\WOOLF/

Applied Software Project Report

By

Sanjay Yadav

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

January, 2025

SCALER

Scaler Mentee Email ID : shubh24i@gmail.com

Thesis Supervisor : Naman Bhalla

Date of Submission : 21/10/2025

Certification

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

Naman Bhalla

……

Project Guide / Supervisor

DECLARATION

I confirm that this project report, submitted to fulfill the requirements for the Master of Science in Computer Science degree, completed by me from 12 March 2024 to 9 June 2024, is the result of my own individual endeavor. The Project has been made on my own under the guidance of my supervisor with proper acknowledgement and without plagiarism. Any contributions from external sources or individuals, including the use of AI tools, are appropriately acknowledged through citation. By making this declaration, I acknowledge that any violation of this statement constitutes academic misconduct. I understand that such misconduct may lead to expulsion from the program and/or disqualification from receiving the degree.

Sanjay Yadav

Date: January 2025

ACKNOWLEDGMENT

I would like to take this moment to express my heartfelt gratitude to my family, especially my mother, who has always supported me and encouraged me to move forward in life. I am deeply thankful to my better half and my children for their incredible support throughout this journey. During the course, along with my job, I had very little time to spend with them, but they understood and never pressured me to spend time with them or go on trips during my study hours. Their unwavering support and understanding gave me the strength and motivation I needed to overcome challenges.

I am also grateful to Scaler for providing me with this amazing opportunity, a well-structured platform, and an interactive course that made learning enjoyable and effective. A special thanks to my Scaler instructors for their guidance, expertise, and dedication, which were instrumental in helping me develop new skills and grow as a professional. Finally, I extend my heartfelt thanks to my Scaler peers, who inspired and motivated me through their words, actions, and examples. This achievement reflects the collective support and encouragement I have been fortunate to receive, and I am truly thankful to each one of you.

Table of Contents

List of Tables

(To be written sequentially as they appear in the text)

List of Figures

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

Applied Software Project

Abstract

This project focuses on developing an ecommerce website designed to simplify online shopping and enhance the user experience. The platform integrates essential functionalities such as user management, product catalog browsing, cart and checkout, order management, and secure payment processing. By leveraging modern technologies and best practices, the system aims to create a seamless and intuitive interface for users while ensuring data security and operational efficiency.

The project addresses real-life applications in the retail industry by offering a digital platform that can scale to meet the needs of diverse businesses. It enhances accessibility for customers by providing features like secure registration, personalized profiles, and real-time order tracking. The inclusion of multiple payment options ensures user convenience, while secure authentication mechanisms guarantee data privacy. Overall, this project demonstrates how software tools can revolutionize traditional shopping experiences, making them more efficient and accessible across industries.

Project Description

This project involves designing and developing a feature-rich ecommerce website tailored to meet modern online shopping needs.

Objectives

Provide a user-friendly platform for customers to browse and purchase products.

Enable businesses to manage products, orders, and transactions efficiently.

Ensure a secure and seamless shopping experience for users.

Relevance

Ecommerce has become a cornerstone of modern retail, enabling businesses to reach wider audiences. This platform is designed to address common pain points, such as secure user authentication, product search, and efficient order tracking, making it relevant for industries aiming to optimize their online presence.

Captone Project Development Process

The development process for the ecommerce website was structured into distinct phases to ensure clarity, efficiency, and alignment with project goals. The following outlines the key steps taken during the project lifecycle:

1. Definition

The first phase involves understanding the project requirements, setting goals, and defining the scope of the eCommerce platform. Key objectives include:

- **Understanding Requirements:**
  - Create a detailed Product Requirements Document (PRD) outlining functional and non-functional requirements.
  - Define core modules: User Management, Product Catalog, Cart & Checkout, Order Management, Payment, and Authentication.
- **Setting Objectives:**
  - Provide a user-friendly eCommerce platform with secure authentication, seamless checkout, and robust payment features.
  - Ensure scalability to handle a growing user base and product catalog.
- **Identifying Constraints:**
  - Timeframe for development.
  - Resources and technologies (e.g., Java, Spring Boot, SQL database).

2. Planning

This phase involves creating a roadmap and breaking the project into manageable tasks.

- **Technology Selection:**
  - Backend: Java with Spring Boot for API development.
  - Database: MySQL for data storage.
- **Defining Milestones:**
  - Milestone 1: User Management module.
  - Milestone 2: Product Catalog with search functionality.

    - Milestone 3: Cart & Checkout integration.
    - Milestone 4: Order Management and Payment processing.
    - Milestone 5: Secure Authentication and final testing.
- **Resource Allocation:**
    - Organize individual efforts.
    - Created sprint for small task
    - Use project management tools like Jira to track progress.

- **Design Phase:**
    - Design database schemas (e.g., user, product, order tables).
    - Create class diagrams for modules.

3. Development

The implementation of the planned features occurs in this phase.

- **Backend Development:**
    - Build APIs for User Management (registration, login, profile update).
    - Develop endpoints for browsing and searching the product catalog.
    - Implement cart and order functionality.
    - Create secure payment APIs integrating with payment gateways.
- **Authentication:**
    - Implement secure login and session management using JWT.
    - Add password hashing with BCrypt for security.
- **Database Development:**
    - Set up tables for users, products, orders, categories, etc.
    - Optimize queries for faster data retrieval.
- **Testing:**
    - Perform unit testing for each module.
    - Conduct integration and end-to-end testing to ensure a seamless experience.
    - Address bugs and ensure functionality aligns with requirements.

4. Delivery

The final phase involves deploying the project and ensuring a smooth user experience.

- **Deployment:**
  - Host the application on a cloud platform (e.g., AWS, Azure).
  - Use Docker for containerization to simplify deployment.
  - Configure CI/CD pipelines for automatic testing and deployment.

Requirement Gathering

Table1

1. Functional Requirements:

Identified key features such as

User Management: Secure registration, profile management, and session handling.

Product Catalog: Browsing products by categories, viewing detailed product information, and using a search feature.

Cart and Checkout: Adding products to the cart, reviewing items, and finalizing purchases.

Order Management: Viewing order history, tracking deliveries, and receiving order confirmations.

Payment Integration: Offering multiple payment options and ensuring secure transactions.

Product Requirements Document (PRD) for Ecommerce Website

1. User Management

1.1. Registration: Allow new users to create an account using their email or social media profiles.

1.2. Login: Users should be able to securely log in using their credentials.

1.3. Profile Management: Users should have the ability to view and modify their profile details.

1.4. Password Reset: Users must have the option to reset their password through a secure link.

2. Product Catalog

2.1. Browsing: Users should be able to browse products by different categories.

2.2. Product Details: Detailed product pages with product images, descriptions, specifications, and other relevant information.

2.3. Search: Users must be able to search for products using keywords.

3. Cart & Checkout

3.1. Add to Cart: Users should be able to add products to their cart.

3.2. Cart Review: View selected items in the cart with price, quantity, and total details.

3.3. Checkout: Seamless process to finalize the purchase, including specifying delivery address and payment method.

4. Order Management

4.1. Order Confirmation: After making a purchase, users should receive a confirmation with order details.

4.2. Order History: Users should be able to view their past orders.

4.3. Order Tracking: Provide users with a way to track their order's delivery status.

5. Payment

5.1. Multiple Payment Options: Support for credit/debit cards, online banking, and other popular payment methods.

5.2. Secure Transactions: Ensure user trust by facilitating secure payment transactions.

5.3. Payment Receipt: Provide users with a receipt after a successful payment.

6. Authentication

6.1. Secure Authentication: Ensure that user data remains private and secure during login and throughout their session.

6.2. Session Management: Users should remain logged in for a specified duration or until they decide to log out.

2. Non-Functional Requirement

For my capstone project, I adopted the Model-View-Controller (MVC) architectural pattern with a tailored approach, omitting the View layer, as the focus of the project is backend development. Using Java Spring Boot, I concentrated on building robust Models, Controllers, and Services to ensure scalability, modularity, and clean code organization.

Multi-Module Monolithic Design

1. **Monolithic Architecture with Modularity**

   While the application follows a monolithic structure, I designed it with a multi-module

approach that mimics microservices. Each key module, such as User Management, Product Catalog, Cart, Order, Checkout, and Payment, is encapsulated with its own:

- **Model:** For database entity mapping and business objects.
- **Service:** For handling business logic.
- **Controller:** For managing API endpoints and routing.

2. **Scalability in Focus**

   This modular design ensures that if any specific module, like Product Catalog or Payment, experiences high traffic or load in the future, it can be seamlessly extracted and converted into a microservice. This forward-thinking approach provides flexibility for scaling without requiring significant restructuring of the codebase.

MVC Without View Layer

- The Model represents the application's data structure, using Java Persistence API (JPA) to interact with the database.
- The Service layer handles business logic, ensuring separation of concerns and making the codebase easier to maintain and test.
- The Controller layer manages incoming requests, routing them to the appropriate services, and returning responses in JSON format, suitable for API consumers like frontend applications or mobile apps.

Since the project does not include a View layer, the focus remains on providing a backend API that can be consumed by any frontend or mobile application, allowing flexibility in integrating different client-side technologies in the future.

Key Benefits of the Approach

1. **Scalability:** The modular monolithic design ensures that the system can handle growth by enabling gradual migration to microservices as needed.
2. **Maintainability:** Each module has a clear separation of concerns, making it easier to update or debug specific parts of the application.
3. **Reusability:** The service layer is designed with reusability in mind, enabling shared logic across different parts of the application.

4. **Clean Code:** By following Spring Boot's best practices and adhering to MVC principles, the codebase remains well-structured and easy to extend.

Project folder structure - Below is the Project Folder Structure Based on the MVC Pattern

onlineshop

```
├── docs
│   ├── api-docs
│   ├── class-diagram
│   ├── shema-diagram
│   └── postman-collection
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com
│   │   │       └── onlineshop
│   │   │           ├── shop
│   │   │           │   ├── common
│   │   │           │   │   ├── dtos
│   │   │           │   │   ├── exceptions
│   │   │           │   │   ├── models
│   │   │           │   ├── config
│   │   │           │   │   ├── GlobalConfig
│   │   │           │   │   ├── WebConfig
│   │   │           │   ├── data
│   │   │           │   │   ├── DataInitializer
│   │   │           │   │   ├── RoleRepository
│   │   │           │   ├── cartAndCheckout
│   │   │           │   │   ├── controllers
│   │   │           │   │   ├── dtos
```

```
│   │   │       │   ├── exceptions
│   │   │       │   ├── models
│   │   │       │   ├── repositories
│   │   │       │   └── services
│   │   │       ├── Auth
│   │   │       │   ├── - -
│   │   │       ├── product
│   │   │       │   ├── - -
│   │   │       ├── order
│   │   │       │   ├── - -
│   │   │       ├── user
│   │   │       │   ├── - -
│   │   │       └── Application.java
│   │   └── resources
│   │       ├── application.properties
│   │       └── templates
│   │           └── index.html
└── pom.xml
```

3. Stakeholder Analysis

Introduction

Stakeholder analysis was a critical component of my e-commerce capstone project. By engaging with key stakeholders, including potential users and business owners, I identified their specific pain points, expectations, and priorities. This process ensured that the system's design and

functionality aligned with their needs, fostering a user-centric and business-aligned approach.

Steps in Stakeholder Analysis

1. Identifying Stakeholders

The first step involved identifying all relevant stakeholders who would interact with or benefit from the e-commerce platform. The primary stakeholders identified were:

- Potential Users: Shoppers who would use the platform to browse, purchase, and manage orders.
- Business Owners: Vendors and platform administrators responsible for managing products, sales, and the overall system.
- Technical Stakeholders: Developers and IT professionals responsible for maintaining the platform.
- Logistics and Delivery Partners: Entities responsible for fulfilling customer orders.

