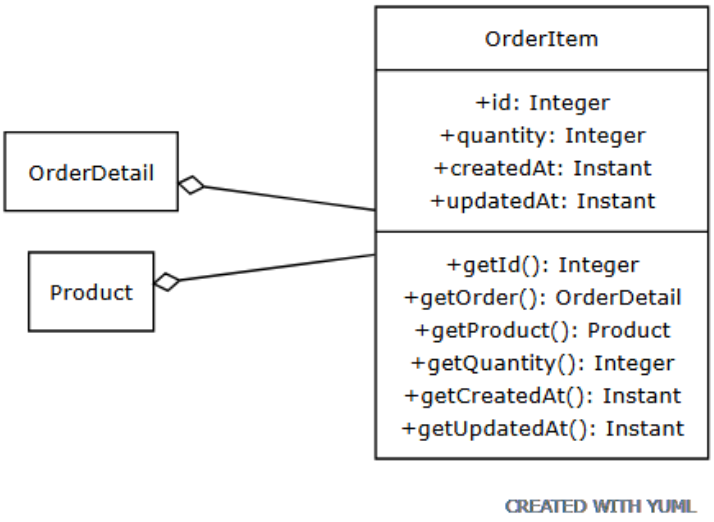
4.4.2.6 Dvd Diagram



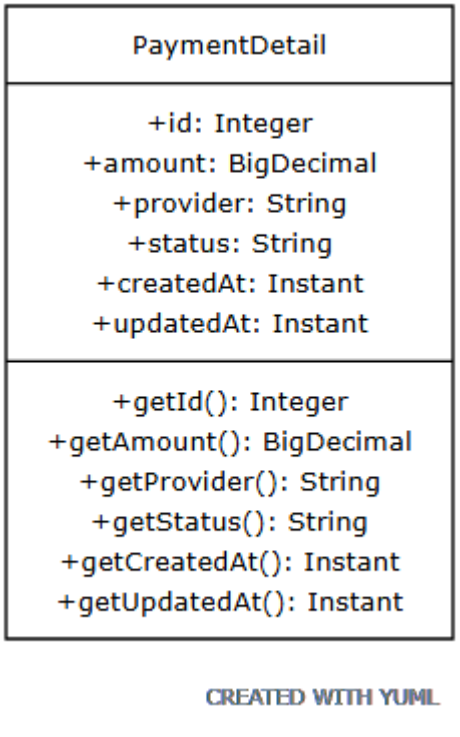
4.4.2.7 Order Detail Diagram



4.4.2.8 Order Item Diagram

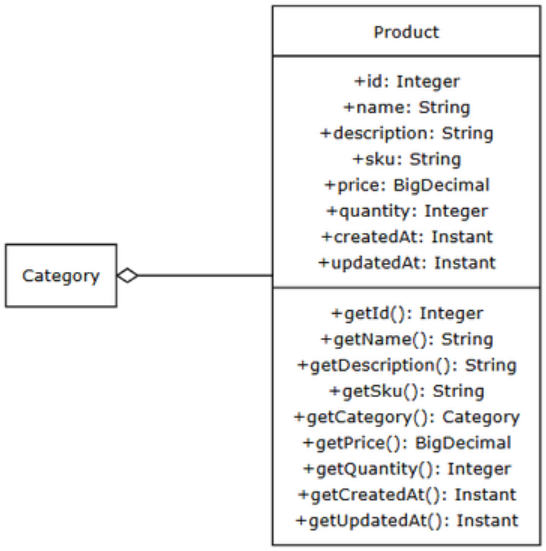


4.4.2.9 Payment Detail Diagram

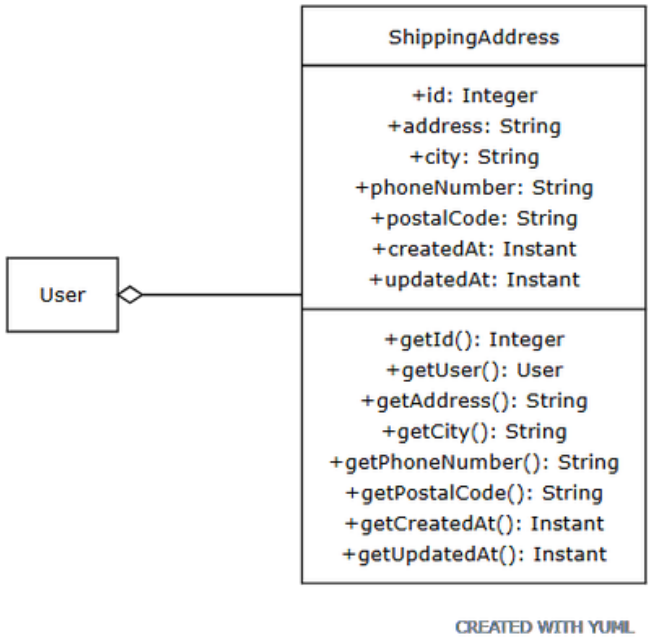


]

