IIIT Hyderabad Mart

Architectural & Implementation Report

Group 22

Ashish Lakhmani (2023202008)
Rugved Thakare (2023201049)
Pratik Singh (2023201082)
Rohit Joshi (2023202016)

GitHub Repository:
https://github.com/RugvedThakare/IIITH$_{M}ART$

April 17, 2025

Contents

1 Task 1: Requirements and Subsystems

1.1 Functional Requirements

The detailed functional requirements include:

- **User Account Management:** User registration, secure login/logout, session handling, and authentication.
- **Product Catalog Management:** Management of product lifecycle, including adding, updating, deleting, viewing, and filtering.
- **Shopping Cart Operations:** Functionality for adding, viewing, updating, and clearing cart items.
- **Order Processing:** Comprehensive order creation workflow including validation, transaction handling, and order history management.
- **Profile Management:** Updating of personal details, addresses, payment information, and account deletion.

1.2 Non-Functional Requirements

Key non-functional requirements:

- **Security:** Secure session management, token expiration, authentication, and role-based authorization.
- **Performance:** Low latency, especially for authentication and checkout processes.
- **Scalability:** Capability to horizontally scale as user traffic increases.
- **Data Consistency and Reliability:** Transactional integrity to ensure ACID properties.
- **Maintainability and Modularity:** Modular architecture enabling ease of maintenance and future enhancements.
- **Usability:** Clear, RESTful API design, and comprehensive API documentation.

1.3 Subsystem Overview

The system is divided into distinct, well-defined subsystems:

- **Authentication and Session Management**
- **Customer Management**
- **Seller Management**
- **Product Catalog Management**
- **Shopping Cart Operations**
- **Order Management**

2 Task 2: Architecture Framework

2.1 Stakeholder Identification (IEEE 42010)

Detailed stakeholder analysis:

Stakeholder	**Concerns**
End Users (Customers)	Security of personal information, system usability, and performance.
Admin Users (Sellers)	Data integrity, system reliability, and authorization.
Development Team	Code maintainability, modularity, and rapid development cycles.
System Administrators	Deployment ease, scalability, and monitoring capabilities.
Business Owners	Cost-effectiveness, system scalability, and compliance with security standards.

2.2 Major Design Decisions (ADRs)

1. **Monolithic Architecture**: Chosen for reduced complexity and higher performance through direct internal method calls, ideal for initial project scale.

2. **Technology Stack (Spring Boot & MySQL)**: Spring Boot provides rapid development capabilities and built-in support for essential features such as transactions, security, and REST API standards. MySQL offers robust transaction support ensuring ACID compliance.

3. **Custom Session Tokens**: Selected over JWT for simplicity, clearer control over session lifecycle, and straightforward debugging.

4. **Separate Customer and Seller Entities**: Decided for clear role distinction, simplifying access controls, and domain logic management.

5. **Layered Architecture**: Clearly separates responsibilities into Controller, Service, and DAO layers, facilitating maintenance, testing, and independent development.

3 Task 3: Architectural Tactics and Patterns

3.1 Architectural Tactics

Detailed architectural tactics addressing non-functional requirements:

- **Authentication & Authorization:** Robust token-based authentication and role verification.
- **Input Validation & Exception Handling:** Comprehensive data validation and consistent exception handling for enhanced reliability.
- **Transactional Operations:** Usage of database transactions to ensure data integrity and consistency.
- **Layered Design for Modularity:** Clear separation between presentation, business, and persistence layers.
- **Caching & Session Optimization:** Efficient session handling and potential caching strategies to enhance performance.

3.2 Implementation Patterns

The following design patterns have been implemented clearly in the architecture:

- **Layered Architecture:** Clearly separates responsibilities into distinct layers to enhance maintainability and modifiability:
  - **Controller Layer:** Handles HTTP requests and responses, validates request data, and performs initial authentication checks.
  - **Service Layer:** Contains business logic, enforces business rules, and ensures proper transaction management. Acts as an intermediary between controllers and data access layers.
  - **DAO (Data Access Object) Layer:** Encapsulates all database operations, using repository interfaces and Spring Data JPA to abstract database interactions, enhancing ease of testing and database portability.
- **MVC (Model-View-Controller) Pattern:** Clearly separates application functionality into three interconnected parts:
  - **Model:** Represents data entities and domain logic.
  - **View:** In this REST API context, views are represented by JSON responses.
  - **Controller:** Manages API endpoints and delegates business logic to services.
- **Repository Pattern:** Abstracts the underlying database implementation by using interfaces to provide data operations. Repositories reduce boilerplate code and simplify data access methods.
- **Domain Model Pattern:** Defines rich domain objects representing real-world entities with clear attributes and relationships, facilitating intuitive domain-driven design and better object-oriented practices.
- **Singleton and Dependency Injection Pattern:** Spring's IoC container creates single instances (Singleton) of services and repositories, injecting these dependencies where needed. This enhances modularity, testability, and manageability.
- **RESTful API Design:** Adheres strictly to REST principles, using standard HTTP methods (GET, POST, PUT, DELETE) and status codes to interact with resources, ensuring interoperability and clear API semantics.

3.3 Entity-Relationship Diagram

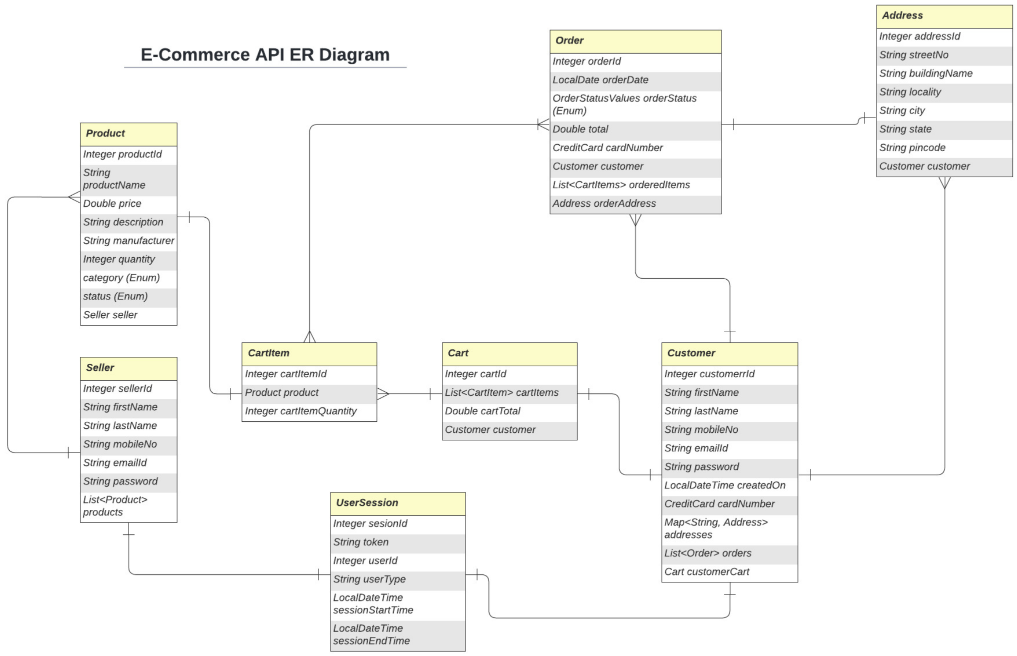

Figure 1: Entity-Relationship Diagram depicting detailed domain model relationships

4 Task 4: Prototype Implementation and Analysis

4.1 Prototype Implementation

Detailed description of implemented core workflows:

- Comprehensive user registration, login, and session management.
- Detailed product catalog management including product lifecycle operations.
- Robust shopping cart functionalities ensuring product availability and accurate updates.
- Complex transactional order placement, ensuring data consistency and accurate inventory handling.
- Extensive profile management supporting personal, address, and payment details.

4.2 API Module Endpoints

4.2.1 Login & Logout Module

- **POST /register/customer**: Registers a new customer with details such as email, mobile number, and password.
- **POST /login/customer**: Authenticates customer using mobile number and password.
- **POST /logout/customer**: Logs out the customer by invalidating session token.
- **POST /register/seller**: Registers a new seller with necessary seller credentials.
- **POST /login/seller**: Authenticates seller.
- **POST /logout/seller**: Logs out the seller by invalidating session token.