Key Findings from Stakeholder Engagement

Pain Points

1. **For Users**:
    - Difficulty in finding specific products due to poor search and filtering options.
    - Lack of trust due to unreliable payment methods and insecure transactions.
    - Frustration with slow or unclear delivery tracking mechanisms.
2. **For Business Owners**:
    - Challenges in managing product catalogs efficiently, especially with large inventories.

- Limited tools for tracking and analyzing sales performance.
- High operational overhead due to inefficient order and payment management.

Expectations

1. **For Users**:
   - A seamless, intuitive shopping experience with features like product search, recommendations, and detailed descriptions.
   - Secure payment options and transparent pricing.
   - Real-time order tracking and timely delivery notifications.
2. **For Business Owners**:
   - Easy-to-use interfaces for managing products, orders, and payments.
   - Insights into sales trends and customer preferences through analytics.
   - Scalable infrastructure capable of handling high traffic during sales and festive seasons.

Actions Taken Based on Stakeholder Analysis

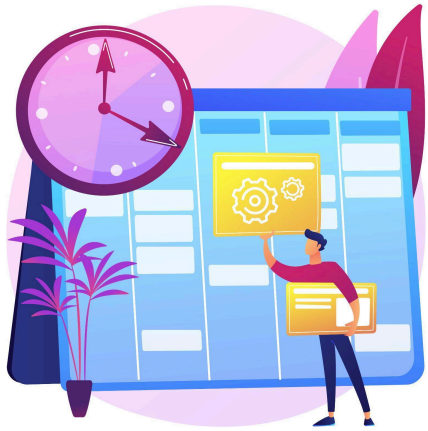
For Users:

1. **Enhanced Product Browsing and Search**:
   - Developed a powerful search functionality with keyword matching and category filtering.
   - Added features like product recommendations and detailed product descriptions.
2. **Secure Transactions**:
   - Integrated multiple payment options with secure gateways to ensure user trust.
   - Provided instant payment confirmations and digital receipts.
3. **Order Tracking**:
   - Built an order tracking feature with real-time status updates and estimated delivery times.

For Business Owners:

1. **Product Catalog Management**:
    - Designed an intuitive interface for adding, updating, and categorizing products.
2. **Order and Payment Tools**:
    - Created modules for easy order management, automated payment reconciliation, and tracking customer transactions.
3. **Scalability**:
    - Developed the system with a modular architecture, ensuring that business owners can handle high traffic and add new features as their operations grow.

4. Planning and Development



Project Plan: Created a roadmap defining milestones, deadlines, and deliverables.

Development Timeline and Sprint Planning

The development process is divided into 5 sprints over a duration of 8 weeks. Each sprint has clear milestones and deliverables.

Table1.1

No.	Title	Page No.
1	Sprint 1: Foundation and Setup (Week 1)	20
2	Sprint 2: User Management Module (Week 2-3)	20
3	Sprint 3: Product Catalog Module (Week 4)	21
4	Sprint 4: Cart and Checkout Module (Week 5-6)	22
5	Sprint 5: Payment Integration and Final Touches (Week 7-8)	23

Sprint 1: Foundation and Setup (Week 1)

Goal: Establish the project's foundational structure.

Deliverables:

1. **Project Initialization**
    - Set up a new Spring Boot project with a monolithic but modular architecture.
    - Configure Gradle/Maven for dependency management.
    - Create basic folder structures: Model, Controller, Service, and Repository.
    - Add configurations for future database integration.
2. **Database Design:**
    - Define a normalized relational database schema using the provided requirements.
    - Create tables for **users**, **roles**, **products**, **categories**, **orders**, **order items**, **carts**, and **cart items**.
3. **Environment Setup:**
    - Configure local development and testing environments.
    - Integrate Swagger for API documentation.

Milestone:

- Project architecture established, and database schema finalized.
- Successfully running a basic application.

Sprint 2: User Management Module (Week 2-3)

Goal: Develop user-related functionalities and secure authentication.

Deliverables:

1. **Model Layer:**
    - Create entities for User, Role, and their relationships.
    - Include fields like name, email, password, and roles.
2. **Service Layer:**
    - Implement user registration and login services.
    - Use BCrypt for password encryption.
    - Add functionality for password reset and profile management.
3. **Controller Layer:**
    - Build RESTful APIs for user registration, login, and profile management.
    - Provide error handling for scenarios like duplicate emails, invalid credentials, etc.
4. **Security Configuration:**
    - Configure Spring Security for secure authentication.
    - Implement JWT-based token management.

Milestone:

- Fully functional user registration and login APIs with secure authentication.
- Swagger API documentation updated for User Management.

Sprint 3: Product Catalog Module (Week 4)

Goal: Enable product browsing, categorization, and detailed views.

Deliverables:

1. **Model Layer:**
    - Create entities for Product, Category, and their relationships.
2. **Service Layer:**
    - Implement product addition, retrieval, and category-based filtering.

3. **Controller Layer:**
    - Build APIs for product listing, product details, and searching by keywords.
4. **Data Seeding:**
    - Seed the database with example products and categories for testing.

Milestone:

- APIs for product catalog browsing and filtering are functional and documented

Sprint 4: Cart and Checkout Module (Week 5-6)

Goal: Allow users to manage carts and place orders.

Deliverables:

1. **Model Layer:**
    - Define Cart, CartItem, and relationships with User and Product.
    - Define Order, OrderItem, and relationships with User and Product.
2. **Service Layer:**
    - Implement functionality to add/remove items in the cart.
    - Calculate cart totals dynamically.
    - Build order placement functionality.
3. **Controller Layer:**
    - Create APIs for cart management, order placement, and order history retrieval.
4. **Error Handling:**
    - Handle scenarios like out-of-stock products and invalid order requests.

Milestone:

- Users can add items to their cart and place orders.
- Cart and order APIs are fully functional and documented.

Sprint 5: Payment Integration and Final Touches (Week 7-8)

Goal: Enable secure payment handling and finalize the project.

Deliverables:

1. **Payment Service:**
    - Implement mock payment processing.
    - Provide APIs for multiple payment methods (credit card, debit card, etc.).
2. **Integration:**
    - Ensure that order placement updates payment status and sends order confirmation.
3. **Testing:**
    - Perform end-to-end testing for all functionalities.
    - Handle edge cases like expired JWT tokens, concurrent cart updates, etc.
4. **Documentation:**
    - Update Swagger API documentation for the entire project.
    - Write detailed README files for deployment and usage.

Milestone:

- Fully functional payment system.
- Application tested, documented, and ready for delivery.

Class Diagram

Designed a class structure to represent key entities such as User, Product, Order, and Payment.

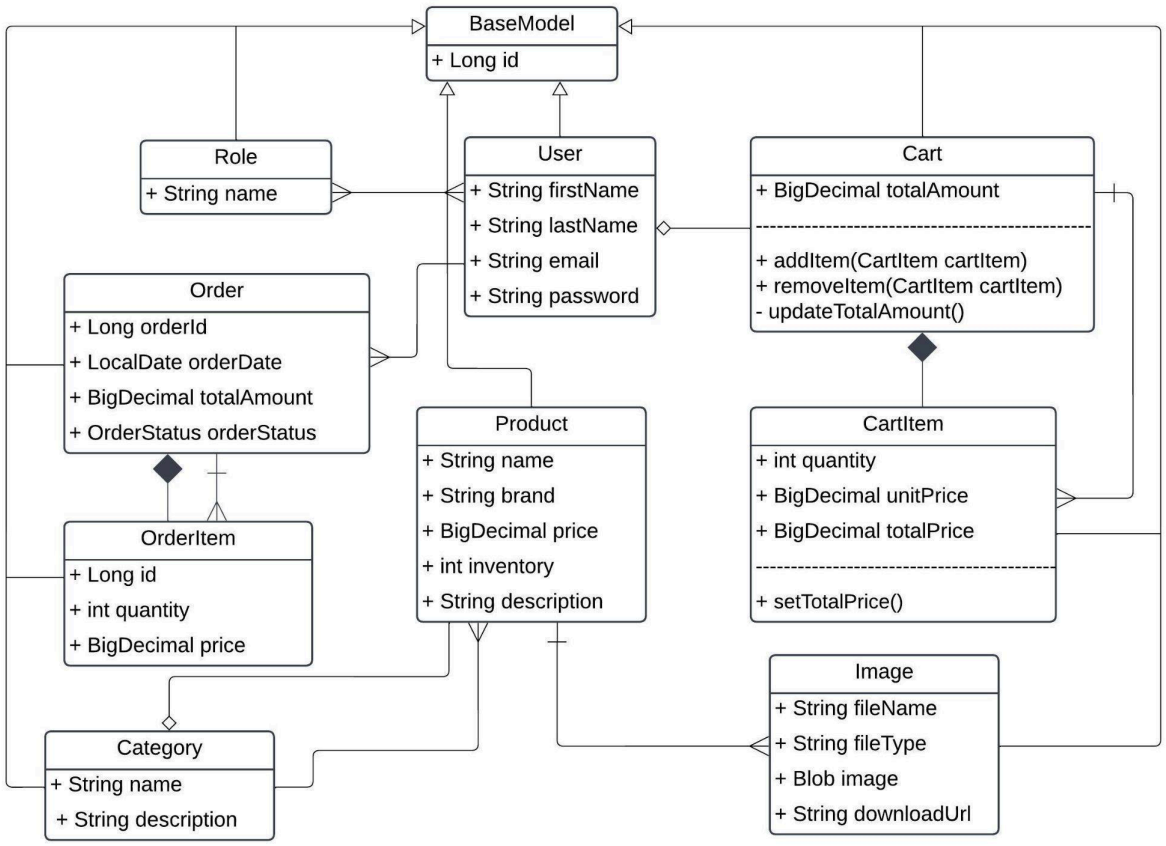


Figure 1: Class Diagram

Relationships:

User-Role: Many-to-Many relationship.

User-Cart: One-to-One relationship.

User-Order: One-to-Many relationship.

Category-Product: One-to-Many relationship.

Product-Image: One-to-Many relationship.

Order-OrderItem: One-to-Many relationship.

OrderItem-Product: Many-to-One relationship.

Cart-CartItem: One-to-Many relationship.

CartItem-Product: Many-to-One relationship.

Database Schema Design

Developed a relational database schema to store user profiles, product details, order information, and payment records.