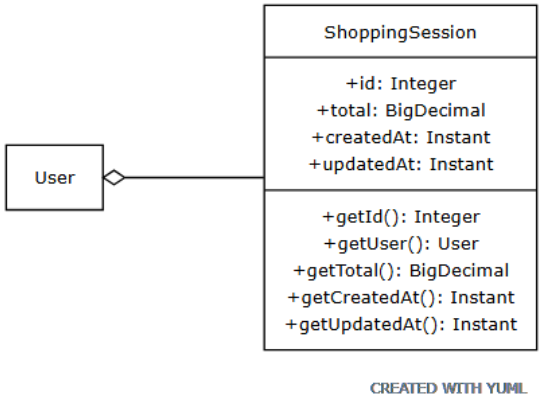
4.4.2.10 Product Diagram



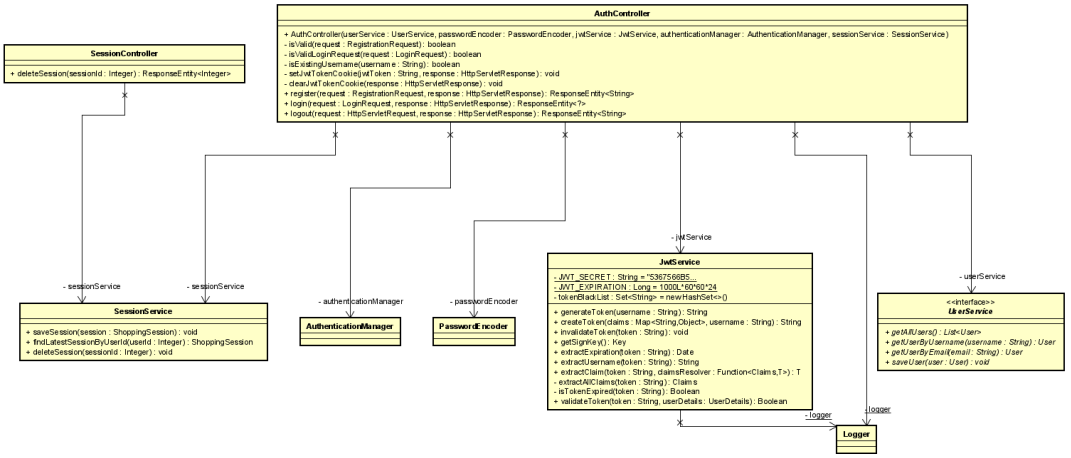
4.4.2.11 Shipping Address Diagram



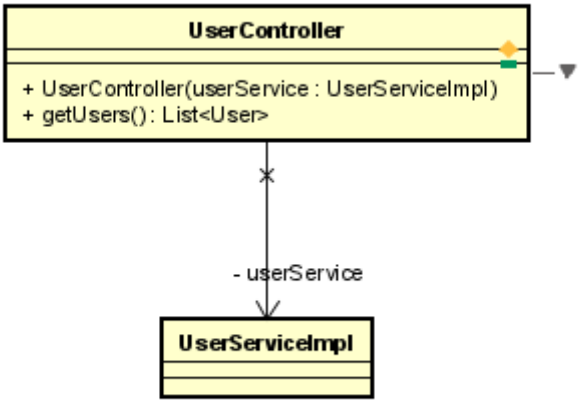
4.4.2.12 Shopping Session Diagram



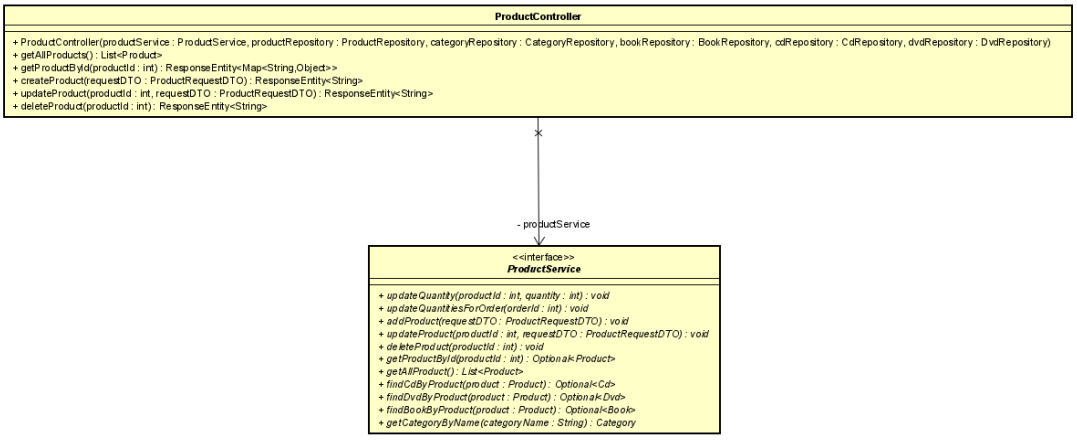
4.4.2.13 Auth Controller and Session Controller Diagram