4.2.2 Customer Module

- **GET /customer/current**: Retrieves currently logged-in customer's details.
- **GET /customer/orders**: Provides order history for the logged-in customer.
- **GET /customers**: Lists all registered customers.
- **PUT /customer**: Updates the logged-in customer's profile information.
- **PUT /customer/update/password**: Updates customer's password securely.
- **PUT /customer/update/card**: Updates customer's credit card details.
- **PUT /customer/update/address?type=home**: Updates the customer's home address.
- **PUT /customer/update/credentials**: Updates email address and mobile number.
- **DELETE /customer**: Deletes logged-in customer account securely.
- **DELETE /customer/delete/address?type=home**: Deletes the customer's home address.

4.2.3 Seller Module

- **GET /seller/{sellerid}**: Retrieves details for a specific seller.
- **GET /seller/current**: Fetches the current logged-in seller's details.
- **GET /sellers**: Lists all registered sellers.
- **POST /addseller**: Adds a new seller to the system.
- **PUT /seller**: Updates existing seller's profile details.
- **PUT /seller/update/password**: Updates seller password.
- **PUT /seller/update/mobile**: Updates seller's mobile number.
- **DELETE /seller/{sellerid}**: Deletes the seller by specified ID.

4.2.4 Product Module

- **GET /product/{id}**: Retrieves product details by product ID.
- **GET /products**: Lists all products available.
- **GET /products/{category}**: Retrieves products by category.
- **GET /products/seller/{id}**: Lists products offered by a specific seller.
- **POST /products**: Adds a new product to the catalog.
- **PUT /products**: Updates product details.
- **PUT /products/{id}**: Updates the quantity of a specific product.
- **DELETE /product/{id}**: Deletes a product by ID.

4.2.5 Cart Module

- **GET /cart**: Retrieves all items in the customer's cart.
- **POST /cart/add**: Adds an item to the cart.
- **DELETE /cart**: Removes a specific item from the cart.
- **DELETE /cart/clear**: Clears all items from the cart.

4.2.6 Order Module

- **GET /orders/{id}**: Retrieves order details by ID.
- **GET /orders**: Retrieves all orders.
- **GET /orders/by/date**: Retrieves orders placed on a specific date (format: DD-MM-YYYY).
- **POST /order/place**: Places a new order based on current cart contents.
- **PUT /orders/{id}**: Updates an existing pending order.
- **DELETE /orders/{id}**: Cancels an order.

Frontend Setup (Next.js)

Navigate to the frontend directory and start the Next.js server:

```
cd ecommerce-frontend
npm run dev
```

API Root Endpoint:
https://localhost:3000/

Backend Setup (Spring Boot)

Navigate to the backend directory, build the project, and run the Spring Boot application:

```
cd E-Commerce-Backend
mvn clean install
./mvnw spring-boot:run
```

Important Configuration Steps:
Before running the backend API server, ensure that the database configuration is correctly set. Update the following details in the file `application.properties`, located at:

```
/E-Commerce-Backend/src/main/resources/application.properties
```

Configure the port number, database URL, username, and password as per your local MySQL database setup:

```
server.port=8009

spring.datasource.url=jdbc:mysql://localhost:3306/ecommercedb
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=root
```

Ensure your MySQL server is running locally and the specified database (`ecommercedb`) exists.
API Root Endpoint:
https://localhost:8009/

4.3 Architectural Analysis

4.3.1 Comparative Analysis: Monolithic vs. Microservices

Detailed comparative analysis highlighting trade-offs:

- **Latency:** Monolith provides lower latency due to internal method calls, while microservices introduce additional network overhead.
- **Throughput & Scalability:** Microservices offer granular scalability and potentially higher throughput, especially under high load conditions.
- **Fault Tolerance & Reliability:** Microservices provide better fault isolation, reducing risk from component failures.
- **Development Complexity & Maintainability:** Monolithic architecture is simpler initially but can face maintainability challenges as the system scales.

4.3.2 Quantitative Analysis

Quantitative comparisons demonstrating specific trade-offs:

- **Latency Example:** Monolithic product retrieval approximately 50ms vs. Microservices approximately 70ms.
- **Throughput Example:** Monolithic order processing 100 orders/minute per instance vs. Microservices higher potential throughput due to independent scalability.