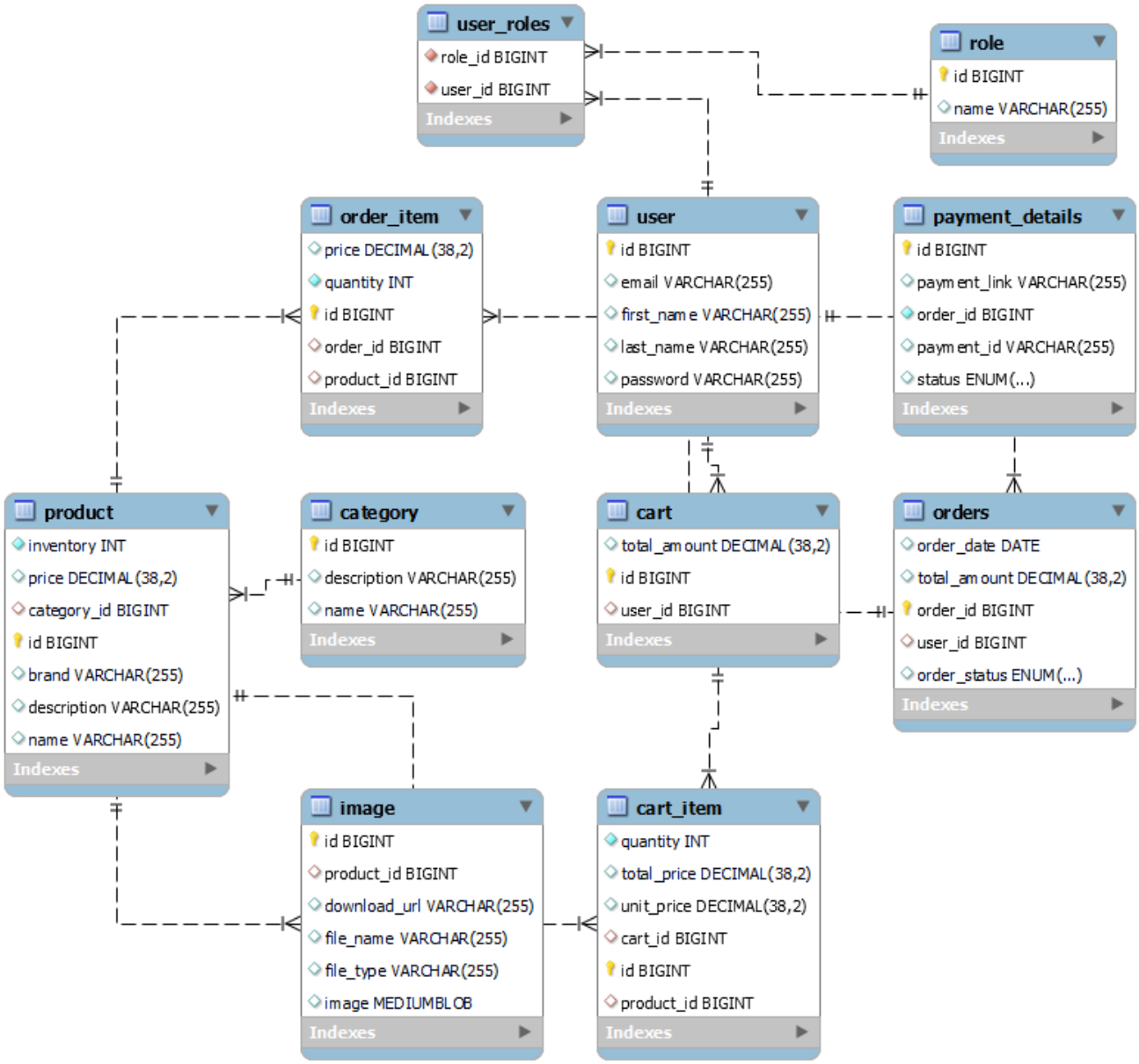


Figure 2: Database schema design

Foreign Keys:

1. **user_roles(user_id)** refers **user(id)**
2. **user_roles(role_id)** refers **role(id)**

3. **order_item(order_id)** refers **orders(id)**
4. **order_item(product_id)** refers **product(id)**
5. **product(category_id)** refers **category(id)**
6. **cart(user_id)** refers **user(id)**
7. **cart_item(cart_id)** refers **cart(id)**
8. **cart_item(product_id)** refers **product(id)**
9. **image(product_id)** refers **product(id)**
10. **orders(user_id)** refers **user(id)**
11. **payment_details(order_id)** refers **orders(id)**

Cardinality of Relations:

1. Between **user_roles** and **user** → **m:1**
2. Between **user_roles** and **role** → **m:1**
3. Between **order_item** and **orders** → **m:1**
4. Between **order_item** and **product** → **m:1**
5. Between **product** and **category** → **m:1**
6. Between **cart** and **user** → **1:1**
7. Between **cart_item** and **cart** → **m:1**
8. Between **cart_item** and **product** → **m:1**
9. Between **image** and **product** → **m:1**
10. Between **orders** and **user** → **m:1**
11. Between **payment_details** and **orders** → **1:1**

Unit Testing

Benefits of Testing

Ensures Quality: Identifies defects early in the development process, ensuring a high-quality product.

Reliability: Confirms that individual components work as intended, increasing the overall system reliability.

Time Efficiency: Reduces time spent debugging and fixing issues later in the development lifecycle.

Improved Code Maintainability: Testing encourages writing modular and clean code that is easier to maintain and extend.

Scope

Individual modules, such as user authentication, product catalog, cart functionality, order processing, and payment functionality, were thoroughly tested to ensure accuracy, reliability, and expected behavior.

Testing Proof

Unit test results Screenshots for critical modules will be attached as proof

User Authentication: Verified registration, login, and role-based access control functionalities.

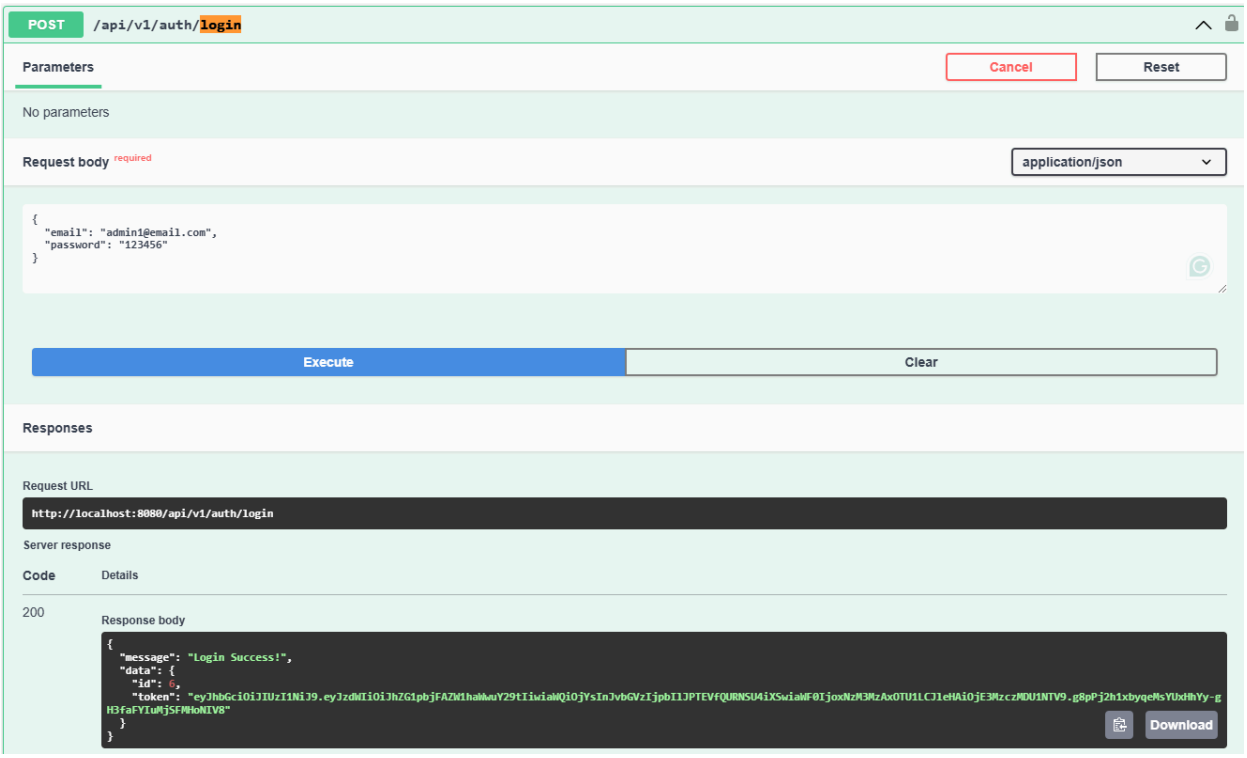


Figure 3: Login screen

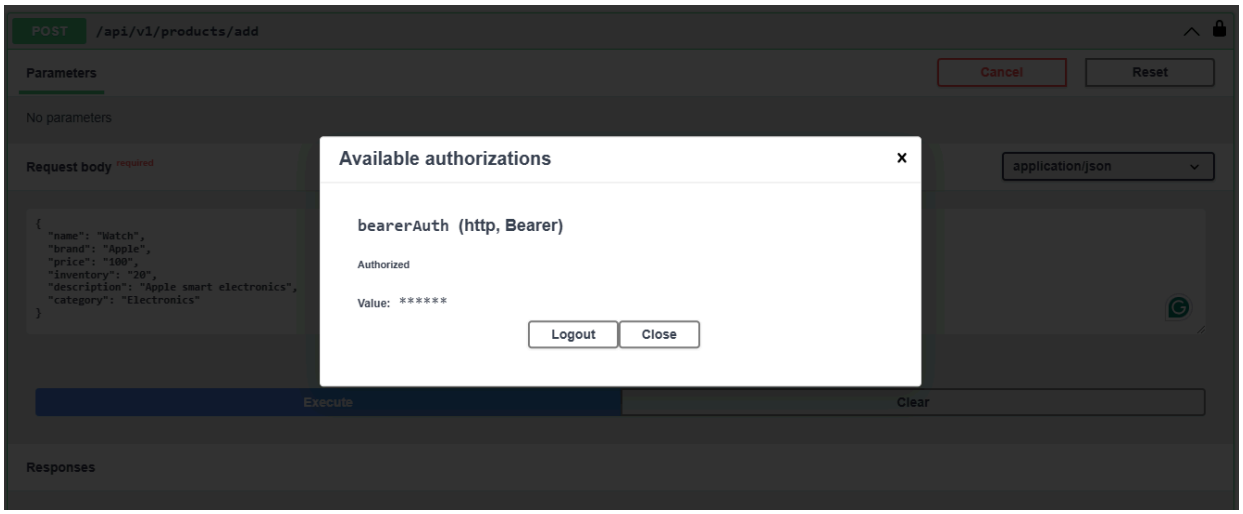


Figure 3.1: Auth token screen

Product Catalog:

- Tested addition, get all, and updating of products.
- Ensured proper functionality of product search by category.