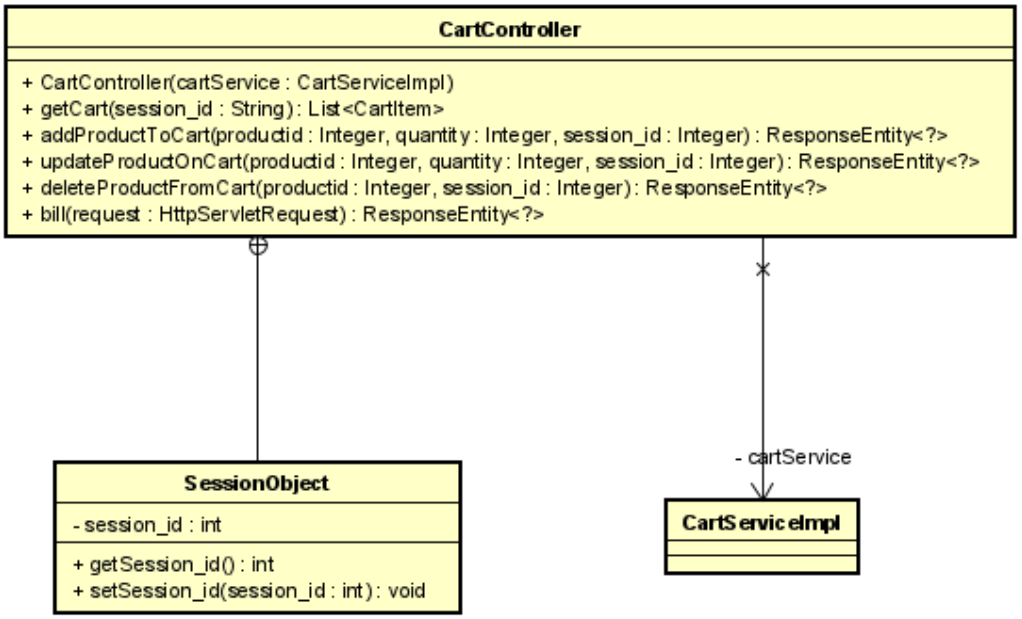
4.4.2.14 User Controller Diagram



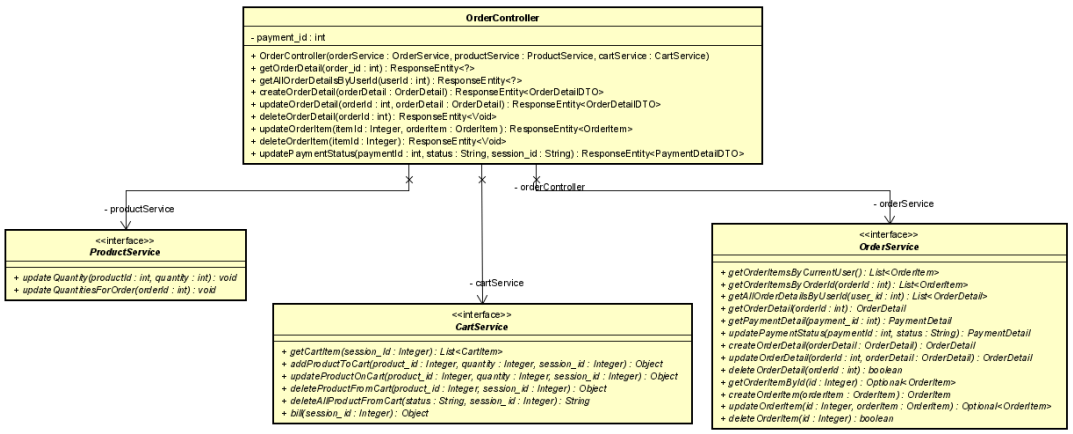
4.4.2.15 Product Controller Diagram



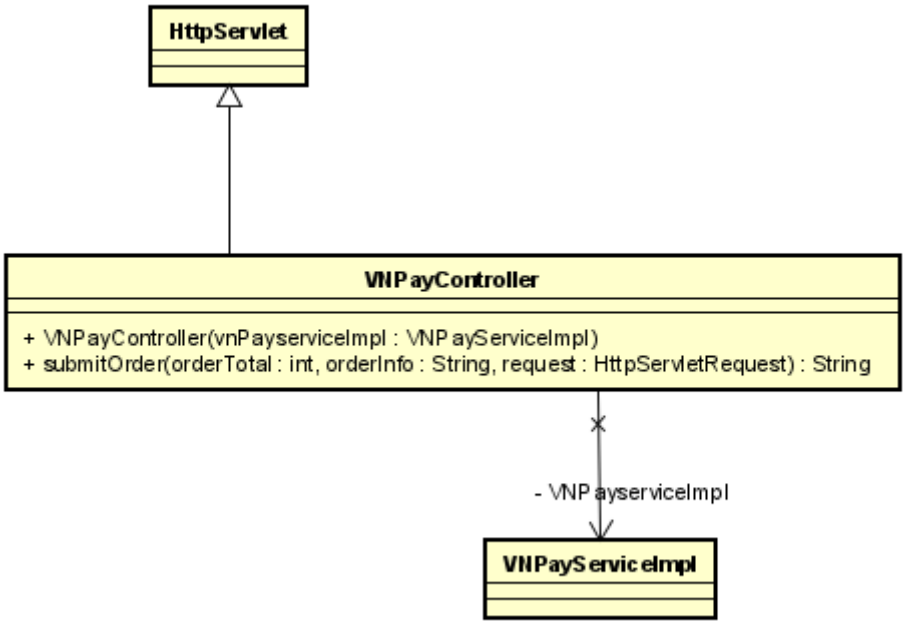
4.4.2.16 Cart Controller Diagram



4.4.2.17 Order Controller Diagram



4.4.2.18 VNPay Controller



4.4.3 Class Design

4.4.3.1 Models

a, Product Class

The Product class is the base class for all products sold in shop. It has the following attributes:

description: A string that describes the product.
name: A string that represents the name of the product.
price: A decimal that represents the price of the product.
quantity: An integer that represents the quantity of the product in stock.
sku: A string that represents the stock keeping unit of the product.
weight: A decimal that represents the weight of the product.
category_id: An integer that represents the ID of the product's category.
The Product class has the following methods:
__init__: The constructor method that initializes the product's attributes.
get_description: A method that returns the product's description.
get_name: A method that returns the product's name.
get_price: A method that returns the product's price.
get_quantity: A method that returns the product's quantity.
get_sku: A method that returns the product's SKU.
get_weight: A method that returns the product's weight.
get_category_id: A method that returns the product's category ID.

b, Book Class

The Book class is a subclass of the Product class that represents a book product. It has the following attributes:

author: A string that represents the author of the book.
cover_type: A string that represents the cover type of the book.
genre: A string that represents the genre of the book.