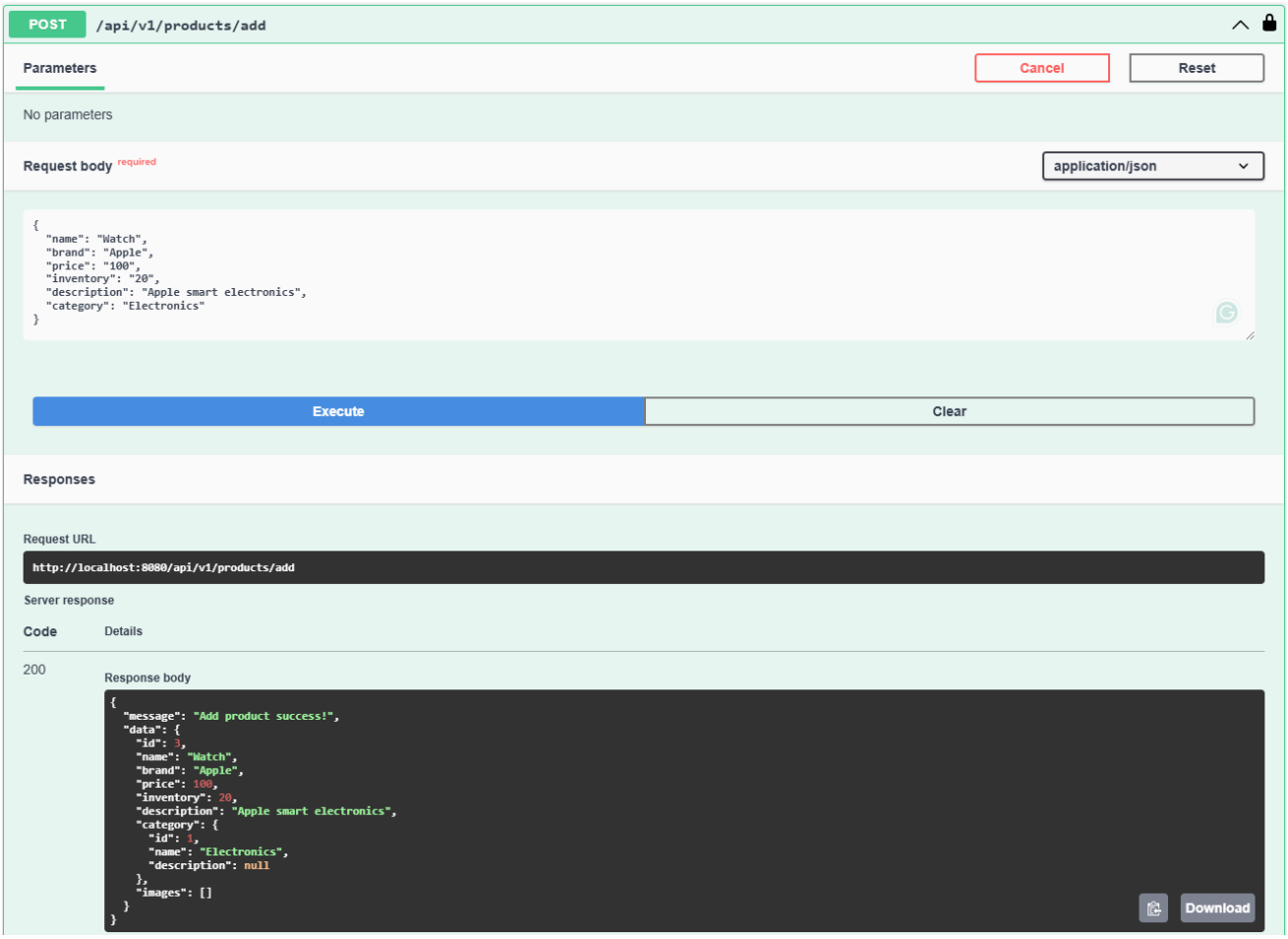


Figure 4: Add Product

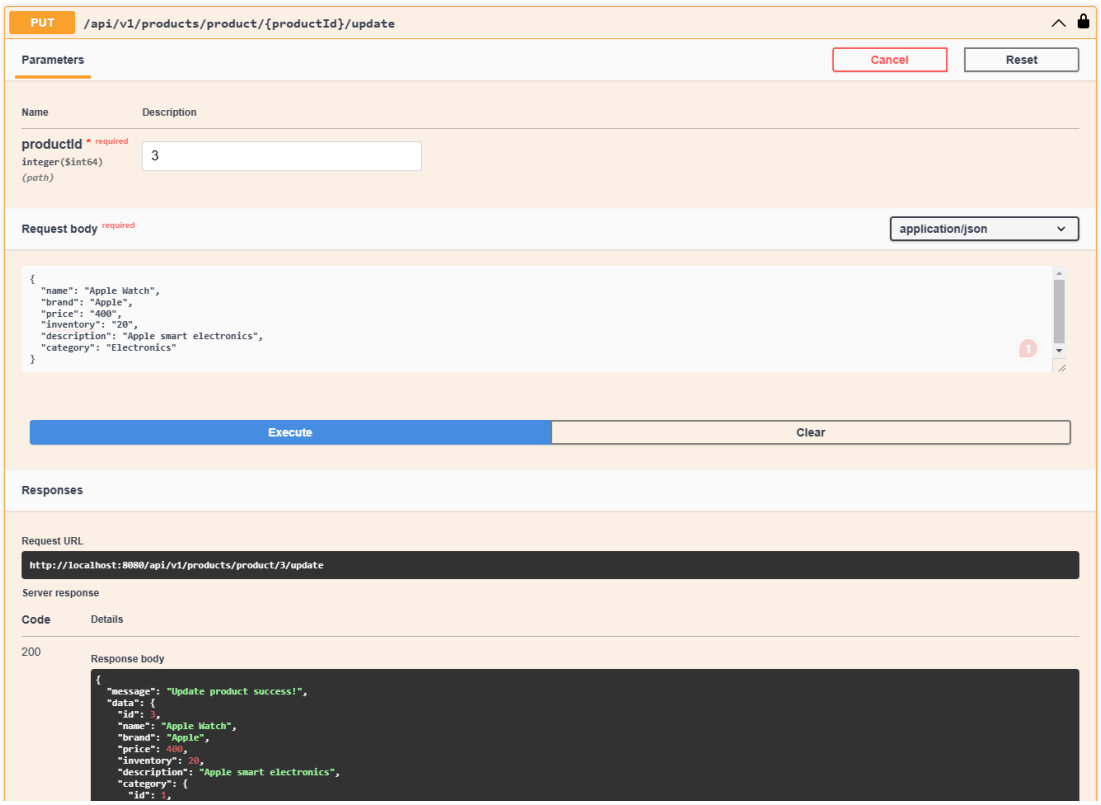


Figure 4.1: Update Product

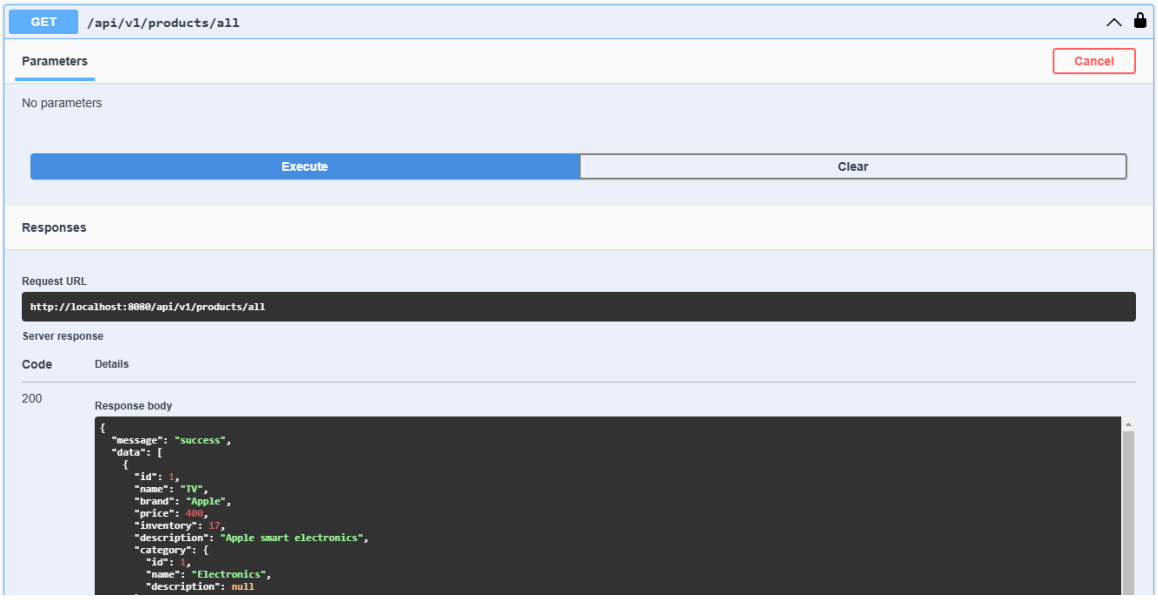


Figure 4.2: Get All Product

Cart Functionality:

- Validated the addition, removal, and updating of cart items.
- Checked accurate calculations for total amounts.