language: A string that represents the language of the book.
number_of_page: An integer that represents the number of pages in the book.
publication_date: A datetime that represents the publication date of the book.
publisher: A string that represents the publisher of the book.
product_id: An integer that represents the ID of the book product.
The Book class has the following methods:

__init__: The constructor method that initializes the book's attributes.
get_author: A method that returns the book's author.
get_cover_type: A method that returns the book's cover type.
get_genre: A method that returns the book's genre.
get_language: A method that returns the book's language.
get_number_of_page: A method that returns the book's number of pages.
get_publication_date: A method that returns the book's publication date.
get_publisher: A method that returns the book's publisher.
get_product_id: A method that returns the book's product ID.

c, CD Class
The CD class is a subclass of the Product class that represents a CD product. It has the following attributes:

albums: A string that represents the albums in the CD.
artist: A string that represents the artist of the CD.
genre: A string that represents the genre of the CD.
record_label: A string that represents the record label of the CD.
release_date: A datetime that represents the release date of the CD.
track_list: A string that represents the track list of the CD.
product_id: An integer that represents the ID of the CD product.
The CD class has the following methods:

__init__: The constructor method that initializes the CD's attributes.
get_albums: A method that returns the CD's albums.
get_artist: A method that returns the CD's artist.
get_genre: A method that returns the CD's genre.
get_record_label: A method that returns the CD's record label.
get_release_date: A method that returns the CD's release date.
get_track_list: A method that returns the CD's track list.
get_product_id: A method that returns the CD's product ID.

d, DVD Class
The DVD class is a subclass of the Product class that represents a DVD product. It has the following attributes:

director: A string that represents the director of the DVD.
disc_type: A string that represents the disc type of the DVD.
genre: A string that represents the genre of the DVD.
language: A string that represents the language of the DVD.
release_date: A datetime that represents the release date of the DVD.

runtime: A decimal that represents the runtime of the DVD.
studio: A string that represents the studio of the DVD.
product_id: An integer that represents the ID of the DVD product.
The DVD class has the following methods:

__init__: The constructor method that initializes the DVD's attributes.
get_director: A method that returns the DVD's director.
get_disc_type: A method that returns the DVD's disc type.
get_genre: A method that returns the DVD's genre.
get_language: A method that returns the DVD's language.
get_release_date: A method that returns the DVD's release date.
get_runtime: A method that returns the DVD's runtime.
get_studio: A method that returns the DVD's studio.
get_product_id: A method that returns the DVD's product ID.

e, Category Class
The Category class represents a product category. It has the following attributes:

name: A string that represents the name of the category.
product_id: An integer that represents the ID of the product in the category.
The Category class has the following methods:

__init__: The constructor method that initializes the category's attributes.
get_name: A method that returns the category's name.
get_product_id: A method that returns the category's product ID.

f, CartItem Class
The CartItem class represents an item in a shopping cart. It has the following attributes:

quantity: An integer that represents the quantity of the item in the cart.
product_id: An integer that represents the ID of the product in the cart.
session_id: An integer that represents the ID of the shopping session.
The CartItem class has the following methods:

__init__: The constructor method that initializes the cart item's attributes.
get_quantity: A method that returns the cart item's quantity.
get_product_id: A method that returns the cart item's product ID.
get_session_id: A method that returns the cart item's session ID.

e, PaymentDetail Class
The PaymentDetail class represents a payment detail. It has the following attributes:

amount: A decimal that represents the amount of the payment.
provider: A string that represents the payment provider.
status: A string that represents the status of the payment.
The PaymentDetail class has the following methods:

__init__: The constructor method that initializes the payment detail's attributes.

get_amount: A method that returns the payment detail's amount.
get_provider: A method that returns the payment detail's provider.
get_status: A method that returns the payment detail's status.

f, OrderItem Class
The OrderItem class represents an item in an order. It has the following attributes:

quantity: An integer that represents the quantity of the item in the order.
order_id: An integer that represents the ID of the order.
product_id: An integer that represents the ID of the product in the order.
The OrderItem class has the following methods:

__init__: The constructor method that initializes the order item's attributes.
get_quantity: A method that returns the order item's quantity.
get_order_id: A method that returns the order item's order ID.
get_product_id: A method that returns the order item's product ID.

g, ShoppingSession Class
The ShoppingSession class represents a shopping session. It has the following attributes:

total: A decimal that represents the total amount of the shopping session.
user_id: An integer that represents the ID of the user who initiated the shopping session.
The ShoppingSession class has the following methods:

__init__: The constructor method that initializes the shopping session's attributes.
get_total: A method that returns the shopping session's total.
get_user_id: A methodthat returns the shopping session's user ID.

h, User Class
The User class represents a user. It has the following attributes:

username: A string that represents the username of the user.
password: A string that represents the password of the user.
email: A string that represents the email of the user.
first_name: A string that represents the first name of the user.
last_name: A string that represents the last name of the user.
role: A string that represents the role of the user.
The User class has the following methods:

__init__: The constructor method that initializes the user's attributes.
get_username: A method that returns the user's username.
get_password: A method that returns the user's password.
get_email: A method that returns the user's email.
get_first_name: A method that returns the user's first name.
get_last_name: A method that returns the user's last name.
get_role: A method that returns the user's role.

j, OrderDetail Class

The OrderDetail class represents an order detail. It has the following attributes:

total: A decimal that represents the total amount of the order.
payment_id: An integer that represents the ID of the payment.
user_id: An integer that represents the ID of the user who initiated the order.
address: A string that represents the address of the order.
city: A string that represents the city of the order.
customer_name: A string that represents the customer name of the order.
phone_number: A string that represents the phone number of the order.
The OrderDetail class has the following methods:

__init__: The constructor method that initializes the order detail's attributes.
get_total: A method that returns the order detail's total.
get_payment_id: A method that returns the order detail's payment ID.
get_user_id: A method that returns the order detail's user ID.
get_address: A method that returns the order detail's address.
get_city: A method that returns the order detail's city.
get_customer_name: A method that returns the order detail's customer name.
get_phone_number: A method that returns the order detail's phone number.

k, ShippingAddress Class
The ShippingAddress class represents a shipping address. It has the following attributes:

address: A string that represents the address of the shipping address.
city: A string that represents the city of the shipping address.
phone_number: A string that represents the phone number of the shipping address.
postal_code: A string that represents the postal code of the shipping address.
user_id: An integer that represents the ID of the user who initiated the shipping address.
The ShippingAddress class has the following methods:

__init__: The constructor method that initializes the shipping address's attributes.
get_address: A method that returns the shipping address's address.
get_city: A method that returns the shipping address's city.
get_phone_number: A method that returns the shipping address's phone number.
get_postal_code: A method that returns the shipping address's postal code.
get_user_id: A method that returns the shipping address's user ID.

4.4.3.2 Controllers
a, Order Controller
This class serves as the primary interface for interacting with the order system. It's responsible for handling requests related to order details, items, and payment status updates. It relies on three other classes: OrderService, ProductService, and CartService.

- ProductService: This interface defines methods for managing product quantities, both individually and for specific orders. It acts as a data source for product information.
- CartService: This interface defines methods for managing shopping cart items. It allows users to add, update, and remove products from their carts. It also handles generating billing information.

- OrderService: This interface provides methods for retrieving, updating, and deleting order-related data. It includes methods to fetch orders by user, order ID, and payment ID.

b, Product Controller

ProductController: Acts as the central point for handling product-related requests. It receives requests from the user interface or other clients and delegates them to appropriate repositories or services.

Responsibilities:

- Retrieves a list of all products.
- Creates a new product.
- Retrieves a product based on its ID.
- Deletes a product.
- Updates a product.

Dependencies:

- ProductService: A service responsible for handling product-specific operations.Updates the quantity of a product. Updates quantities of products for a given order.

c, Cart Controller

Responsibilities: Acts as the controller for all shopping cart related operations. Handles user requests, interacts with the CartService to perform operations, and returns appropriate responses.

Methods:

- CartController(cartService: CartServiceImpl): Constructor initializing the controller with a reference to the CartServiceImpl instance.
- getCart(session_id: String): List<Cartitem>: Retrieves the cart associated with a given session.
- addProductToCart(productid: Integer, quantity: Integer, session_id: Integer): ResponseEntity<?>: Adds a product to a user's cart.
- updateProductOnCart(productid: Integer, quantity: Integer, session_id: Integer): ResponseEntity<?>: Updates the quantity of a product in a user's cart.
- deleteProductFromCart(productid: Integer, session_id: Integer): ResponseEntity<?>: Removes a product from a user's cart.
- bill(request: HttpServletRequest): ResponseEntity<?>: Processes the user's cart for checkout, generating the final bill.

Relationships:

- Uses: CartServiceImpl – The controller utilizes the services provided by the CartServiceImpl to perform cart operations.
- Uses: SessionObject – The controller likely requires access to the user's session **information.**

CartServiceImpl:

- Responsibilities: Implements the actual logic for managing shopping carts. Handles interactions with the underlying data storage (database, cache, etc.) for cart information.
- Methods: Implementation details for the methods are not depicted in this UML, but they would correspond to the methods defined in the CartController class.

- Relationships:
- Used By: CartController – The CartServiceImpl is used by the controller to carry out cart operations.

SessionObject:

- Responsibilities: Manages the user's session information, including the session ID.
- Methods:
- getSession_id(): int: Returns the session ID.
- setSession_id(session_id: int): void: Sets the session ID.
- Relationships:
- Used By: CartController – The controller potentially accesses session information through this object.

d, User Controller

Responsibilities:

- Manages user-related operations.
- Retrieves a list of users from the UserServiceImpl.

Methods:

- UserController(userService: UserServiceImpl): Constructor that initializes the UserController with a dependency on UserServiceImpl.
- getUsers(): List<User>: Retrieves a list of all users from the UserServiceImpl.

Dependencies:

- UserServiceImpl: The service class responsible for user-related operations.

e, Auth Controller

Responsibilities:

- Manages user authentication and registration.
- Validates registration and login requests.
- Encodes user passwords.
- Saves new users to the database.
- Generates JWT tokens for authenticated users.

Methods:

- AuthController(userService: UserService, passwordEncoder: PasswordEncoder, jwtService: JwtService, authenticationManager: AuthenticationManager, sessionService: SessionService): Constructor that initializes the AuthController with dependencies on UserService, PasswordEncoder, JwtService, AuthenticationManager, and SessionService.
- register(registrationRequest: RegistrationRequest): ResponseEntity<String>: Validates and registers a new user.
- login(loginRequest: LoginRequest): ResponseEntity<?>: Authenticates a user and generates a JWT token.

Dependencies:

- UserService: The service class responsible for user-related operations.
- PasswordEncoder: Encodes user passwords.
- JwtService: Generates JWT tokens.
- AuthenticationManager: Authenticates users.
- SessionService: Manages user sessions.

f, Session Controller

Responsibilities:

- Manages user sessions.
- Deletes user sessions.
- Retrieves session information.