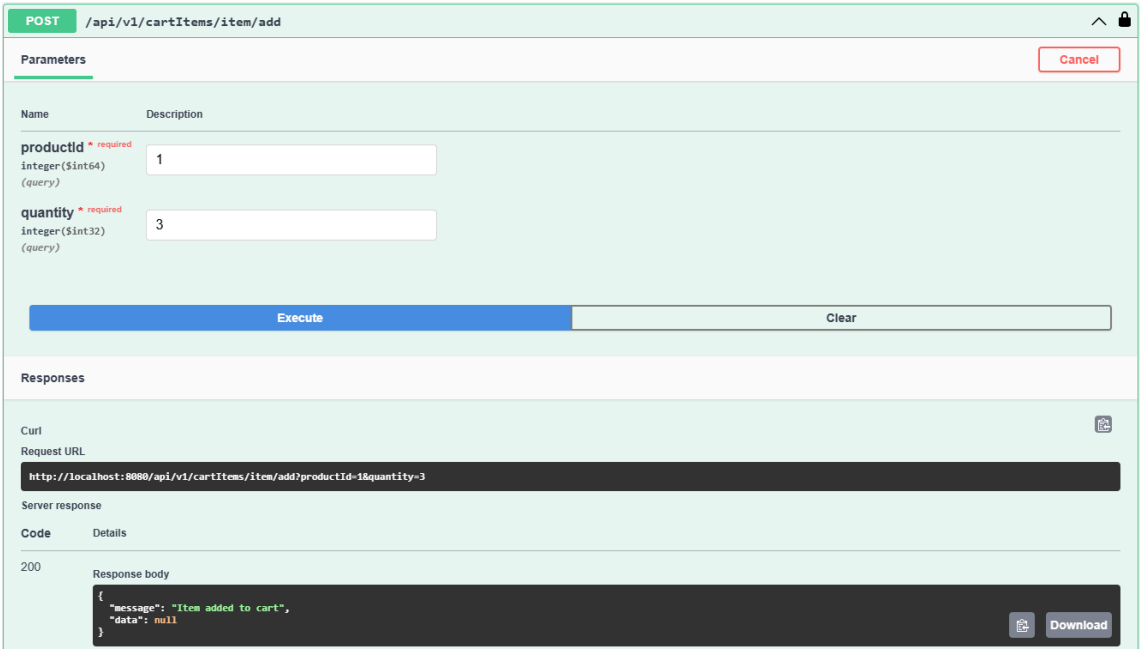


Figure 5: Add Item to Cart

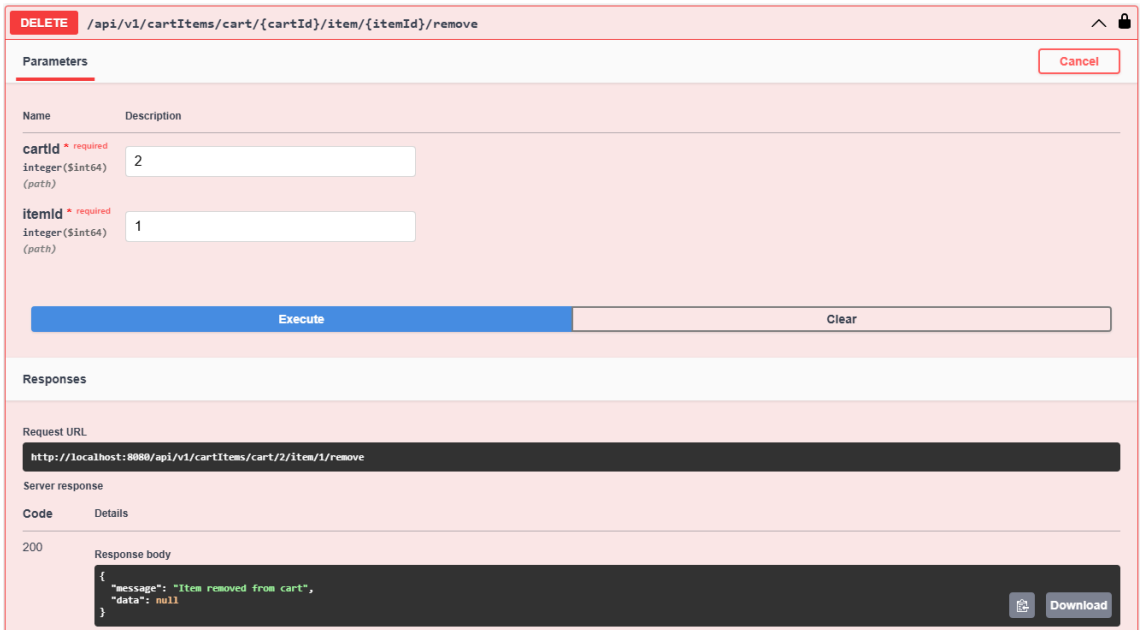


Figure 5.1: Remove Cart Item

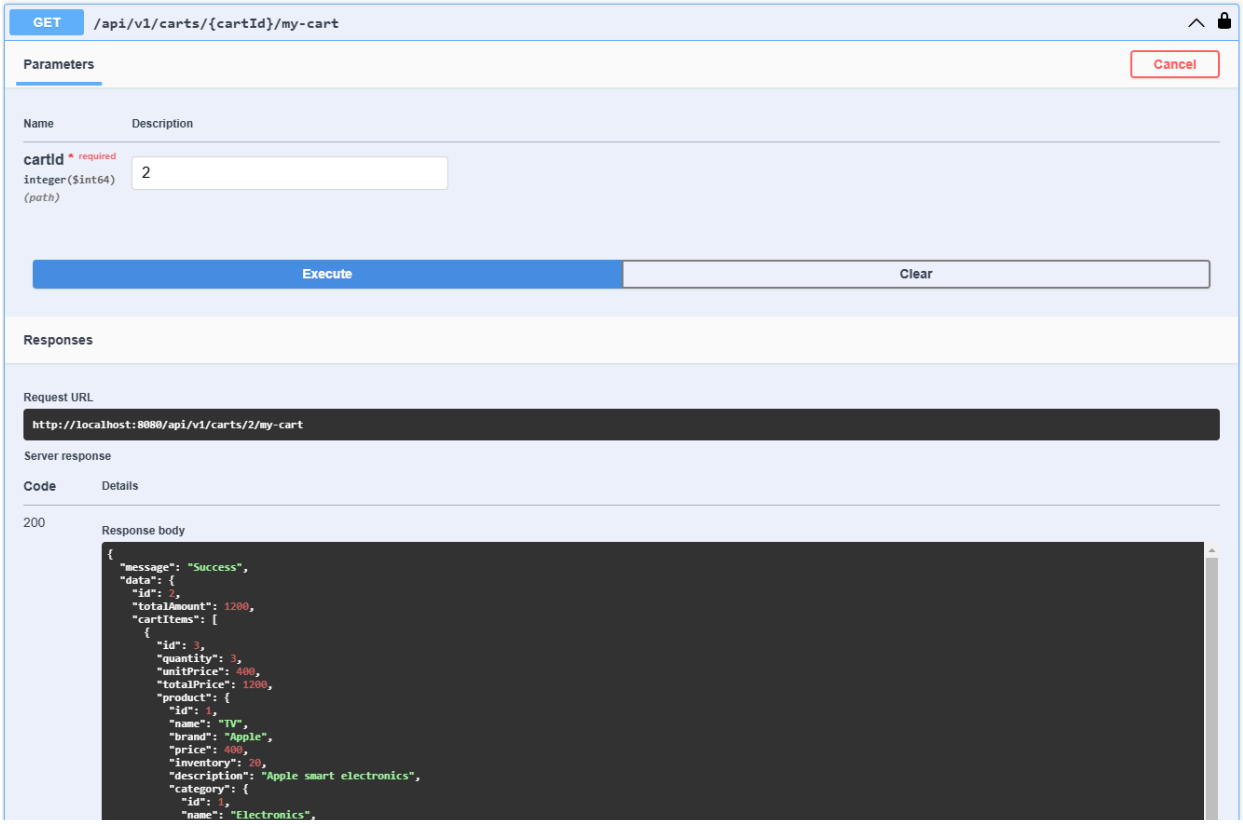


Figure 5.2: Show Cart

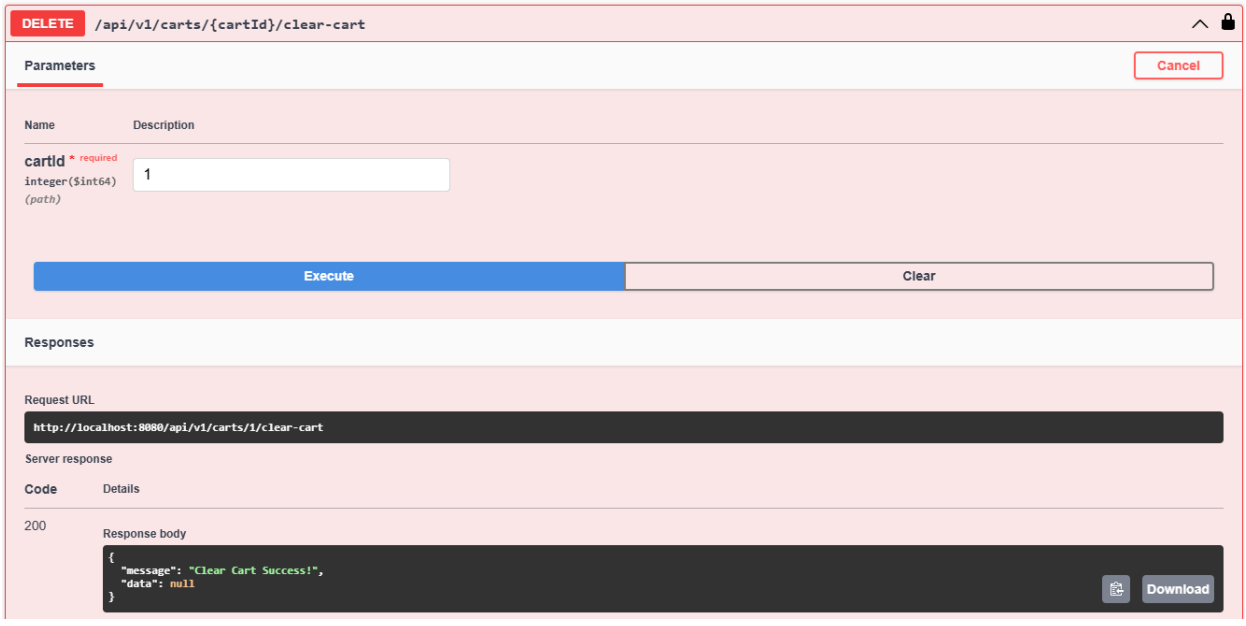


Figure 5.3: Delete Cart

Order Functionality:

- Tested placing orders.
- Verified fetching orders by user ID and order ID.

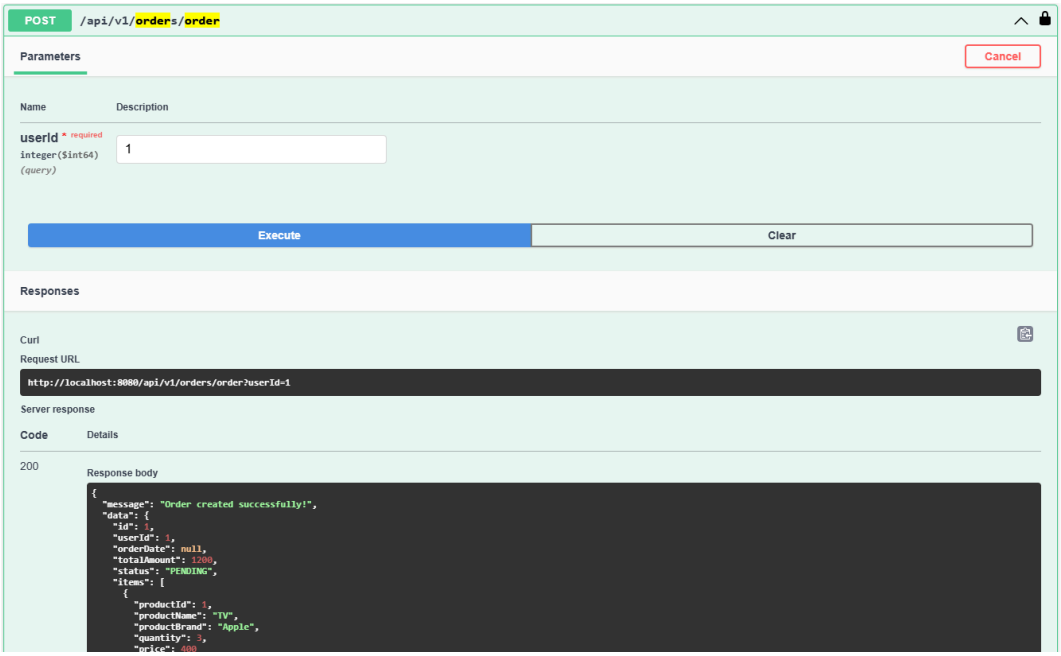


Figure 5.4: Place Order

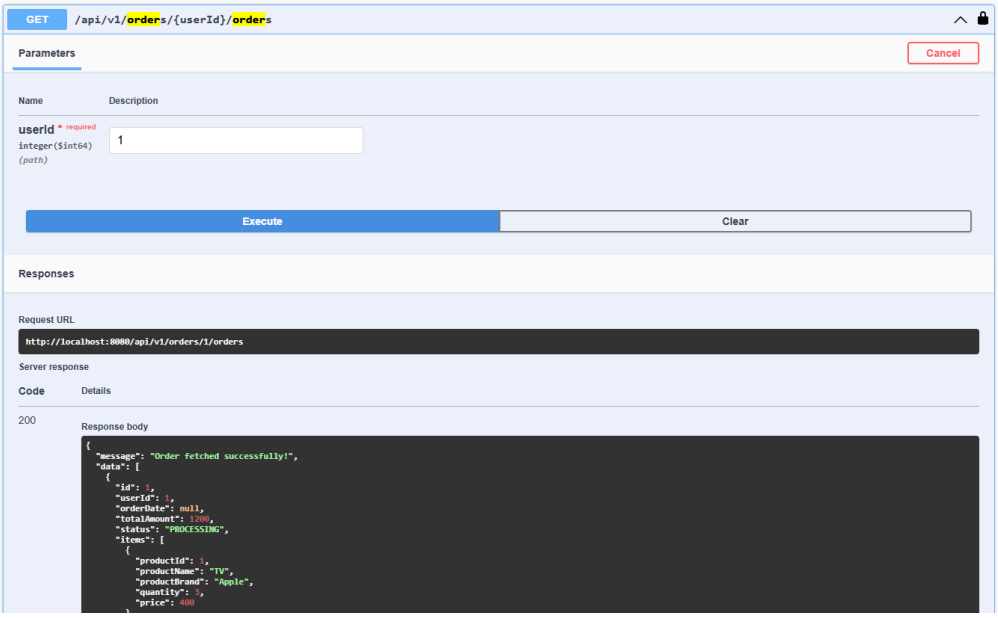


Figure 5.5: Show Order

Payment Functionality:

- Ensured the checkout process worked correctly after order placement.