Methods:

- SessionController(sessionService: SessionService): Constructor that initializes the SessionController with a dependency on SessionService.
- deleteSession(sessionId: String): ResponseEntity<Void>: Deletes a user session.

Dependencies:

- SessionService: The service class responsible for session-related operations.

g, VNPay Controller

Responsibilities:

- Manages VNPay-related operations.
- Submits an order to VNPay.

Methods:

- VNPayController(vnPayService: VNPayServiceImpl): Constructor that initializes the VNPayController with a dependency on VNPayServiceImpl.
- submitOrder(orderTotal: int, orderInfo: String, request: HttpServletRequest): String: Submits an order to VNPay using the VNPayServiceImpl.

Dependencies:

- VNPayServiceImpl: The service class responsible for VNPay-related operations

5 Design Considerations

5.1 Goals and Guidelines

5.1.1. Goals.

User Experience (UX)

- Ease of Use: The interface should be intuitive and easy to navigate for customers of all ages and technical backgrounds. This includes clear categorization of products (CDs, DVDs, books) and a straightforward checkout process.
- Consistency: The system should have a consistent look and feel across all pages and functions, ensuring a seamless experience.
- Responsiveness: The application must be fully functional on various devices, including desktops, tablets, and smartphones.

Performance

- Speed: The system should prioritize fast loading times for all pages, ensuring a smooth shopping experience. This includes efficient search and filter functionalities.
- Scalability: The system should be capable of handling high traffic volumes, particularly during sales or promotions, without degradation in performance.

Scalability:

- Data Protection: Customer data, including personal information and payment details, must be securely handled and stored in compliance with relevant regulations (e.g., GDPR).
- Transaction Security: Secure payment gateways and encryption should be used to protect transactions.

Product Management

- Ease of Management: Managers should have a user-friendly interface to add, update, and remove products. This includes bulk upload functionalities and real-time inventory updates.
- Reporting: The system should provide detailed sales and inventory reports, helping managers make informed decisions.

5.1.2. Coding guidelines & conventions.

Code Quality

- Readability: Code should be easy to read and understand. Use meaningful variable and function names, and maintain consistent indentation and formatting.

- Documentation: Each module and function should be well-documented, explaining its purpose, parameters, and return values.
- Modularity: Code should be organized into reusable modules. This makes maintenance easier and promotes the use of clean, encapsulated components.

Version Control

- Git Usage: All code should be version-controlled using Git. Feature branches should be used for new features, and pull requests should be reviewed before merging.
- Commit Messages: Commit messages should be descriptive and follow the convention of starting with a verb in the imperative mood (e.g., "Add product search functionality").

Testing

- Automated Testing: Implement automated tests for critical parts of the system, including unit tests, integration tests, and end-to-end tests.
- Continuous Integration: Use a CI/CD pipeline to run tests on every commit, ensuring that new changes do not break existing functionality.

5.1.3. Design policies & tactics.

Technology Stack

- Frontend: Use React.js for building the user interface, leveraging its component-based architecture for modularity and reusability.
- Backend: Use Spring Boot for the server-side application, ensuring scalability, performance, and ease of development.
- Database: Utilize MySQL for its robustness and advanced features, ensuring reliable and efficient data management.

API Design

- RESTful Principles: Design APIs following RESTful principles, ensuring clear and consistent URL endpoints and HTTP methods.
- Error Handling: Implement comprehensive error handling, returning meaningful error messages and appropriate HTTP status codes.

User Interface Design

- Accessibility: Ensure the interface meets accessibility standards (e.g., WCAG 2.1), making it usable for various people.
- Visual Design: Follow a consistent color scheme and typography. Use whitespace effectively to avoid clutter and improve readability.

Product Management Interface

- Real-time Updates: Implement real-time inventory updates to reflect the current stock levels accurately.

- Role-based Access: Restrict certain functionalities (e.g., price changes, product removal) to authorized personnel only.

5.2 Architectural Strategies

5.2.1. Technology Stack

Backend Framework: Spring Boot

- Reasoning: Spring Boot is chosen for its ability to create stand-alone, production-grade Spring-based applications with minimal configuration. It provides embedded servers, which simplify deployment, and integrates well with various data access technologies.
- Trade-offs: While Spring Boot offers extensive features, it might introduce complexity for simple applications. However, the benefits in scalability, performance, and security outweigh these concerns.

Frontend Framework: React.js

- Reasoning: React.js is selected for its component-based architecture, which allows for the development of reusable UI components. It enhances performance through virtual DOM and provides a rich ecosystem of libraries.
- Trade-offs: React.js has a steep learning curve and requires additional libraries for state management and routing (e.g., Redux, React Router). The trade-off is justified by the robust user experience React can deliver.

Database: MySQL

- Reasoning: MySQL is chosen for its robustness, advanced features (such as ACID compliance, complex queries, and indexing), and strong community support. It ensures reliable and efficient data management.
- Trade-offs: MySQL may require more initial setup and maintenance compared to simpler databases like SQLite. However, its scalability and feature set make it suitable for a growing e-commerce platform.

5.2.2. API Design

RESTful API Principles

- Reasoning: RESTful APIs provide a clear and consistent way to structure endpoints and HTTP methods, ensuring that the system is easily understandable and maintainable.
- Trade-offs: RESTful APIs can sometimes be less efficient for complex transactions compared to other protocols like GraphQL. However, the simplicity and widespread adoption of REST justify its use.