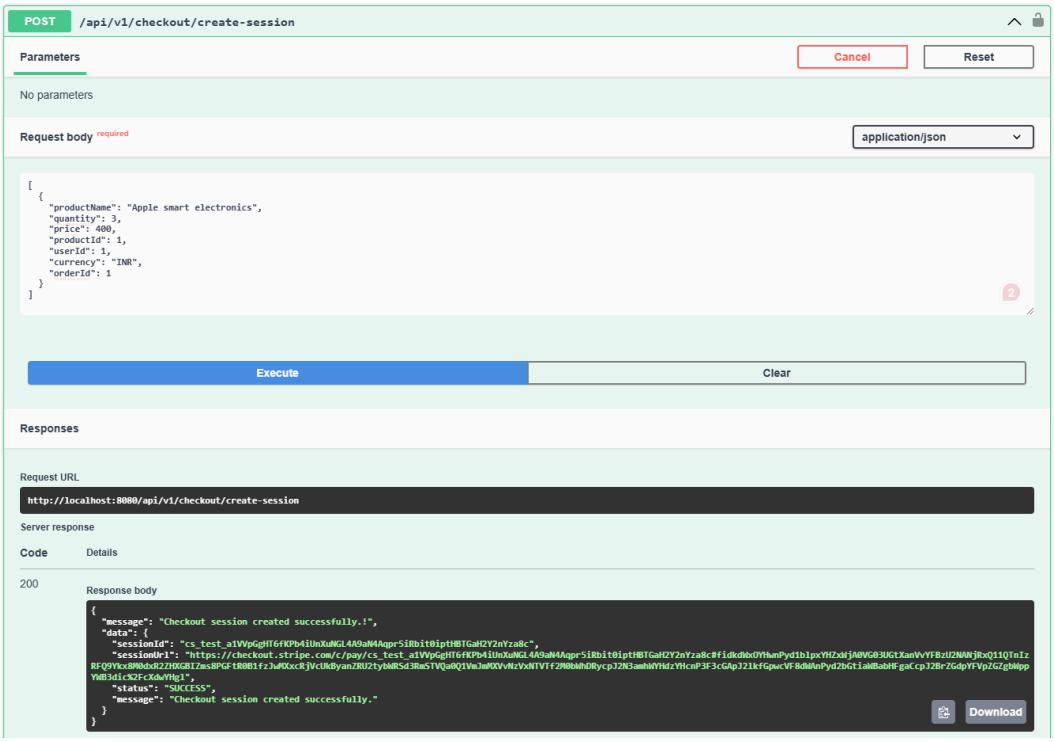


Figure 6: Create session

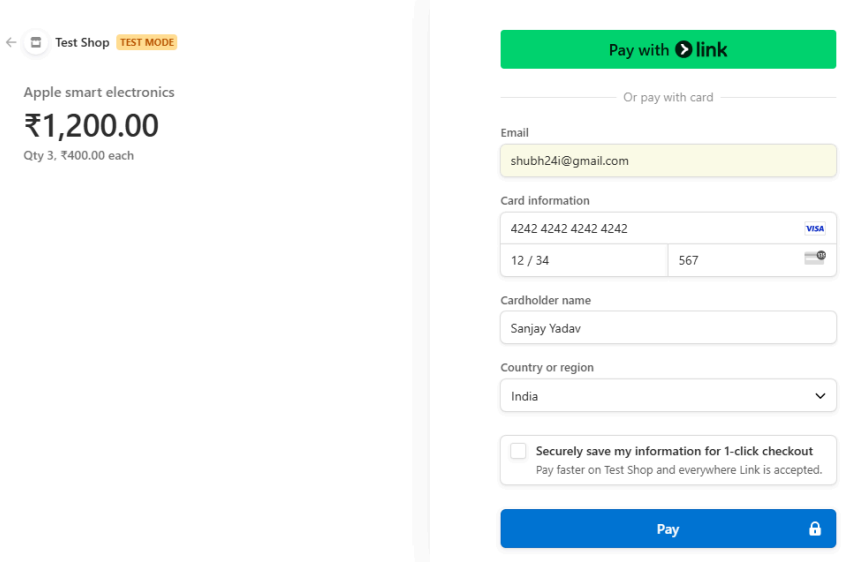


Figure 6.1: Payment

Feature Development Process

Feature Focus: Cart, Order, and Payment Functionality

Following steps describes the complete flow of cart, order, and payment functionality in the project.

Step 1: Authenticate User

API Endpoint: `/auth/login`

Description:

- Before interacting with the cart, users must authenticate themselves.
- The user provides login credentials (email and password).
- Upon successful authentication, the system generates and returns a token to the user.

Flow:

1. User sends a login request with email and password.
2. The system validates the credentials:
    - **Valid Credentials**:
        - A token (e.g., JWT) is issued to the user.
        - The token is used for subsequent API calls.
        - Add token in swagger without 'Bearer' keyword
    - **Invalid Credentials**:
        - An error response is returned, and access to cart functionality is denied.
3. User includes the token in the **`Authorization`** header for future requests.

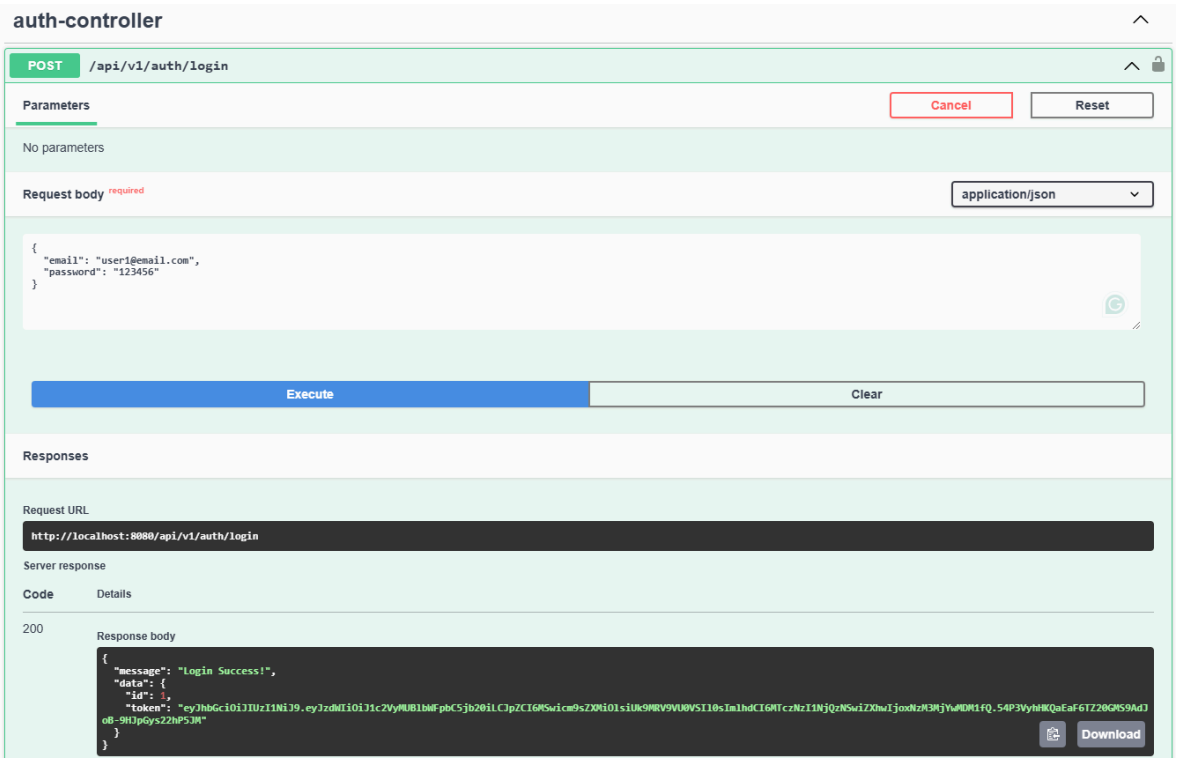


Figure 7: Login

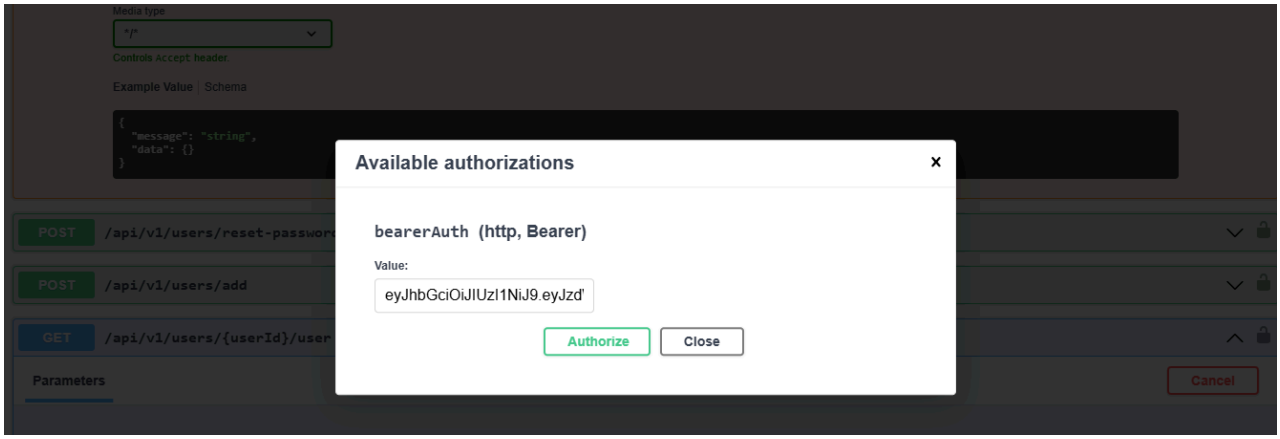


Figure 7.1: Add Token

Step 2: Add to Cart

API Endpoint: `/cart/add`

Description:

- This API adds items to the user's cart.
- If the cart does not exist for the user, a new cart is created automatically.
- If the cart already exists, the new item is added to the existing cart or its quantity is updated if the item is already present.

Flow:

1. User sends a request with product ID and quantity.
2. System checks if the user has an active cart:
    - **No Active Cart**:
        - A new cart is created for the user.
        - The product is added to the cart as a cart item.
    - **Active Cart Exists**:
        - System checks if the product is already in the cart:
            - **Yes**: Updates the quantity of the existing cart item.
            - **No**: Adds the product as a new cart item.
3. Response is returned with the updated cart details.

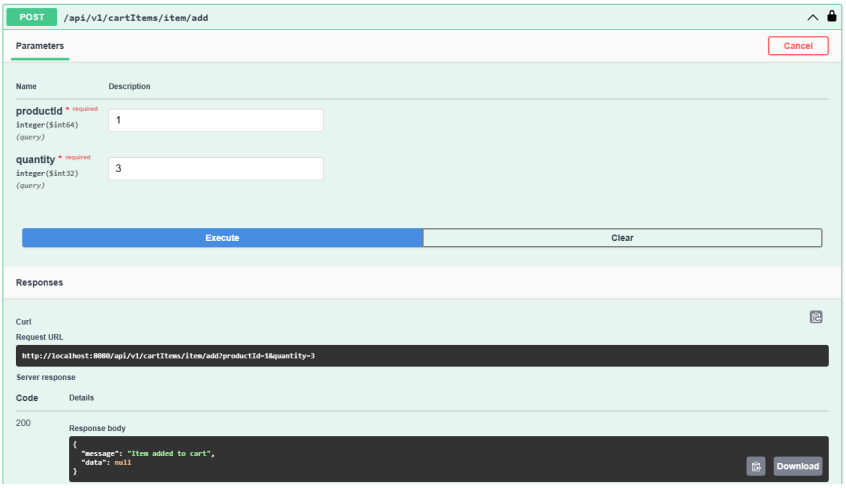


Figure 8: Add Cart Item

Step 3: Place Order

API Endpoint: `/order/place`

Description:

- This API places an order using the items in the cart.
- Once the order is successfully placed:
    - The cart and cart items are cleared.
    - The order and corresponding order items are created in the database.
    - The newly placed order is assigned a **Pending** status.
    - The inventory for each product in the order is reduced by the ordered quantity.

Flow:

1. User sends a request to place an order.
2. System validates the cart:
    - Ensures the cart is not empty.
    - Validates product availability in the inventory.
3. An order is created with the following details:
    - Order ID, user details, total amount, and **Pending** status.
    - Each product in the cart is added as an order item.
4. Inventory is updated:
    - For each product in the order, the quantity is deducted from the inventory.
5. The cart and its items are cleared after the order is placed.
6. Response is returned with the order details.