5.2.3. Error Detection and Recovery

Centralized Exception Handling

- Reasoning: Implementing centralized exception handling using Spring Boot's @ControllerAdvice allows for consistent error responses and logging, simplifying debugging and enhancing user experience.
- Trade-offs: Centralized handling may obscure the source of some errors if not properly logged. However, it significantly improves maintainability and consistency.

5.2.4. Security

Spring Security Integration

- Reasoning: Spring Security is integrated to handle authentication and authorization, providing robust security mechanisms and simplifying the implementation of secure access controls.
- Trade-offs: Spring Security can be complex to configure for specific use cases. However, its comprehensive security features and integration with Spring Boot make it a strong choice.

5.2.5. Data Persistence and Management

Spring Data JPA

- Reasoning: Spring Data JPA simplifies database interactions by providing a high-level abstraction over common data access tasks, reducing boilerplate code and improving developer productivity.
- Trade-offs: Abstraction can sometimes lead to less control over SQL queries. However, the benefits in productivity and maintainability outweigh these concerns.

5.2.6. User Interface Paradigms

Component-Based UI (React.js)

- Reasoning: Using a component-based approach with React.js allows for the creation of modular, reusable UI components, improving maintainability and scalability of the front-end codebase.
- Trade-offs: Requires a careful design to manage component state and props efficiently. The trade-off is justified by the enhanced reusability and maintainability.

5.2.7. Future Plans for Extending or Enhancing the Software

Microservices Architecture

- Reasoning: As the application grows, transitioning to a microservices architecture can help manage complexity, improve scalability, and facilitate independent deployment of services.
- Trade-offs: Microservices introduce challenges in terms of inter-service communication, data consistency, and overall system complexity. However,

the long-term benefits in scalability and maintainability make it a strategic direction.

GraphQL for Complex Queries

- Reasoning: Introducing GraphQL for complex data retrieval scenarios can optimize performance by allowing clients to request exactly the data they need.
- Trade-offs: GraphQL requires additional setup and learning, but the flexibility and efficiency it provides can significantly enhance the system's performance for specific use cases.

5.3 Coupling and Cohesion

The frontend and backend communicate through RESTful APIs, which allows for loose coupling between the two services. The frontend sends HTTP requests to the backend, and the backend responds with the requested data in JSON format. This approach allows for changes to be made to either the frontend or backend without affecting the other, as long as the API contract is maintained.

In terms of cohesion, each service in our project has a single, well-defined responsibility. The frontend is responsible for handling user interactions and displaying data, while the backend is responsible for processing data and providing it to the frontend through the API. This high level of cohesion makes it easier to maintain and modify each service independently.

To improve the levels of coupling and cohesion in my project, we used clear and consistent API contracts to ensure that the frontend and backend could communicate effectively. I also made sure to keep each service's responsibilities well-defined and separate, which helped to maintain high levels of cohesion.

To test and explore our APIs, we used Postman, a popular API client that allows us to send HTTP requests to our API endpoints and view the responses. With Postman, we can easily test our API endpoints, validate the responses, and even generate code snippets for our API requests. This helped us to ensure that our API was working correctly and that the frontend and backend were able to communicate effectively.

5.4 Design Principles

Our design follows the SOLID principles, ensuring adaptability to new or changing requirements. Here's how each principle is adhered to, along with specific improvements made:

Single Responsibility Principle (SRP)

- Current Design: Each microservice in our architecture has a well-defined responsibility. For instance, the UserService handles all user-related operations, and the ProductService manages product-related tasks.
- Proof and Improvement: Initially, the responsibilities of different entities were intertwined, leading to complex code. By refactoring, each service now has a single responsibility, improving maintainability and reducing complexity.

Open/Closed Principle (OCP)

- Current Design: Our system is designed to be open for extension but closed for modification. New features can be added through extension rather than altering existing code.
- Proof and Improvement: Previously, adding new features required modifying existing classes, risking the introduction of bugs. By leveraging interfaces and abstract classes, new functionalities are now added via extensions, preserving the integrity of existing code.

Liskov Substitution Principle (LSP)

- Current Design: Subclasses can replace their base classes without affecting the correctness of the program. We use well-defined interfaces that all subclasses implement.
- Proof and Improvement: Originally, some subclasses violated LSP by not fully implementing base class functionalities. Refactoring ensured all subclasses properly adhere to their base classes' contracts, ensuring reliable and predictable behavior.

Interface Segregation Principle (ISP)

- Current Design: We have created specific interfaces for distinct functionalities, preventing clients from depending on interfaces they do not use.
- Proof and Improvement: Before improvements, interfaces were too broad, leading to implementation of unnecessary methods in some classes. By breaking down these broad interfaces into more specific ones, each class now only implements what it actually needs.

Dependency Inversion Principle (DIP)

- Current Design: High-level modules do not depend on low-level modules but on abstractions. Dependency injection is used to inject dependencies, making the system more flexible and easier to test.
- Proof and Improvement: Initially, high-level modules were directly dependent on low-level modules, leading to tight coupling. Implementing dependency injection via frameworks like Spring Boot significantly reduced coupling and enhanced flexibility.

Previous Design

- Issue: The UserService class directly depended on the UserRepository implementation, violating DIP and making unit testing difficult.

Improved Design

- Solution: Refactored to use dependency injection and interfaces.
- Explanation: This change ensures that the UserService depends on the UserRepository interface, not its implementation, promoting flexibility and easier testing.

5.5 Design Patterns

5.5.1. Dependency Injection (DI) Pattern

Dependency Injection reduces the coupling between classes and promotes code reuse, testability, and flexibility. SpringBoot uses DI extensively to manage the lifecycle of beans and their dependencies.