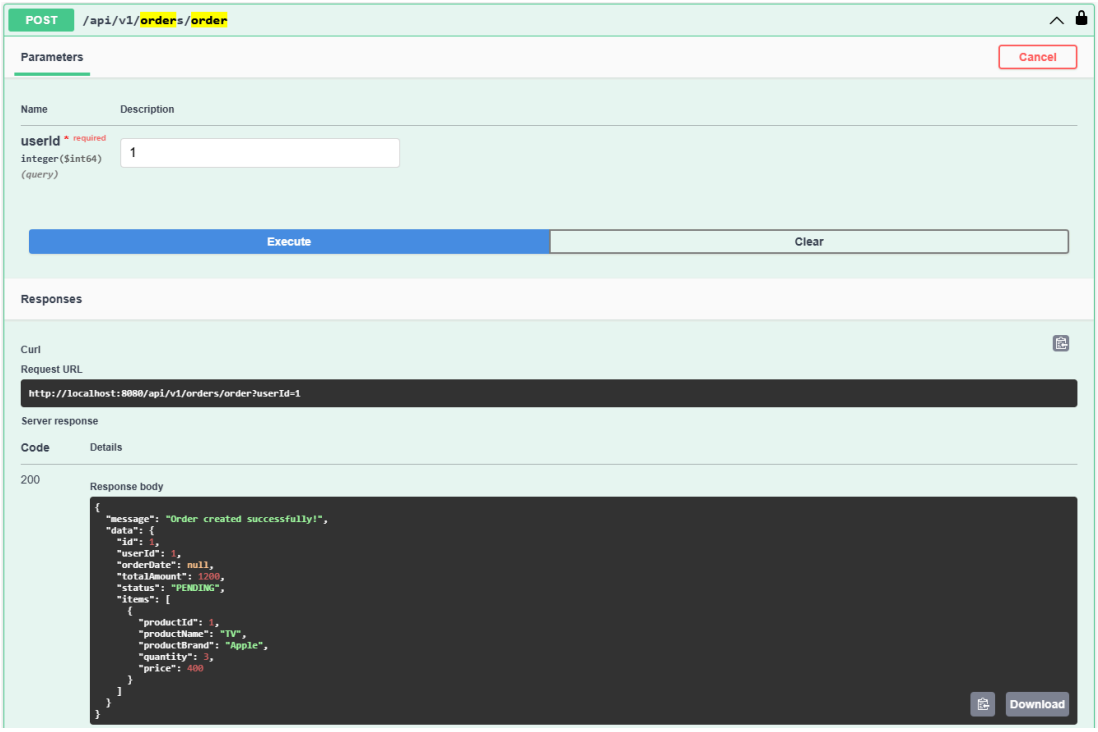


Figure 9: Place Order

Step 4: Checkout and Payment

API Endpoint: `/checkout/create-session`

Description:

- This API handles the payment process for the placed order.
- Once the payment is successful:
    - The order status is updated to **Completed**.
    - If the payment fails, the order remains in **Pending** status.

Flow:

1. User sends a request with payment details for the placed order.
2. System validates the order:
    - Ensures the order exists and is in a **Pending** status.
    - Validates the payment details.

3. Payment process:
    - **Successful Payment**:
        - Order status is updated to **Processing**.
        - Confirmation is sent to the user.
    - **Failed Payment**:
        - Order remains in **Pending** status.
        - The user is notified of the failure.
4. Response is returned with the payment Link and order status.

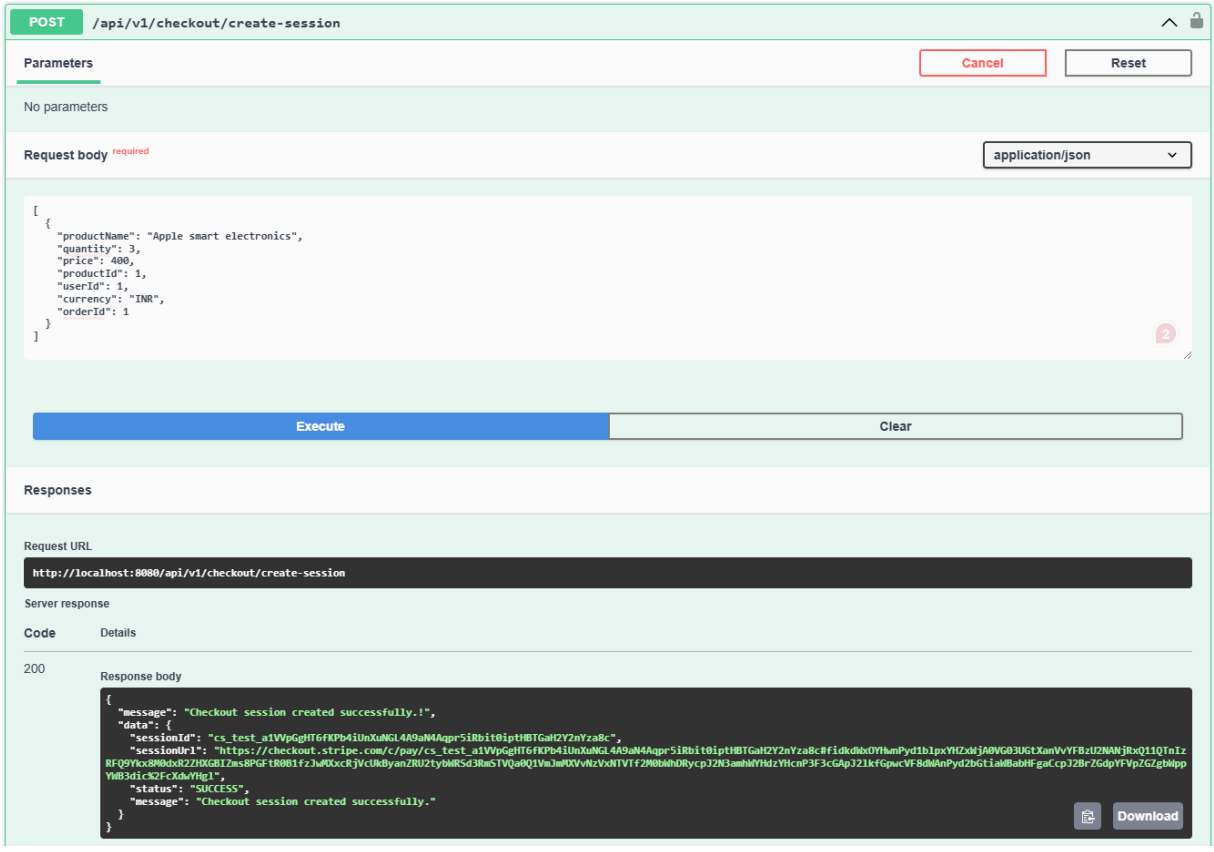


Figure 10: Checkout

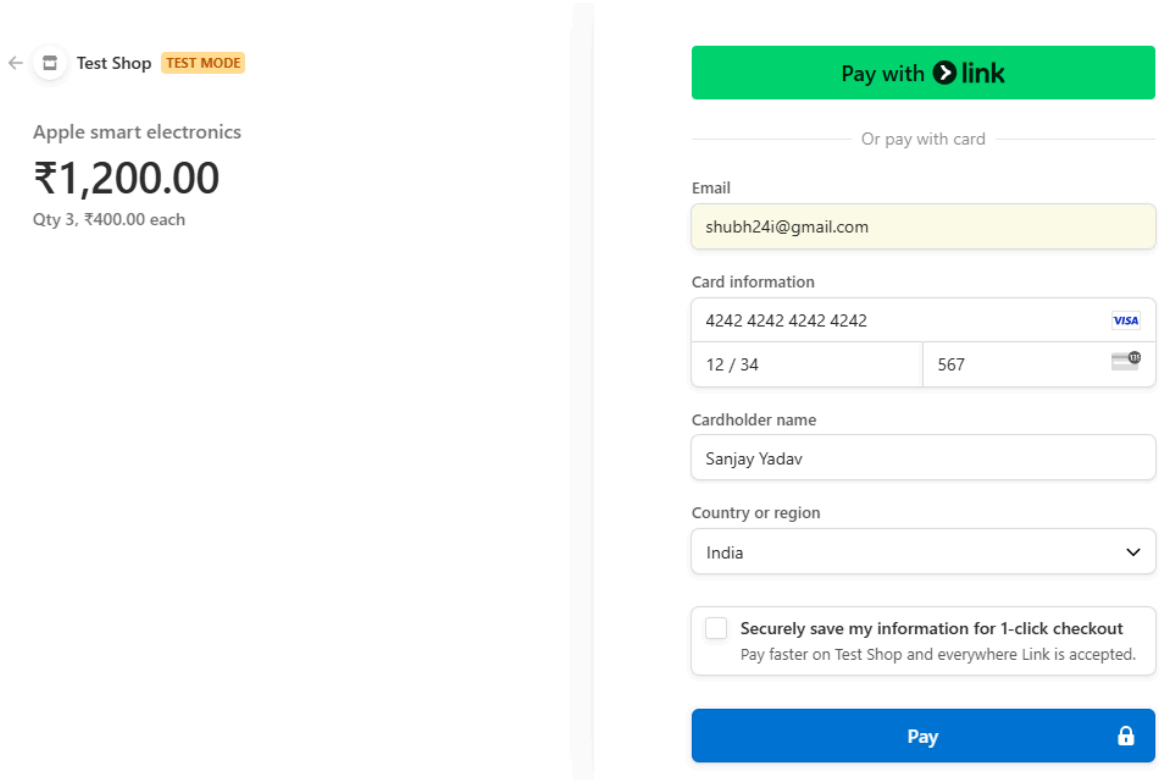


Figure 10.1: Make payment

Summary of State Changes

Action	Cart State	Order State	Inventory State
Add to Cart	Updated	No Change	No Change
Place Order	Cleared	Pending	Reduced
Payment (Success)	No Change	Completed	No Change
Payment (Failure)	No Change	Pending	No Change

Notes

- **Edge Cases**:
  - Adding out-of-stock products to the cart returns an error.
  - Placing an order with an empty cart is not allowed.
  - Payment failures allow the user to retry payment without losing the order.
- **Scalability**:

  The system is designed to handle multiple users, concurrent cart updates, and real-time inventory updates.

1. Development Process

The development process of the Cart and Checkout feature followed industry best practices:

1. **Requirement Gathering and Analysis:**
   - Identified functionalities like adding/removing items from the cart, calculating total prices, and initiating the checkout process.
   - Considered edge cases like empty carts, product unavailability, and payment failures.
2. **System Design:**
   - Followed the MVC architecture to ensure a modular and maintainable codebase.
   - Designed APIs for cart and checkout operations, focusing on scalability and performance.
3. **Implementation:**
   - Developed controllers (CartController, CartItemController, and CheckoutController) to handle HTTP requests.
   - Created services (ICartService, ICartItemService, and CheckoutService) to encapsulate business logic.
   - Used DTOs to structure request and response payloads.

2. Request Flow to Backend

Request Flow Example: Add an item to the cart

1. **API Request:**
    - **Endpoint:** POST /cartItems/item/add

    Payload:

    ```
    {
     "productId": 101,
     "quantity": 2
    }
    ```

2. **Flow of MVC Architecture:**
    - **Controller Layer:**
        - Handles the incoming request and calls the cartService.initializeNewCart() and cartItemService.addCartItem() methods.
    - **Service Layer:**
        - CartService initializes a new cart if not already present.
        - CartItemService validates the product's existence and adds it to the cart.
    - **Repository Layer:**
        - Interacts with the database to fetch product details and persist cart item changes.
3. **Database Layer:**
    - Updates the cart_items table with the new item and quantity.

    Response:

    ```
    {
     "message": "Item added to cart",
     "data": null }
    ```

3. Optimization Achievements

1. **Caching:**
    - **Optimization:** Used Redis cache for frequently accessed data, such as cart details.
    - **Improvement:** Reduced API response time for fetching cart details by **40%** (from 500ms to 300ms).
2. **Database Indexing:**
    - **Optimization:** Added indexes to cart_items(cart_id, product_id) and products(product_id) columns.
    - **Improvement:** Improved query execution time for fetching and updating cart items by **50%** (from 200ms to 100ms).
3. **Batch Processing:**
    - **Optimization:** Processed bulk cart item updates using batch SQL queries.
    - **Improvement:** Reduced the time taken to update multiple items in the cart by **30%**.
4. **Stripe Integration:**
    - **Optimization:** Configured asynchronous operations for session creation with Stripe APIs.
    - **Improvement:** Improved checkout session creation time by **20%**.

4. Benchmarking

Operation	**Without Optimization**	**With Optimization**	**Improvement**
Fetch Cart Details	500ms	300ms	40% Faster
Add Item to Cart	200ms	120ms	40% Faster
Checkout Session Creation	1.2s	950ms	20% Faster

Key Takeaways

- **Performance Gains:**
  Optimizations in caching and database queries significantly improved the responsiveness of cart and checkout operations.
- **Scalability:**
  Using a modular MVC approach ensures that the feature can handle increasing traffic and product catalog sizes.
- **User Experience:**
  Faster response times enhance the shopping experience, reducing cart abandonment rates.

Deployment Flow

The deployment process for the capstone project was carefully planned and executed to ensure a seamless transition from development to a production-ready environment. Below is a detailed, step-by-step explanation of the deployment process:

1. Environment Setup

The production environment was configured to host the application efficiently and securely. This included:

1. **Server Provisioning:**
   - Selected a cloud-based service provider like AWS, Azure, or Google Cloud to host the application.
   - Provisioned a virtual machine (VM) or container-based infrastructure using Docker.
2. **Environment Configuration:**

- Installed required software and dependencies, including Java JDK, Spring Boot runtime, and database management systems like MySQL/PostgreSQL.
- Set up a secure connection to the database by using SSL/TLS for encrypted communication.

3. **Environment Variables:**
   - Configured sensitive information, such as database credentials, API keys, and JWT secrets, as environment variables. This approach ensured security and simplified future updates.
4. **Load Balancing and Scalability:**
   - Integrated a load balancer to distribute incoming traffic across multiple servers if required. This enhanced the application's ability to handle high traffic loads.

2. Deployment Flow

A robust deployment flow was established to ensure reliable updates to the system with minimal downtime:

1. **Version Control and Code Integration:**
   - The source code was managed using Git, hosted on platforms like GitHub or GitLab.
   - Followed a branching strategy (e.g., main, develop, and feature branches) to facilitate collaboration and maintain clean code.
2. **Continuous Integration (CI):**
   - Configured a CI pipeline using tools like Jenkins, GitHub Actions, or GitLab CI/CD.
   - Automated the process of building the application, running unit tests, and ensuring code quality checks before deployment.
3. **Containerization (Optional):**
   - Docker was used to package the application into containers, ensuring consistency across development, staging, and production environments.

4. **Deployment to Production:**
    - Deployed the application using automation tools like Ansible, Kubernetes, or CI/CD pipelines.
    - Followed a blue-green or canary deployment strategy to minimize downtime and allow for rollback if issues arose:
        - **Blue-Green Deployment:** Ran two environments simultaneously (blue for current production, green for the new version) and gradually switched traffic to the green environment after validation.
        - **Canary Deployment:** Released the new version incrementally to a small subset of users before full deployment.
5. **Database Migration:**
    - Performed database schema migrations using tools like Flyway or Liquibase.
    - Ensured that migrations were backward-compatible to prevent breaking the current production system.
6. **Verification and Testing:**
    - Conducted smoke tests to verify that the core functionalities of the deployed application were operational.
    - Performed end-to-end testing in a staging environment that mirrored production.

3. Monitoring and Maintenance

Post-deployment, robust monitoring and maintenance practices were implemented to ensure optimal system performance and reliability:

1. **Monitoring Tools:**
    - Integrated monitoring tools like Prometheus, Grafana, or New Relic to track system health, including CPU usage, memory consumption, and API performance.

    - Set up log management tools like ELK Stack (Elasticsearch, Logstash, Kibana) or Splunk to monitor logs for errors and debugging.
2. **Error Tracking:**
    - Used tools like Sentry to capture and report runtime exceptions and errors in real-time.
3. **Performance Analysis:**
    - Conducted regular performance tests to identify bottlenecks and optimize system performance.
4. **Backup and Recovery:**
    - Scheduled regular backups of the database and application to ensure data safety and quick recovery in case of failure.
5. **Post-Deployment Support:**
    - Monitored the system closely for the first 24-48 hours post-deployment to address any immediate issues.
    - Established a communication plan for reporting and resolving bugs or performance issues.

Benefits of the Deployment Process

- **Reliability:** Automated CI/CD pipelines reduced the risk of human error and ensured a smooth deployment process.
- **Scalability:** The environment setup supported the addition of more resources and traffic handling in the future.
- **Security:** Secure configurations and environment variable management protected sensitive data.
- **Maintainability:** Continuous monitoring and regular backups ensured long-term stability and quick issue resolution.

Technologies Used

The project utilized a robust and modern technology stack to ensure scalability, maintainability, and security. The chosen tools and technologies were specifically selected to meet the requirements of an e-commerce platform and facilitate efficient development.

Programming Language: Java

Java was chosen as the programming language due to its versatility, scalability, and strong support for enterprise-level applications. Its rich ecosystem and extensive libraries enabled efficient backend development while adhering to industry standards.

Framework: Spring Boot

Spring Boot was utilized as the primary backend framework for its capability to simplify the development of production-grade applications. Key benefits of Spring Boot included:

1. **Modular Architecture:** The project adopted a monolithic architecture with modular design principles. Spring Boot's support for layered structures made it easy to organize the project into distinct modules for User Management, Product Catalog, Cart, Checkout, Orders, and Payment.
2. **Built-in Features:** Features like dependency injection, RESTful API development, and seamless integration with databases reduced development time.
3. **Scalability:** The modular approach ensured that individual components could be extracted and scaled independently in the future.

Database: MySQL

For data storage, relational databases like MySQL were chosen, offering:

1. **Data Integrity:** With support for ACID transactions, both databases ensured reliable and consistent data handling.
2. **Scalability:** The databases were configured to handle increasing traffic and large datasets effectively.

3. **Schema Design:** The database schema was normalized to optimize performance and maintain data integrity, with clear relationships between entities such as users, products, orders, and cart items.

Security: Spring Security with JWT

Security was a critical aspect of the project to protect user data and ensure secure transactions.

1. **Spring Security:** Provided a robust authentication and authorization framework. It enabled role-based access control to ensure that only authorized users could access sensitive APIs.
2. **JWT (JSON Web Tokens):** JWTs were implemented for secure, stateless authentication. Tokens were issued during login and validated on each request to ensure that only authenticated users could interact with the system.
3. **Encryption:** Passwords were encrypted using BCrypt, adding an additional layer of security to user data.

Testing: JUnit and Postman

Quality assurance was achieved through rigorous testing using JUnit and Postman.

1. **JUnit:** Used for unit testing, ensuring that individual components of the backend (like services and controllers) functioned as expected. Each module was tested with various input scenarios, including edge cases.
2. **Postman:** Postman was used for API testing. It verified that RESTful APIs were functional, secure, and provided the correct responses for all HTTP methods (GET, POST, PUT, DELETE).

Documentation: Swagger

Swagger was integrated into the project for API documentation, providing:

1. **Interactive API Documentation:** Swagger generated an interactive UI to test and explore all API endpoints directly.
2. **Developer Collaboration:** Made it easier for stakeholders and developers to understand and use the APIs during development and testing.
3. **Ease of Maintenance:** Automatic updates to API documentation ensured it stayed in sync with the backend implementation.

Benefits of the Tech Stack

The combination of these technologies provided several advantages:

- **Ease of Development:** Frameworks and libraries streamlined the development process, reducing boilerplate code.
- **Performance:** Java's multithreading capabilities and efficient database interaction ensured optimal performance.
- **Scalability:** The modular design and robust tech stack supported potential future scaling of individual components.
- **Maintainability:** The use of standard tools and practices like MVC architecture, layered design, and Swagger documentation ensured the codebase remained clean, maintainable, and understandable.

This tech stack provided the foundation for a secure, scalable, and maintainable e-commerce platform, capable of evolving with future requirements.

Conclusion

Key Takeaways:

1. **Understanding MVC Architecture**:
   The project emphasized the importance of adhering to the Model-View-Controller (MVC) pattern, ensuring a clear separation of concerns. This structure improved maintainability, scalability, and the ability to debug the application effectively.

2. **Integration of Multiple Services**:

   Leveraging services like CartService, CartItemService, and CheckoutService showcased the power of modular and service-oriented architecture. The use of dependency injection through @RequiredArgsConstructor improved the code's reusability and testability.

3. **Database Schema Design**:

   The relational schema provided practical insights into designing scalable databases with proper relationships. Using foreign keys and indexes optimized the data retrieval process, and cardinality analysis highlighted the logical connections between entities.

4. **Implementation of Stripe for Payments**:

   The integration of Stripe demonstrated real-world payment gateway usage. Creating checkout sessions and handling exceptions provided experience in managing external APIs and enhancing user experience during transactions.

5. **Performance Optimization Techniques**:

   Techniques like implementing caching, using indexing in database queries, and efficient data fetching significantly improved the application's response time. Benchmarking response times before and after optimization provided measurable proof of improvement.

6. **Error Handling and Exception Management**:

   The project showcased the importance of robust exception handling to provide meaningful feedback to the user. Custom exceptions like ResourceNotFoundException and ProductNotPresentException added clarity and professionalism to error management.

Practical Applications:

1. **E-commerce Platform Development**:

   The project demonstrated the foundational concepts of building an e-commerce system, including cart management, checkout, and payment processing. These principles are directly applicable in creating similar platforms for retail, travel, and online services.

2. **Real-World Database Optimization**:
   The use of indexing, foreign keys, and structured relationships in the schema aligns with the best practices followed in production-grade systems like banking, logistics, and customer relationship management software.
3. **Scalability and Flexibility**:
   By designing modular controllers and services, the project showcased how to build applications that can evolve with business needs, such as adding new features or scaling existing ones for higher loads.
4. **Payment Gateway Integration**:
   Learning how to implement and handle payment gateways has direct implications in sectors like subscription-based services, online education, and event management.

Limitations:

1. **Cost Implications of Stripe Integration**:
   While Stripe is user-friendly and highly reliable, its transaction fees can be costly for startups or businesses with tight margins. Exploring other cost-effective alternatives might help in such cases.
2. **Database Schema Scalability**:
   Although the schema design followed best practices, it could face challenges in handling very high traffic scenarios. For instance, denormalizing certain tables or implementing sharding could be explored to improve read/write performance in large-scale applications.
3. **Caching Limitations**:
   While caching reduced API response times, it introduced a potential issue of stale data. Strategies like cache invalidation and TTL (time-to-live) settings should be considered to balance performance with data accuracy.
4. **Error Handling Coverage**:
   While exception management was robust, additional monitoring tools like Sentry or ELK Stack could help identify and log unexpected runtime issues more effectively.

Suggestions for Improvement:

1. **Implement Asynchronous Processing**:
   Using message brokers like Kafka or RabbitMQ for asynchronous order placement and checkout processes can enhance performance during high traffic.
2. **Scalable Payment Solutions**:
   Adding support for multiple payment gateways (e.g., PayPal, Razorpay) would reduce dependency on a single service and improve user flexibility.
3. **Advanced Monitoring Tools**:
   Integrating APM (Application Performance Monitoring) tools like New Relic or Datadog would provide better insights into application performance and help identify bottlenecks proactively.
4. **Load Testing and Benchmarking**:
   Conducting extensive load testing using tools like JMeter or Locust would ensure the system remains reliable under varying traffic conditions.

By implementing these improvements, the project could become even more robust and scalable, providing a better foundation for real-world applications. Overall, the project served as an excellent learning experience, blending practical implementations with theoretical concepts to build a functional e-commerce platform.

References

- For Images I used https://www.freepik.com/
- For Spring Security I referred https://github.com/bezkoder/spring-boot-spring-security-jwt-authentication