SpringBoot DI: The @Autowired annotation is used to inject dependencies into Spring-managed beans.

```
@Service
public class UserServiceImpl implements UserService {
    private final UserRepository userRepository;

    @Autowired
    public UserServiceImpl(UserRepository userRepository){
        this.userRepository = userRepository;
    }

    @Override
    public List<User>getAllUsers(){
        return userRepository.findAll();
    }
}
```

React Context API: In the frontend, React's Context API can be used for dependency injection, providing a way to pass data through the component tree without having to pass props down manually at every level.

```
ReactDOM.createRoot(document.getElementById("root")!).render(
  <BrowserRouter>
    <QueryClientProvider client={queryClient}>
      <Provider store={store}>
```

```
        <PersistGate loading={null} persistor={persistor}>
          <App />
        </PersistGate>
      </Provider>
    </QueryClientProvider>
  </BrowserRouter>
);
```

5.5.2. Repository Pattern

The Repository pattern abstracts the data layer, providing a clean API for data access and manipulation. It decouples the business logic from data access logic, making the system more modular and easier to test.

Spring Data JPA Repositories: Define repository interfaces that extend JpaRepository to provide CRUD operations without boilerplate code.

```
public interface UserRepository extends
JpaRepository<User,Integer> {
    User findByUsername(String username);
    User findByEmail(String email);
}
```

5.5.3. Strategy Pattern

The Strategy pattern allows the definition of a family of algorithms, encapsulates each one, and makes them interchangeable. It promotes the Open/Closed Principle by enabling new strategies to be added without modifying existing code.

React Custom Hooks: Custom hooks encapsulate reusable logic for state management and side effects. This makes the logic interchangeable and composable.

```
import { useQuery } from "@tanstack/react-query";
import client from "../clientAPI";
import { useSelector } from "react-redux";
import { selectCurrentUserId } from "../../redux/auth/authSlice";
const getOrders = async (user_id: string | null) => {
  const response = await client.get(`orders/user/${user_id}`);
  return response.data;
};

const useOrders = () => {
```

```
  const user_id = useSelector(selectCurrentUserId);
  return useQuery({
    queryKey: ["orders"],
    queryFn: () => getOrders(user_id),
  });
};

export default useOrders;
```

5.5.4. Factory Pattern

The Factory pattern provides a way to create objects without specifying the exact class of the object that will be created. This promotes loose coupling and flexibility in the creation of objects.

Spring Beans: Spring's @Bean annotation defines factory methods for creating bean instances. This centralizes configuration and promotes reuse.

```
public class SecurityConfig {
    private final JwtAuthFilter authFilter;
    @Autowired
    public SecurityConfig(JwtAuthFilter authFilter) {
        this.authFilter = authFilter;
    }
    // User creation
    @Bean
    public UserDetailsService userDetailsService() {
        return new UserInfoService();
    }
    // Configuring HttpSecurity
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
    }
}
```

5.5.5. Singleton Pattern

The Singleton pattern ensures a class has only one instance and provides a global point of access to it. Spring Boot beans are singletons by default, promoting resource sharing and consistency.

Spring Boot Singleton Beans: Services and repositories are defined as singletons by default, ensuring a single instance per application context.

```
@Service
public class UserServiceImpl implements UserService {
     private final UserRepository userRepository;

    @Autowired
    public UserServiceImpl(UserRepository userRepository){
        this.userRepository = userRepository;
    }
}
```

5.5.6. Observer Pattern

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. This is essential for managing state and rendering updates in user interfaces.

Redux in React: Redux's store acts as the subject, and components subscribe to state changes. When actions are dispatched, the store notifies all subscribed components about the state updates.

```
const store = createStore(rootReducer);

const MyComponent = () => {
    const state = useSelector(state => state.someState);
    const dispatch = useDispatch();

    const handleAction = () => {
        dispatch(someActionCreator());
    };

    return (
        <div>
            {state}
            <button onClick={handleAction}>Do Action</button>
        </div>
    );
};
```

5.5.7. Command Pattern

The Command pattern encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations. This is particularly useful for implementing undoable operations and handling actions as objects.

Redux Actions: Actions in Redux encapsulate all the information needed to perform a certain operation, and these actions are dispatched to modify the state.

```
import { createSlice } from "@reduxjs/toolkit";
import { RootState } from "../store";
type authState = {
  isSignedIn: boolean;
  user_id: string | null;
  session_id: string | null;
};
const initialState: authState = {
  isSignedIn: false,
  user_id: "",
  session_id: "",
};
const authSlice = createSlice({
  name: "auth",
  initialState,
  reducers: {
    logIn: (state, action) => {
      const { userId, sessionId } = action.payload;
      state.isSignedIn = true;
      state.user_id = userId;
      state.session_id = sessionId;
    },
    logOut: (state) => {
      state.isSignedIn = false;
      state.session_id = "";
      state.user_id = "";
    },
  },
});
export const { logIn, logOut } = authSlice.actions;
export default authSlice.reducer;
export const selectCurrentIsSignedIn = (state: RootState) =>
  state.auth.isSignedIn;
export const selectCurrentSessionId = (state: RootState) =>
  state.auth.session_id;
export const selectCurrentUserId = (state: RootState) =>
state.auth.user_id;
```

5.5.8. Template Method Pattern

The Template Method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of an algorithm without changing its structure.

Abstract Service Methods: Base service classes can provide default implementations of certain methods, while allowing subclasses to override specific steps.

```
public abstract class BaseService<T> {
    public T save(T entity) {
        // default implementation
    }

    public abstract void validate(T entity);
}
```